Fast polynomial multiplication using matrix multiplication accelerators with applications to NTRU on Apple M1/M3 SoCs

Décio Luiz Gazzoni Filho[1,2], Guilherme Brandão[3] and Julio López[1]

[1] Universidade Estadual de Campinas (UNICAMP), Instituto de Computação, Campinas, Brazil
[2] State University of Londrina, Department of Electrical Engineering, Londrina, Brazil
[3] Independent Researcher, Londrina, Brazil

Abstract. Efficient polynomial multiplication routines are critical to the performance of lattice-based post-quantum cryptography (PQC). As PQC standards only recently started to emerge, CPUs still lack specialized instructions to accelerate such routines. Meanwhile, deep learning has grown immeasurably in importance. Its workloads call for teraflops-level of processing power for linear algebra operations, mainly matrix multiplication. Computer architects have responded by introducing ISA extensions, coprocessors and special-purpose cores to accelerate such operations. In particular, Apple ships an undocumented matrix-multiplication coprocessor, AMX, in hundreds of millions of mobile phones, tablets and personal computers. Our work repurposes AMX to implement polynomial multiplication and applies it to the NTRU cryptosystem, setting new speed records on the Apple M1 and M3 systems-on-chip (SoCs).

Keywords: PQC · NTRU · Apple Silicon · Accelerators

1 Introduction

In the 1990s, Shor [Sho97] described an efficient quantum algorithm to solve hard problems (integer factorization and discrete logarithms) for classical computers, compromising many existing cryptosystems. *Post-quantum cryptography* (PQC) seeks to develop new public-key cryptosystems based on hard computational problems resistant to quantum attacks, such as those based on lattices. Research into efficient implementation flourished as multiprecision integer arithmetic gave way to polynomial multiplication modulo "small" (CPU word size) integers for lattice-based schemes, introducing new challenges and performance tradeoffs.

In their Turing lecture [HP19], computer architecture pioneers Hennessy and Patterson claim that "(a)n era without Dennard scaling, along with reduced Moore's Law and Amdahl's Law in full effect means inefficiency limits improvement in performance to only a few percent per year", suggesting that further hardware improvements must come from *domain-specific architectures* tailored to specific applications. This is seen in the addition of SIMD extensions and special-purpose instructions (such as for symmetric cryptography and binary polynomial multiplication) to CPUs, as well as increasingly popular accelerators such as programmable graphics processing units (GPUs), FPGAs and others. Such developments have opened up new research agendas for efficient implementation of cryptosystems.

The meteoric rise of artificial intelligence, machine learning and deep learning in the 2010s, and their demand for teraflops-level linear algebra performance, led to an introduction of domain-specific architectures such as tensor cores for NVIDIA's GPUs

E-mail: decio.gazzoni@ic.unicamp.br,dgazzoni@uel.br (Décio Luiz Gazzoni Filho), brandaogbs@gmail.com (Guilherme Brandão), jlopez@ic.unicamp.br (Julio López)

This work is licensed under a "CC BY 4.0" license.

Date of this document: 2024-01-10.

[MCL+18], IBM Power ISA's Matrix-Multiply Assist (MMA) [MBB+21], Intel's Advanced Matrix Extensions (AMX) [Int22], and ARMv9-A's Scalable Matrix Extensions (SME) [WMS22]. The first three have launched in production hardware in 2017, 2021 and 2023, respectively, whereas the latter, to our knowledge, hasn't shipped as of 2023. Although less publicized, a matrix multiplication accelerator has shipped in hundreds of millions of mobile phones, tablets and personal computers, specifically those made by Apple, starting with the 2019 A13 system-on-chip (SoC) used in the iPhone 11 [Rod20, Section 7.6]. It is a coprocessor named AMX (no relationship to Intel AMX), also present in newer generations of Apple SoCs, such as the M-series powering ARM-based Apple personal computers.

Our work investigates repurposing AMX to accelerate the core polynomial multiplication routine of lattice-based post-quantum cryptosystems, by implementing NTRU [HPS98], specifically the proposal [CDH+20] that advanced to round 3 of NIST's PQC standardization process [Nat17], for Apple's M1 and M3 SoCs. Despite NIST's preference for Kyber [ABD+19], NTRU is standardized by IEEE Std 1363.1 [Ins09] and ASC X9.98 [Ame17]. It is also representative of a wider class of lattice-based schemes based on NTT-unfriendly rings such as SABER [BMD+20] and FrodoKEM [BCD+16, ABD+21]; the latter is recommended by the German BSI [Bun21] and under standardization by ISO [Int23c]. In case of cryptanalytic attacks on Kyber, NTRU may be the target of renewed interest.

A potential roadblock is that AMX is undocumented by Apple; programmers are intended to access it via calls to Apple's Accelerate framework [App23], which provides accelerated versions of the industry-standard BLAS library [LHKK79] and the proprietary BNNS neural network API. Fortunately, extensive reverse-engineering efforts [Joh22b, Han23, Caw23] document AMX's instruction set and performance characteristics, allowing us to develop a full NTRU implementation with AMX-accelerated polynomial multiplication routines, as presented in this paper. The authors were not involved in these reverse-engineering efforts, working only with the information made public in these references.

Related works. Google's Tensor Processing Unit is described in [JYP+17]. It achieves a peak throughput of 92 TOPS/s and 15–30× faster neural network inference than CPUs and GPUs of that era. [MBB+21] reports a 2× speedup by using MMA over regular vector code for double-precision matrix multiplication in the POWER10 CPU.

Some works demonstrate speedups from NVIDIA tensor cores, including for PQC schemes [MCL+18, WZF+22, LSH+22, LSZH22]. However, GPU architectures, being high-throughput but high-latency, need aggressive batching (thousands to tens of thousands of parallel KEM/signature operations) to realize their performance potential, severely limiting applicable use cases. We view direct comparisons of batch and online implementations (for CPUs and, as we shall show, AMX) as unfair, and refrain from doing so in this paper.

We found no references to Apple AMX in the scientific literature, and Apple does not officially document AMX functionality or performance characteristics. We resort to reverse engineering efforts of [Joh22b, Han23, Caw23], which we summarize in Section 3.

Lastly, we consider ARMv8-A implementations of NTRU. Karatsuba and Toom-Cook algorithms are implemented in [NG21], achieving speedups on the M1 of up to 6.68× and 8.49× for encapsulation and decapsulation, respectively, versus the reference implementation. [CCHY23] set new speed records for the HPS2048677 and HRSS701 parameter sets using the TMVP algorithm. They also optimized polynomial inversion and constant-time sorting, speeding up key generation, encapsulation and decapsulation by 7.67×, 2.48× and 1.77×, respectively, compared to [NG21] on the Cortex-A72 core.

Our contributions. We present the first (to our knowledge) implementation targeting AMX in the scientific literature, and also the first cryptographic implementation on a CPU-coupled matrix multiplication accelerator; previous works targeted GPU tensor cores.

Unlike GPUs, CPU-coupled matrix multiplication accelerators (coprocessors or instruction set extensions) can achieve high throughput without sacrificing latency. Indeed, we set new NTRU speed records on Apple M1/M3 SoCs, outperforming state-of-the-art NEON implementations under conditions previously seen in CPU implementations only: latency (cycle count) for a *single* execution of the scheme's operations, with *no batching at all*.

SIMD extensions required a paradigm shift from a scalar to a vector (1D) view of computation; matrix multiplication accelerators further shift towards a matrix (2D) view. AMX's peak throughput is only achieved for its main matrix (outer product[1]) operation; vector operation throughput is similar to NEON. Also, its instruction set is rather limited, seemingly designed to implement only a restricted set of algorithms. Our main contribution is repurposing AMX for a new application (polynomial multiplication) while realizing its performance potential; this required a deep rethinking of polynomial multiplier architectures to expose two-dimensional parallelism, maximizing the ratio of matrix to vector operations to reap as much of its throughput as possible. We conjecture that the techniques we propose for AMX may be applicable to many similar accelerators soon to reach the market.

Our implementation is freely available at `https://github.com/...`[2]

2 Preliminaries

Notation. Let $a(x) = a_0 + a_1 x + \ldots a_{n-1} x^{n-1}$. A "slice" of the i-th through j-th coefficients ($j \geq i$) of $a(x)$ is written as $a_{i:j}(x) = a_i x^i + a_{i+1} x^{i+1} + \ldots + a_j x^j$. Boldface variable names refer to an associated row vector representation: $\mathbf{a}_{i:j} = [a_i, a_{i+1}, \ldots, a_j]$. Let $\mathbf{x}_{i:j} = [x^i, x^{i+1}, \ldots, x^j]$; then $a_{i:j}(x) = \mathbf{a}_{i:j} \mathbf{x}_{i:j}^\mathsf{T}$. Finally, $i : k : j$ represents ranges with a non-unit step of k; e.g. $\mathbf{a}_{i:k:j} = [a_i, a_{i+k}, a_{i+2k}, \ldots, a_j]$. We also combine slice notation with C's array indexing notation, i.e. $\mathtt{X}[i : j]$ selects elements of index $i, \ldots, j$ of $\mathtt{X}$.

The NTRU cryptosystem. Based on the pioneering work of [HPS98], NTRU [CDH+20] is a key encapsulation mechanism (KEM) merging the previous NTRUEncrypt [CHWZ17] and NTRU-HRSS-KEM [HRSS17b] proposals, providing the suggested NTRU-HPS and NTRU-HRSS parameter sets, respectively. Based on structured lattices, it uses polynomial arithmetic modulo $x^n - 1$ for prime n with coefficients reduced either modulo 3 or a power of two, q. The original cryptosystem [HPS98] is a partially correct, probabilistic public-key encryption (PKE) scheme. The NIST submission uses the techniques of [HPS96] to obtain a perfectly correct, deterministic PKE, and constructs a KEM using a generic transformation from this PKE [HHK17] which is IND-CCA2 secure in the random oracle model (ROM), and in the quantum-accessible ROM under a non-standard assumption.

Let $\Phi_1 = x - 1$ and $\Phi_n = x^{n-1} + x^{n-2} + \ldots + 1$, so that $\Phi_1 \Phi_n = x^n - 1$. Let $S_3 = \mathbb{Z}_3 / \Phi_n$; we define the canonical representative of an element in S_3, denoted as $\underline{S_3}$, as the polynomial of degree at most $n-2$ with coefficients in $\{-1, 0, 1\}$.

Algorithms 2.1, 2.2 and 2.3 define the NTRU PKE, where most of the execution time is spent; for a full description of the scheme, including the routine $\mathtt{Sample}_{f,g}$, see [CDH+20]. $\mathtt{Lift}(m)$, for $m(x)$ a polynomial, is defined as $\underline{S_3}(m)$ for HPS and $\Phi_1 \cdot \underline{S_3}(m/\Phi_1)$ for HRSS parameter sets. Algorithms 2.2 and 2.3 call for 1 and 3 polynomial multiplications in the HPS parameter sets, and `Lift` adds an extra multiplication for each in HRSS. Algorithm 2.1 appears to use 5 polynomial multiplications, but polynomial inversion modulo q in line 3 is usually realized as modulo-2 inversion followed by 4 Newton iterations to lift the result to $\mathbb{Z}_q/(\Phi_1 \Phi_n)$. This calls for an additional 8 multiplications, for a total of 13.

The submission includes four parameter sets: HPS2048509, HPS2048677, HPS4096821, and HRSS701. The first three use fixed-weight sample spaces proposed by Hoffstein, Pipher,

[1] Also known as a rank-1 update, implemented e.g. by the BLAS [LHKK79] level 2 subroutine `xGER`.
[2] A GitHub repository will be made available following the paper's publication.

Algorithm 2.1 `PKE.KeyGen`: PKE keypair generation

Input: seed
Output: $(\mathrm{sk} = (f, f_p, h_q), \mathrm{pk} = h)$
1: $(f, g) \leftarrow \texttt{Sample}_{f,g}(\mathrm{seed})$
2: $f_p \leftarrow (1/f) \pmod{3, \Phi_n}$
3: $v_1 \leftarrow 1/(3gf) \pmod{q, \Phi_n}$
4: $h \leftarrow (v_1(3g)^2) \pmod{q, \Phi_1\Phi_n}$
5: $h_q \leftarrow (v_1 f^2) \pmod{q, \Phi_1\Phi_n}$
6: **return** $(\mathrm{sk} = (f, f_p, h_q), \mathrm{pk} = h)$

Algorithm 2.2 `PKE.Enc`: PKE encryption

Input: $h, (r, m)$
Output: c
1: $m' \leftarrow \texttt{Lift}(m)$
2: $c \leftarrow r \cdot h + m' \pmod{q, \Phi_1\Phi_n}$
3: **return** c

Algorithm 2.3 `PKE.Dec`: PKE decryption

Input: $((f, f_q, h_q), c)$
Output: (r, m, fail)
1: $v_1 \leftarrow cf \pmod{q, \Phi_1\Phi_n}$
2: $m_0 \leftarrow v_1 f_p \pmod{3, \Phi_n}$
3: $m_1 \leftarrow \texttt{Lift}(m_0)$
4: $r \leftarrow (c - m_1)h_q \pmod{q, \Phi_n}$
5: **if** $(r, m_0) \in \mathcal{L}_r \times \mathcal{L}_m$ **then**
 ▷ See [CDH+20, Sections 1.2 and 1.3] for definitions of $\mathcal{L}_r$ and $\mathcal{L}_m$
6: **return** $(r, m_0, 0)$
7: **else**
8: **return** $(r, m_0, 1)$

Table 1: NTRU parameters and sizes (in bytes).

Parameter	HPS2048509	HPS2048677	HPS4096821	HRSS701
(n, q)	$(509, 2048)$	$(677, 2048)$	$(821, 4096)$	$(701, 8192)$
Key sizes (public, private)	699, 935	930, 1234	1230, 1590	1138, 1450
Ciphertext sizes	699	930	1230	1138

and Silverman [HPS96, HPS98]; the last uses arbitrary-weight sample spaces proposed by Hülsing, Rijneveld, Schanck, and Schwabe [HRSS17a]. Table 1 lists parameters and sizes.

3 Apple's AMX coprocessor

AMX is a coprocessor from Apple to accelerate matrix multiplication operations, first introduced in the Apple A13 SoC powering the iPhone 11 [Rod20]. All the information in this section comes from reverse engineering efforts and analyses of Apple patents by [Joh22b, Han23, Caw23], and is, by its nature, conjectural, although backed by extensive functional and performance experiments. Throughout this section, we focus on the M1 AMX unit, as we found no information about changes in the recently released M3, although our results of Section 5 indicate that M3 delivers increased AMX performance over M1.

3.1 Programmer's model

AMX exposes 80 64-byte registers, split into eight `X` and eight `Y` registers ($\texttt{X}_0$ to $\texttt{X}_7$ and $\texttt{Y}_0$ to $\texttt{Y}_7$), and 64 `Z` registers better viewed as rows of a matrix, denoted as $\texttt{Z}[0], \ldots, \texttt{Z}[63]$. Figure 2, redrawn from an Apple patent [SBG+16, Figure 2], is a visualization of intended register file organization. Some instructions can concatenate together either the `X` or `Y` registers for bytewise addressing as 512-byte circular buffers, for which we use the slice notation defined in Section 2. Figure 1 shows different `X`/`Y` register addressing notations.

`X` and `Y` are inputs and `Z` are outputs for most instructions, with a few exceptions. Data cannot be moved between CPU and AMX registers directly; it must go through memory.

Bytes	0	...	63	64	...	127	128	...	447	448	...	512
Registers	X_0, Y_0			X_1, Y_1			...			X_7, Y_7		
Slice notation (16-bit elements)	X[0 : 31] Y[0 : 31]			X[32 : 63] Y[32 : 63]			...			X[224 : 255] Y[224 : 255]		

Figure 1: Relationship between bytes, registers and slices in X and Y AMX registers.

	X[0]	X[1]	···	X[n]
Y[0]	Z[0][0] += Y[0]X[0]	Z[0][1] += Y[0]X[1]	···	Z[0][n] += Y[0]X[n]
Y[1]	Z[1][0] += Y[1]X[0]	Z[1][1] += Y[1]X[1]	···	Z[1][n] += Y[1]X[n]
⋮	⋮	⋮	⋱	⋮
Y[n]	Z[n][0] += Y[n]X[0]	Z[n][1] += Y[n]X[1]	···	Z[n][n] += Y[n]X[n]

Figure 2: AMX register file organization.

Different data types can be represented (8-, 16- or 32-bit integers, or 16-, 32- or 64-bit floating-point values). Input/output lane widths may be identical or mixed in specific combinations [Caw23]; in this paper we only use 16-bit inputs and outputs. The inputs to outer product operations (X_i and Y_i) are 32-element vectors of 16 bits each, resulting in a 32×32 output matrix of 16-bit elements; this is smaller than Z's available storage space (16,384 out of an available 32,768 bits). In AMX, each row of the output matrix is mapped to either the even or odd rows of Z, which are fully populated with output coefficients, as the size of a row of Z (512 bits) matches the size of a row of the output matrix.

CPUs in a cluster can share an AMX unit through per-CPU replication of architectural state [Joh22a, Han23]. Instructions are tagged with their source CPU to identify their copy of the state; thus, multiple CPUs may interleave instruction execution [Han23]. Special instructions (`set`/`clr`) control AMX's enabled/disabled status for each thread [Caw23].

3.2 Programming interface

AMX instructions are inserted into the CPU's instruction stream, and once no longer speculative, are dispatched to AMX via the CPU's store units [Han23]. They are encoded as A64 instructions within a reserved opcode space [Caw23]; given A64's fixed 32-bit instruction encoding, only 10 bits remain, 5 of which encode the instruction's opcode. The remaining 5 bits either encode the index of a scalar 64-bit register through which parameters are passed, or an immediate. This layout is shown in Figure 3. [Caw23] provides preprocessor macros to emit AMX instructions, with an intrinsics-like syntax.

31 … 10	9 … 5	4 … 0
0000 0000 0010 0000 0001 00	opcode	register/immediate

Figure 3: AMX instruction encoding.

3.3 Instruction set

We now list all known AMX instructions, reviewing those used by our polynomial multiplication implementation, as well as parameters of interest; for an exhaustive specification, see [Caw23]. We propose a taxonomy of instructions in Table 2, categorized by functionality.

Table 2: A taxonomy of AMX instructions. Opcodes given in base 10.

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

We first consider loads and stores. The least-significant 56 bits of their 64-bit argument are treated as a pointer to the source/target memory address; remaining bits encode parameters as per Table 3. While many other instructions can address `X`/`Y` registers at arbitrary byte positions, loads and stores require aligning to the start of a register (64-byte boundary); register indices are encoded in bits 58–56 of the argument (or 61–56 for `Z` rows). Memory accesses can be 64 bytes (bit 62 = 0) or 128 bytes (bit 62 = 1) wide.

Table 3: Parameters for AMX load and store instructions.

Bits	`ldx`, `ldy`, `stx`, `sty`	`ldz`, `stz`
63		
62	Load/store single register (0) or pair of consecutive registers (1)	
61–59		Z-row index (0 to 63)
58–56	Register index (0 to 7)	
55–0	Least significant 56 bits of pointer	

We illustrate [Caw23]'s macros with an example of a load from an array with base address `v` to register $\mathtt{X}_4$. We show a version using just their macros, a second using helper macros defined by us, and a simplified notation used in the algorithms of Section 4:

C notation	C notation, our helper macros	Algorithm notation
`AMX_LDX((uint64_t)v | 4 << 56);`	`AMX_LDX(AMX_PTR(v) | LDX_REG(4));`	$\mathtt{X}_4 \leftarrow \mathtt{ldx}(\mathtt{v}[0:31])$

We now briefly review other instructions, listing only functionalities of interest. Due to the default role of `X` and `Y` as input and `Z` as output registers, at times data must be moved from `Z` to `X` or `Y`, for which we use the `extrh` instruction. Its basic operation is copying a chosen row of `Z` to a byte-addressable 64-byte slice of either `X` or `Y`; it also supports other functionalities not required by us. For a visualization of `extrh` in use, refer to Figure 4. Our algorithmic notation for an example operation of extracting `Z`[63] to $\mathtt{Y}_0$ is:

$$Y_0 \leftarrow \mathtt{extrh}(\mathtt{Z}[63])$$

The main AMX arithmetic instruction we use is `mac16`, which supports *vector* and *matrix* modes. Let $\mathbf{x}, \mathbf{y}, \mathbf{z}$ be 32-element vectors of 16-bit integers. Vector-mode `mac16` computes pointwise multiply-accumulates (MACs): $\mathbf{z} \leftarrow \mathbf{z} + \mathbf{x} \circ \mathbf{y}$, where $+$ is vector addition and $\circ$ is the Hadamard (pointwise) product. Concretely, $\mathbf{z}$ is a row of `Z`, and $\mathbf{x}$ and $\mathbf{y}$ are 32-element slices of `X` and `Y`. Each of $\mathbf{x}, \mathbf{y}, \mathbf{z}$ can be optionally "skipped" to realize

different operations: vector addition ($\mathbf{z} \leftarrow \mathbf{z} + \mathbf{x}$ or $\mathbf{z} \leftarrow \mathbf{z} + \mathbf{y}$), Hadamard product without accumulation ($\mathbf{z} \leftarrow \mathbf{x} \circ \mathbf{y}$), or copying $\mathbf{x}$ or $\mathbf{y}$ to $\mathbf{z}$, i.e. the `extrh` instruction in reverse. We now define an algorithmic notation, using an example computation of $\mathbf{z} \leftarrow \mathbf{z} + \mathbf{x}$ (skipping the $\mathbf{y}$ input), in which we choose $\mathbf{z}$ to be the first row (i.e. that of index 0) of Z and $\mathbf{x}$ to be elements 64 to 95 of X (i.e. bytes 128 to 191, or equivalently, the full $\mathtt{X}_2$ register):

$$\mathtt{Z}[0] \leftarrow \mathtt{mac16}(\mathtt{Z}[0] + \mathtt{X}[64:95])$$

Let $\mathbf{Z}$ be a 32×32 matrix, and $\mathbf{x}$ and $\mathbf{y}$ be 32-element (row) vectors. Matrix-mode `mac16` realizes an outer product operation with accumulation, as shown in Figure 2: $\mathbf{Z} \leftarrow \mathbf{Z} + \mathbf{y}^T\mathbf{x}$. Concretely, the operands $\mathbf{x}, \mathbf{y}, \mathbf{Z}$ are mapped to slices of X and Y, and even or odd rows of Z, respectively. As with vector-mode `mac16`, operands can be skipped, although skipping $\mathbf{Z}$ appears to be the only sensible possibility, which disables accumulation. It is also possible to partially skip either the X or Y registers (i.e. select only a range of elements within each register for the computation). Consider an example operation of computing the outer product $\mathtt{Y}_0^{\mathsf{T}}\mathtt{X}_0$ (i.e. elements 0 to 31 of each of these registers), which are accumulated with, and saved to, the even rows of Z, while updating only columns 0 to 15 of the output matrix, using the partial X skip feature. Our algorithmic notation for this operation is:

$$\mathtt{Z}[0:2:62] \leftarrow \mathtt{mac16}(\mathtt{Z}[0:2:62] + \mathtt{Y}_0^{\mathsf{T}}\mathtt{X}_0,\ \text{columns} = 0:15)$$

When the “columns” parameter is omitted, it is understood that all columns of the output matrix are updated. It should be clear, from the operation performed, whether vector- or matrix-mode `mac16` is intended. A caveat: although within the context of AMX, the Y registers are best understood as column vectors, our choice of notation takes $\mathbf{a}$ to be a row vector, and thus $\mathbf{a}^{\mathsf{T}}$ as a column vector, throughout the paper. We extend this convention to AMX's X and Y registers, and thus write outer products in the form $\mathtt{Y}_i^{\mathsf{T}}\mathtt{X}_j$.

Finally, `vecint` is a vector (pointwise) instruction which generalizes vector-mode `mac16`; in particular, a single instruction can compute $\mathbf{z} \leftarrow \mathbf{z}+\mathbf{x}+\mathbf{y}$, attaining twice the throughput as vector-mode `mac16`, a fact which we've used to speed up some of our algorithms. As an example, this is how we denote an example operation of 16-bit integer addition of elements 16 to 47 of the X register (i.e., bytes 32 to 95 of X, incorporating elements of both $\mathtt{X}_0$ and $\mathtt{X}_1$) with elements 0 to 31 of the Y register (i.e. $\mathtt{Y}_0$), accumulating to Z[1]:

$$\mathtt{Z}[1] \leftarrow \mathtt{vecint}(\mathtt{Z}[1] + \mathtt{X}[16:47] + \mathtt{Y}[0:31])$$

3.4 Performance considerations

The M1 has two AMX coprocessors, one per core cluster (performance and efficiency), implementing a second generation of AMX instructions [Caw23]. It runs at the same clock speed as the cores and accesses memory via the L2 cache [Joh22b]. It supports out-of-order execution independently of the CPU, with buffers for 28 to 32 operations [Joh22b].

Outer product latency and throughput is discussed in [Joh22b] for floating-point operations. The performance cluster's AMX unit has an array of pipelined MAC units with 4-cycle latency, capable of executing one 32- or 64-bit outer product operation per cycle, or a 16-bit operation every two cycles. To achieve maximum performance, inter-instruction data dependencies must be avoided; either by using multiple *accumulators*, i.e. different subsets of Z as a destination (such as even and odd rows), or by issuing instructions from different CPUs, due to independent per-CPU AMX register files (see Section 3.1).

AMX microbenchmarking code is supplied in [Caw23]. We have run this code on an M1-based laptop, and report selected throughput figures, restricted to single-threaded results, presumably running on the AMX performance-cluster unit. Multiply-accumulates (MACs) count as 2 operations, and we report best-case scenarios, often using multiple accumulators; performance may degrade, significantly in certain cases, if fewer accumulators are used.

Outer products achieve up to 3.05 TFLOPS/s throughput with 16-bit floating-point data or mixed-lane integer arithmetic (8- and 16-bit operands); throughput drops to 1.53 TOPS/s for pure 16-bit integer arithmetic. In vector mode, up to 381 GFLOPS/s is achievable with floating-point or mixed 8-/16-bit integer MACs, dropping to 191 GOPS/s for pure 16-bit integer arithmetic. Replacing MACs with additions or multiplications alone further halves throughput. As a comparison, peak NEON throughput on M1 performance cores for 16-bit integer data is 204.8 GOPS/s for MACs and 102.4 GOPS/s for additions or multiplications alone, based on the microarchitectural investigations of [Joh22a].

Thus, AMX's peak throughput is $\approx 8\times$ higher for matrix (outer product) operations versus vector operations, whose peak performance approaches NEON's. That is, *outer products are much cheaper than vector operations.* Throughout the paper, we consistently strive to reduce vector operation count, which are the bottleneck of our computations.

While our implementation takes the usual constant-time precautions (foregoing branches and memory accesses conditional/indexed on secret data), it implicitly relies on constant-time execution of AMX instructions. Recently, ARM and Intel have started guaranteeing data-independent timing for some instructions [ARM, Int23b, Int23a]. Given AMX's lack of official documentation, it is impossible to obtain such assurances, although experiments in Section 5.1 suggest AMX displays data-independent timing. The previous lack of constant-time guarantees for CPUs hasn't prevented implementors from striving for, and achieving in practice, implementations with apparent constant-time characteristics, and may have played a key part in incentivizing manufacturers to provide such guarantees. Our paper is a first step in that direction for CPU-coupled matrix-multiplication accelerators.

4 Implementing polynomial multiplication on AMX

We now describe our AMX implementation of schoolbook multiplication in $\mathbb{Z}_{2^{16}}/(x^n - 1)$.

4.1 Basic block: multiplication of 32-coefficient slices

The basic block of our implementation multiplies slices of 32 coefficients from each input polynomial. We define a specific notation for polynomial slices starting at indices that are a multiple of 32: $a_{32k:32k+31}(x) = a_{\mathbf{k}}(x)$ (note boldface $\mathbf{k}$). We also define a notation for the product of 32-coefficient polynomial slices: $c_{\mathbf{k},\mathbf{l}}(x) = a_{\mathbf{k}}(x)b_{\mathbf{l}}(x)$. To clarify, we have:

$$\begin{aligned} c_{\mathbf{k},\mathbf{l}}(x) &= (a_{32k}x^{32k} + \ldots + a_{32k+31}x^{32k+31})(b_{32l}x^{32l} + \ldots + b_{32l+31}x^{32l+31}) \\ &= c_{32(k+l)}x^{32(k+l)} + \ldots + c_{32(k+l)+62}x^{32(k+l)+62}, \end{aligned}$$

where $c_m = \sum_{i+j=m} a_i b_j$. As before, $\mathbf{c}_{\mathbf{k},\mathbf{l}}$ is the associated row vector representation. Throughout the rest of Section 4, we resort to examples using a hypothetical 1/8-size AMX unit for space reasons. Boldface indices will then refer to 4-coefficient slices as appropriate for these examples, rather than 32-coefficient slices for full AMX; thus, $\mathbf{a}_{\mathbf{k}}$ would refer to $\mathbf{a}_{4k:4k+3}$. It will be clear from the context whether we refer to 4- or 32-coefficient slices.

We first note that the outer product $\mathbf{b}_{\mathbf{l}}^{\mathsf{T}}\mathbf{a}_{\mathbf{k}}$ generates the required partial products; shifting each row produces the usual multiplication parallelogram. We then perform sum-reduction of partial products in each column (an operation which we refer throughout the rest of Section 4 as *flattening* to avoid confusion with polynomial reduction) to obtain $\mathbf{c}_{\mathbf{k},\mathbf{l}}$. This can be visualized by considering an analogous operation in a hypothetical 1/8-size AMX unit, operating on 4-coefficient slices starting at the constant coefficient:

	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			

We then flatten the parallelogram by sum-reduction of columns and multiplication by the corresponding power of x to obtain $a_3b_3x^6 + (a_2b_3 + a_3b_2)x^5 + \ldots + (a_0b_1 + a_1b_0)x + a_0b_0$.

On AMX, outer products are easily realized by loading $\mathbf{a_k}$ and $\mathbf{b_l}$ to slices of the X or Y registers, and executing the matrix-mode mac16 instruction. By default, results are accumulated with the current values of Z, although there is a flag to skip current contents of Z, so as to realize multiplication without accumulation. The result occupies half of the Z register file, either the odd or even rows of Z; we assume the latter for the sake of example.

To realize shifts and flattenings, we initialize the two ends of a 3-register-wide slice of X (say X_0 and X_2) as well as Z[1], the first odd row of Z, with zeros. For each even row of Z except the first, we use the extrh instruction to extract it to the remaining X register of the slice (X_1 in this example). Recall that AMX allows addressing any 64-byte slice of the full 512-byte X register; we use this feature to select slices containing the appropriate number of zero 16-bit values to realize the desired shifting. These slices are used as input to a pair of vecint or vector-mode mac16 instructions, performing additions only, in order to accumulate the least significant coefficients to Z[0], and the most significant coefficients to Z[1]. Figure 4 illustrates this operation, again on a hypothetical 1/8-size AMX unit.

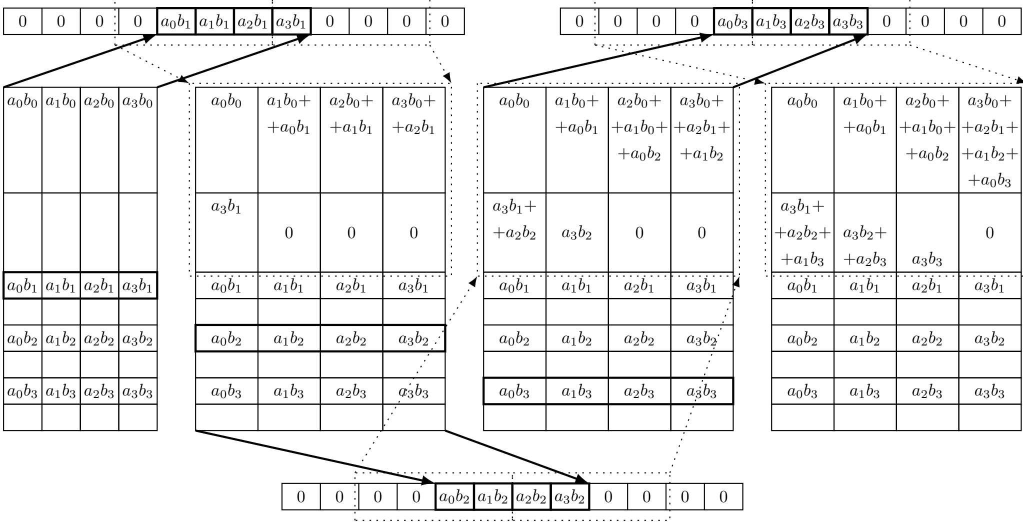

Figure 4: Full multiplication process of 4-coefficient polynomial slices on a hypothetical 1/8-size AMX unit. The sequence of operations starts from the leftmost state of the Z registers (result of the outer product operation) and follows the arrows, representing different states of Z and the relevant part of X. Thick arrows indicate an extrh operation of the thick-bordered row of Z to X, while dotted arrows indicate an accumulation of each dotted slice of X with a corresponding row (0 or 1) of the target Z registers.

Performance is further improved by working on pairs of even rows of Z, extracting one to X and the other to Y. As discussed in Section 3.3, a single vecint can accumulate slices of both X and Y registers to the desired Z row. Microbenchmarks indicate that vecint, on its own, achieves twice the arithmetic throughput of addition with X or Y alone; however, this improvement is not fully realized as there appears to be contention with resources used to execute extrh, and moreover, microbenchmarks suggest there is a penalty for unaligned accesses to the X and Y registers, as required to shift the input operands.

Algorithm 4.1 formalizes this procedure for polynomial multiplication of 32-coefficient slices. The reader may wish to review notations in Sections 2 and 3, in particular Figure 1.

Algorithm 4.1 PolyMul32×32($\mathbf{c_{k,l}}, \mathbf{a_k}, \mathbf{b_j}$): modulo-$2^{16}$ multiplication of two 32-coefficient polynomials using AMX.

Input: $\mathbf{a_k}, \mathbf{b_l}$ (arrays of 32-coefficient slices of $\mathbf{a}$ and $\mathbf{b}$)
Output: $\mathbf{c_{k,l}}$ (array of 63 coefficients of $c_{\mathbf{k},\mathbf{l}}(x) = a_{\mathbf{k}}(x) \times b_{\mathbf{l}}(x)$)
Notes: since AMX stores 64 bytes (in this case, 32 coefficients of 16 bits each) at a time, output array must be allocated with enough space for 64 coefficients

1: $\mathtt{X}_0, \mathtt{X}_2, \mathtt{Y}_0, \mathtt{Y}_2 \leftarrow \mathtt{ldx}([0\ 0\ \dots\ 0])$ ▷ load array of 32 zeros to each indicated register
2: $\mathtt{X}_1,\ \mathtt{Y}_1 \leftarrow \mathtt{ldx}(\mathbf{a_k}),\ \mathtt{ldy}(\mathbf{b_l})$
3: $\mathtt{Z}[0:2:62] \leftarrow \mathtt{mac16}(\mathtt{Y}_1^\mathsf{T}\mathtt{X}_1)$
4: $\mathtt{X}_1 \leftarrow \mathtt{extrh}(\mathtt{Z}[2])$
5: $\mathtt{Z}[0] \leftarrow \mathtt{vecint}(\mathtt{Z}[0] + \mathtt{X}[31:62])$
6: $\mathtt{Z}[1] \leftarrow \mathtt{X}[63:94]$ ▷ implement using e.g. `mac16` with `Y` and `Z` skipped
7: **for** $i \in \{2, 4, 6, \dots, 30\}$ **do**
8: $\quad\mathtt{X}_1,\ \mathtt{Y}_1 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i]),\ \mathtt{extrh}(\mathtt{Z}[2(i+1)])$
9: $\quad\mathtt{Z}[0] \leftarrow \mathtt{vecint}(\mathtt{Z}[0] + \mathtt{X}[32-i:63-i] + \mathtt{Y}[31-i:62-i])$
10: $\quad\mathtt{Z}[1] \leftarrow \mathtt{vecint}(\mathtt{Z}[1] + \mathtt{X}[64-i:95-i] + \mathtt{Y}[63-i:94-i])$
11: $\mathbf{c_{k,l}}[0:31],\ \mathbf{c_{k,l}}[32:63] \leftarrow \mathtt{stz}(\mathtt{Z}[0]),\ \mathtt{stz}(\mathtt{Z}[1])$

4.2 Polynomial multiplication by product-scanning of basic blocks

We turn to the issue of realizing a full multiplication of n-coefficient inputs. Since NTRU requires n to be prime, and the basic block of Section 4.1 works on 32-coefficient slices, we work around this issue by zero-padding input polynomials to $n' = 32\lceil n/32 \rceil$ coefficients.

In principle, we could apply Algorithm 4.1 to each slice and accumulate results using vector instructions to realize a full-size multiplication. However, we propose a different architecture, based on blockwise product scanning with lazy sum-reduction, to increase the balance of faster matrix (outer product) to slower vector (extraction/addition) operations. We again illustrate with a small example for a hypothetical 1/8-size AMX unit, multiplying 8-coefficient input polynomials split into 4-coefficient slices as in Section 4.1.

The parallelogram of Figure 5 consists of 4 outer products: $\mathbf{b_0^\mathsf{T} a_0}$, $\mathbf{b_1^\mathsf{T} a_0}$, $\mathbf{b_1^\mathsf{T} a_0}$ and $\mathbf{b_1^\mathsf{T} a_1}$. It is seen in Figure 5(b) that $\mathbf{b_1^\mathsf{T} a_0}$ and $\mathbf{b_0^\mathsf{T} a_1}$ are aligned columnwise and can be accumulated (by matrix addition) prior to shifting and flattening; that is, shifting and flattening are performed lazily. In general, outer products of the form $\mathbf{b_j^\mathsf{T} a_i}$ and $\mathbf{b_l^\mathsf{T} a_k}$ are aligned, and can be accumulated, if $i + j = k + l$. Note that matrix addition does not need to be performed explicitly in AMX, as the outer product instructions accumulate with the existing contents of the `Z` registers at no extra cost. The accumulation procedure is shown as Algorithm 4.2, a subroutine of our polynomial multiplication algorithm given later.

Algorithm 4.2 AccumulateOuterProducts($\mathbf{a}, \mathbf{b}, j, r$): accumulate outer products $\mathbf{b_l^\mathsf{T} a_k}$ such that $k + l = j$ to $\mathtt{Z}[r:2:62+r]$, where $r = 0$ or 1.

1: $\mathtt{X}_1,\ \mathtt{Y}_1 \leftarrow \mathtt{ldx}(\mathbf{a}[32j:32j+31]),\ \mathtt{ldy}(\mathbf{b}[0:31])$
2: $\mathtt{Z}[r:2:62+r] \leftarrow \mathtt{mac16}(\mathtt{Y}_1^\mathsf{T}\mathtt{X}_1)$
3: **for** $1 \le l \le j$ **do**
4: $\quad k \leftarrow j - l$ ▷ Thus: $k + l = j$
5: $\quad\mathtt{X}_1,\ \mathtt{Y}_1 \leftarrow \mathtt{ldx}(\mathbf{a}[32k:32k+31]),\ \mathtt{ldy}(\mathbf{b}[32l:32l+31])$
6: $\quad\mathtt{Z}[r:2:62+r] \leftarrow \mathtt{mac16}(\mathtt{Z}[r:2:62+r] + \mathtt{Y}_1^\mathsf{T}\mathtt{X}_1)$

We now consider a full-size AMX unit and realistic polynomial sizes, say n' coefficients. The naïve approach suggested at the beginning of this section would call for $(n'/32)^2 = O(n^2)$ applications of Algorithm 4.1, that is $O(n^2)$ outer products and shifts/flattenings, plus extra operations to combine the results to obtain the desired polynomial multiplication

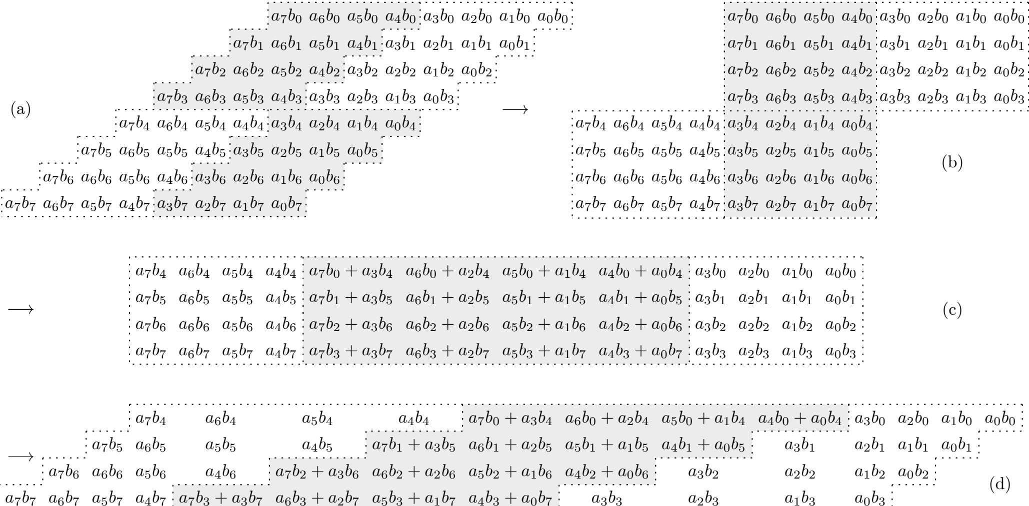

Figure 5: Transformations of the multiplication parallelogram for 8-coefficient polynomials for efficient implementation on a hypothetical 1/8-size AMX unit. Dotted lines enclose each outer product of 4-coefficient slices. Starting from the regular parallelogram (a), the i-th row is shifted right by i mod 4 positions (i.e. the reverse of the shifting step of Figure 4) to obtain (b), revealing how two outer products are aligned. In (c), these two outer products are summed prior to left-shifting the i-th row by i positions in the last state shown, (d). The final result is obtained by flattening each column (not shown).

result. The advantage of lazy shifting/flattening is clear from extrapolation of Figure 5: we see that it results in $2(n'/32) - 1 = O(n)$ matrices of dimension 32×32 in the third step of the process (Figure 5(c)), which are shifted (Figure 5(d)) and flattened (not shown) to obtain the final polynomial multiplication result. While we still compute the same amount of outer products, $(n'/32)^2 = O(n^2)$, they are considerably cheaper than shifts and flattenings realized by AMX vector instructions, of which only $O(n)$ are required.

Moving on to practical implementation issues, the `Z` register, where outer products accumulate, can only store two 32×32 matrices, one in the even rows and another in the odd rows. Careful sequencing of shift and flattening operations is needed to avoid spills and reloads of `Z`'s contents. Consider the flattened parallelogram in Figure 5(d), which is split into three sub-parallelograms, each corresponding to one of three matrices of Figure 5(c), formed from accumulating outer products; we denote them, from right to left, as M_0, M_1 and M_2, respectively. Working through the final flattening step (i.e. sum-reduction of columns, not shown in the figure), from the rightmost column towards the leftmost one, we see that the first columns (0 to 3) contain only elements from M_0. As we move left, columns 4 to 6 contain elements from M_0 and M_1; column 7 from M_1 only; columns 8 to 10 from M_1 and M_2; and columns 11 to 14 from M_2 only. A general pattern emerges: each column contains either elements from M_j only, or from M_j and M_{j+1}.

This suggests a structure for our polynomial multiplication algorithm, described in terms of full-size (n' coefficients) input polynomials. Formalizing the concept from the previous paragraph, we define $M_j = \sum_{k+l=j} \mathbf{b_l^\intercal a_k}$; the computation of each M_j is performed by Algorithm 4.2. We first compute M_0 and M_1, storing them in the even and odd rows of `Z`, respectively, as needed to perform shifts and flattenings to obtain the first 64 coefficients $c_0, \ldots, c_{63}$ of $c(x) = a(x)b(x)$, a procedure we formalize as Algorithm 4.3. We now compute M_2 overwriting the even rows of `Z`, and use both M_1 (still available in the odd rows of `Z`) and M_2 to obtain the next 32 coefficients of $c(x)$, i.e. $c_{64}, \ldots, c_{95}$, by the procedure of

Algorithm 4.4. The pattern continues for $j = 3, \ldots, 2(n/32) - 3$ with M_j overwriting even or odd rows of Z, according to whether j is even or odd, respectively, and using M_{j-1} and M_j to compute $c_{32j}, \ldots, c_{32j+31}$ using Algorithm 4.4. Finally, the case $j = 2(n'/32) - 2$ is handled by a third procedure, Algorithm 4.5, which computes the final batch of coefficients of $c(x)$ from M_{j-1} and M_j. We formalize the complete procedure as Algorithm 4.6.

Algorithm 4.3 FlattenFirstTwoBlocks(**c**): sum-reduction of columns from the two least significant multiplication sub-parallelograms.

Input: $\mathtt{Z}[0:2:62] = \mathbf{b_0^\intercal a_0}$ and $\mathtt{Z}[1:2:63] = \mathbf{b_0^\intercal a_1} + \mathbf{b_1^\intercal a_0}$ (implicitly)
Output: $\mathbf{c}[0:63]$ (coefficients of x^0 through x^{63} of the polynomial multiplication result)
Notes: assumes $\mathtt{X}_0 = \mathtt{Y}_0 = [0, 0, \ldots, 0]$

1: $\mathtt{X}_1,\ \mathtt{X}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[2]),\ \mathtt{extrh}(\mathtt{Z}[3])$
2: $\mathtt{Z}[0],\ \mathtt{Z}[1] \leftarrow \mathtt{mac16}(\mathtt{Z}[0] + \mathtt{X}[31:62]),\ \mathtt{mac16}(\mathtt{Z}[1] + \mathtt{X}[63:94])$
3: **for** $i \in \{2, 4, 6, \ldots, 30\}$ **do**
4: $\quad \mathtt{X}_1,\ \mathtt{X}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i]),\ \mathtt{extrh}(\mathtt{Z}[2i+1])$
5: $\quad \mathtt{Y}_1,\ \mathtt{Y}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i+2]),\ \mathtt{extrh}(\mathtt{Z}[2i+3])$
6: $\quad \mathtt{Z}[0] \leftarrow \mathtt{vecint}(\mathtt{Z}[0] + \mathtt{X}[32-i:63-i] + \mathtt{Y}[31-i:62-i])$
7: $\quad \mathtt{Z}[1] \leftarrow \mathtt{vecint}(\mathtt{Z}[1] + \mathtt{X}[64-i:95-i] + \mathtt{Y}[63-i:94-i])$
8: $\mathbf{c}[0:31],\ \mathbf{c}[32:63] \leftarrow \mathtt{stz}(\mathtt{Z}[0]),\ \mathtt{stz}(\mathtt{Z}[1])$

4.3 Integrated reduction modulo $x^n - 1$

The algorithms of Section 4.2, culminating in Algorithm 4.6, multiply polynomials of $n \times n$ coefficients and return the full result with $2n - 1$ coefficients. In NTRU, the result must be reduced modulo $x^n - 1$, a special form which allows efficient implementation as a post-processing step using e.g. NEON instructions, but also allowing reduction to be merged with multiplication in an integrated procedure, reducing the number of shifting/flattening operations by $\approx 50\%$ compared to Algorithm 4.6, improving performance considerably.

To exemplify the integrated procedure, we again resort to a hypothetical 1/8-size AMX unit, now performing polynomial multiplication modulo $x^{10} - 1$, which illustrates the main issues faced with larger polynomials on a full-size AMX unit. Although 10 coefficients are enough to represent polynomial inputs and outputs for this case, note that each block in this hypothetical 1/8-size AMX unit perfoms to a 4×4 outer-product operation. Thus, we work with the next multiple of 4 coefficients, which is 12. We refer to Figure 6 as we work through the procedure, and assume input polynomials have $a_i = b_j = 0$ for $i, j \geq 10$.

The results of all required outer product calculations are shown explicitly in Figure 6(a). The actual procedure accumulates 4×4 outer product results of a given color (white, light gray or gray) into a single per-color 4×4 matrix, as in Figure 5(c); for didactic reasons, we omit this step in the course of the explanation. Zero partial products are struck out in the figure; for some of these, this is a natural result of having a_i or b_j as input for $i, j \geq 10$, and for the others, we explicitly disable some columns during outer product computations.

Viewing Figure 6(a) as a block matrix of 4×4 submatrices, note that submatrices in and above the secondary (block) diagonal are constructed identically to the example of Figure 5, as formalized in Algorithm 4.2. Notice the pattern followed by partial products $a_i b_j$ in each row: those to its left and right are of the form $a_{i+1} b_j$ and $a_{i-1} b_j$, respectively. The rightmost columns of each submatrix in the secondary (block) diagonal have $i = 0$, and the pattern continues to their right, with $i - 1$ taken modulo n (10 in the example). For each partial product $a_i b_j$, with k its row index modulo 4 and l its column index, as indicated to the left and below the matrix in Figure 6(a), we have $k + l \equiv i + j \pmod{n}$.

Submatrices in and above the secondary (block) diagonal are outer products of the form $\mathbf{b}^\intercal_{4l:4l+3} \mathbf{a}_{4k:4k+3}$ (no relation to k and l in the previous paragraph), while submatrices

Algorithm 4.4 FlattenMiddleBlock($\mathbf{c}, j, r$): sum-reduction of columns from multiplication sub-parallelograms.

Input: j (index of current sub-parallelogram)
Input: $r \in \{0, 1\}$ (indicates relative order of even and odd rows of $\mathtt{Z}$)
Output: $\mathbf{c}[32j : 32j + 31]$ (coefficients of x^{32j} through x^{32j+31} of the polynomial multiplication result)
Notes: assumes that $\mathtt{Z}[1 - r : 2 : 63 - r]$ and $\mathtt{Z}[r : 2 : 62 + r]$ contain the output of Algorithm 4.2 for sub-parallelogram indices $j - 1$ and j, respectively

1: $\mathtt{X}_1,\ \mathtt{X}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[3 - r]),\ \mathtt{extrh}(\mathtt{Z}[2 + r])$
2: $\mathtt{Z}[r] \leftarrow \mathtt{mac16}(\mathtt{Z}[r] + \mathtt{X}[63 : 94])$
3: **for** $i \in \{2, 4, 6, \ldots, 30\}$ **do**
4: $\quad\mathtt{X}_1,\ \mathtt{X}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i + 1 - r]),\ \mathtt{extrh}(\mathtt{Z}[2i + r])$
5: $\quad\mathtt{Y}_1,\ \mathtt{Y}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i + 3 - r]),\ \mathtt{extrh}(\mathtt{Z}[2i + 2 + r])$
6: $\quad\mathtt{Z}[r] \leftarrow \mathtt{vecint}(\mathtt{Z}[r] + \mathtt{X}[64 - i : 95 - i] + \mathtt{Y}[63 - i : 94 - i])$
7: $\mathbf{c}[32j : 32j + 31] \leftarrow \mathtt{stz}(\mathtt{Z}[r])$

Algorithm 4.5 FlattenLastTwoBlocks($\mathbf{c}, j$): sum-reduction of columns from the two most significant multiplication sub-parallelograms.

Input: j (index of current block)
Output: $\mathbf{c}[32j : 32j + 62]$ (coefficients of x^{32j} through x^{32j+62} of $c(x) = a(x) \times b(x)$)
Notes: assumes $\mathtt{X}_3 = \mathtt{Y}_3 = [0, 0, \ldots, 0]$, and that $\mathtt{Z}[1 : 2 : 63]$ and $\mathtt{Z}[0 : 2 : 62]$ contain the output of Algorithm 4.2 for sub-parallelogram indices $j - 1$ and j, respectively; one extra coefficient written at $\mathbf{c}[32j + 63]$ since AMX writes 64 bytes at a time

1: $\mathtt{X}_1,\ \mathtt{X}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[3]),\ \mathtt{extrh}(\mathtt{Z}[2])$
2: $\mathtt{Z}[0],\ \mathtt{Z}[1] \leftarrow \mathtt{mac16}(\mathtt{Z}[0] + \mathtt{X}[63 : 94]),\ \mathtt{mac16}(\mathtt{Z}[1] + \mathtt{X}[95 : 126])$
3: **for** $i \in \{2, 4, 6, \ldots, 30\}$ **do**
4: $\quad\mathtt{X}_2,\ \mathtt{X}_1 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i]),\ \mathtt{extrh}(\mathtt{Z}[2i + 1])$
5: $\quad\mathtt{Y}_2,\ \mathtt{Y}_1 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i + 2]),\ \mathtt{extrh}(\mathtt{Z}[2i + 3])$
6: $\quad\mathtt{Z}[0] \leftarrow \mathtt{vecint}(\mathtt{Z}[0] + \mathtt{X}[32 - i : 63 - i] + \mathtt{Y}[31 - i : 62 - i])$
7: $\quad\mathtt{Z}[1] \leftarrow \mathtt{vecint}(\mathtt{Z}[1] + \mathtt{X}[64 - i : 95 - i] + \mathtt{Y}[63 - i : 94 - i])$
8: $\mathbf{c}[32j : 32j + 31],\ \mathbf{c}[32j + 32 : 32j + 63] \leftarrow \mathtt{stz}(\mathtt{Z}[0]),\ \mathtt{stz}(\mathtt{Z}[1])$

Algorithm 4.6 PolyMul($\mathbf{c}, \mathbf{a}, \mathbf{b}$): modulo-$2^{16}$ multiplication of two n-coefficient polynomials using AMX. Assumes n a multiple of 32; input must be zero-padded otherwise.

Input: $\mathbf{a}, \mathbf{b}$ (arrays of n coefficients each of $a(x)$ and $b(x)$)
Output: $\mathbf{c}$ (array of $2n - 1$ coefficients of $c(x) = a(x)b(x)$)
Notes: since AMX stores 64 bytes (in this case, 32 coefficients of 16 bits each) at a time, output array must be allocated with $2n$ coefficients

1: $\mathtt{X}_0, \mathtt{X}_3, \mathtt{Y}_0, \mathtt{Y}_3 \leftarrow \mathtt{ldx}([0, 0, \ldots, 0])$ ▷ load array of 32 zeros to each indicated register
2: AccumulateOuterProducts($\mathbf{a}, \mathbf{b}, 0, 0$)
3: AccumulateOuterProducts($\mathbf{a}, \mathbf{b}, 1, 1$)
4: FlattenFirstTwoBlocks($\mathbf{c}$)
5: **for** $i = 2$ **to** $2(n/32) - 3$ **do**
6: $\quad$AccumulateOuterProducts($\mathbf{a}, \mathbf{b}, i, i \bmod 2$)
7: $\quad$FlattenMiddleBlock($\mathbf{c}, i, i \bmod 2$)
8: AccumulateOuterProducts($\mathbf{a}, \mathbf{b}, 2(n/32) - 2, 0$)
9: FlattenLastTwoBlocks($\mathbf{c}, 2(n/32) - 2$)

(a)

	11	10	9	8	7	6	5	4	3	2	1	0
0	~~$a_{11}b_0$~~	~~$a_{10}b_0$~~	a_9b_0	a_8b_0	a_7b_0	a_6b_0	a_5b_0	a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
1	~~$a_{11}b_1$~~	~~$a_{10}b_1$~~	a_9b_1	a_8b_1	a_7b_1	a_6b_1	a_5b_1	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1
2	~~$a_{11}b_2$~~	~~$a_{10}b_2$~~	a_9b_2	a_8b_2	a_7b_2	a_6b_2	a_5b_2	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2
3	~~$a_{11}b_3$~~	~~$a_{10}b_3$~~	a_9b_3	a_8b_3	a_7b_3	a_6b_3	a_5b_3	a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3
0	~~a_7b_4~~	~~a_6b_4~~	a_5b_4	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4	a_9b_4	a_8b_4	a_7b_4	a_6b_4
1	~~a_7b_5~~	~~a_6b_5~~	a_5b_5	a_4b_5	a_3b_5	a_2b_5	a_1b_5	a_0b_5	a_9b_5	a_8b_5	a_7b_5	a_6b_5
2	~~a_7b_6~~	~~a_6b_6~~	a_5b_6	a_4b_6	a_3b_6	a_2b_6	a_1b_6	a_0b_6	a_9b_6	a_8b_6	a_7b_6	a_6b_6
3	~~a_7b_7~~	~~a_6b_7~~	a_5b_7	a_4b_7	a_3b_7	a_2b_7	a_1b_7	a_0b_7	a_9b_7	a_8b_7	a_7b_7	a_6b_7
0	~~a_3b_8~~	~~a_2b_8~~	a_1b_8	a_0b_8	a_9b_8	a_8b_8	a_7b_8	a_6b_8	a_5b_8	a_4b_8	a_3b_8	a_2b_8
1	~~a_3b_9~~	~~a_2b_9~~	a_1b_9	a_0b_9	a_9b_9	a_8b_9	a_7b_9	a_6b_9	a_5b_9	a_4b_9	a_3b_9	a_2b_9
2	~~a_3b_{10}~~	~~a_2b_{10}~~	~~a_1b_{10}~~	~~a_0b_{10}~~	~~a_9b_{10}~~	~~a_8b_{10}~~	~~a_7b_{10}~~	~~a_6b_{10}~~	~~a_5b_{10}~~	~~a_4b_{10}~~	~~a_3b_{10}~~	~~a_2b_{10}~~
3	~~a_3b_{11}~~	~~a_2b_{11}~~	~~a_1b_{11}~~	~~a_0b_{11}~~	~~a_9b_{11}~~	~~a_8b_{11}~~	~~a_7b_{11}~~	~~a_6b_{11}~~	~~a_5b_{11}~~	~~a_4b_{11}~~	~~a_3b_{11}~~	~~a_2b_{11}~~

(b)

12	11	10	9	8	7	6	5	4	3	2	1	0
			a_9b_0	a_8b_0	a_7b_0	a_6b_0	a_5b_0	a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
		a_9b_1	a_8b_1	a_7b_1	a_6b_1	a_5b_1	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
	a_9b_2	a_8b_2	a_7b_2	a_6b_2	a_5b_2	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
a_9b_3	a_8b_3	a_7b_3	a_6b_3	a_5b_3	a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
			a_5b_4	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4	a_9b_4	a_8b_4	a_7b_4	a_6b_4
		a_5b_5	a_4b_5	a_3b_5	a_2b_5	a_1b_5	a_0b_5	a_9b_5	a_8b_5	a_7b_5	a_6b_5	
	a_5b_6	a_4b_6	a_3b_6	a_2b_6	a_1b_6	a_0b_6	a_9b_6	a_8b_6	a_7b_6	a_6b_6		
a_5b_7	a_4b_7	a_3b_7	a_2b_7	a_1b_7	a_0b_7	a_9b_7	a_8b_7	a_7b_7	a_6b_7			
			a_1b_8	a_0b_8	a_9b_8	a_8b_8	a_7b_8	a_6b_8	a_5b_8	a_4b_8	a_3b_8	a_2b_8
		a_1b_9	a_0b_9	a_9b_9	a_8b_9	a_7b_9	a_6b_9	a_5b_9	a_4b_9	a_3b_9	a_2b_9	

(c)

9	8	7	6	5	4	3	2	1	0
a_9b_0	a_8b_0	a_7b_0	a_6b_0	a_5b_0	a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
a_8b_1	a_7b_1	a_6b_1	a_5b_1	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	a_9b_1
a_7b_2	a_6b_2	a_5b_2	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2	a_9b_2	a_8b_2
a_6b_3	a_5b_3	a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3	a_9b_3	a_8b_3	a_7b_3
a_5b_4	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4	a_9b_4	a_8b_4	a_7b_4	a_6b_4
a_4b_5	a_3b_5	a_2b_5	a_1b_5	a_0b_5	a_9b_5	a_8b_5	a_7b_5	a_6b_5	a_5b_5
a_3b_6	a_2b_6	a_1b_6	a_0b_6	a_9b_6	a_8b_6	a_7b_6	a_6b_6	a_5b_6	a_4b_6
a_2b_7	a_1b_7	a_0b_7	a_9b_7	a_8b_7	a_7b_7	a_6b_7	a_5b_7	a_4b_7	a_3b_7
a_1b_8	a_0b_8	a_9b_8	a_8b_8	a_7b_8	a_6b_8	a_5b_8	a_4b_8	a_3b_8	a_2b_8
a_0b_9	a_9b_9	a_8b_9	a_7b_9	a_6b_9	a_5b_9	a_4b_9	a_3b_9	a_2b_9	a_1b_9

Figure 6: Sequence of operations to implement integrated polynomial multiplication and reduction of 12-coefficient polynomials modulo $x^{10} - 1$ on a hypothetical 1/8-size AMX unit. See text for a description of transformations from each subfigure to the next.

below this diagonal are obtained from outer products $\mathbf{b}^{\mathsf{T}}_{4l:4l+3}\mathbf{a}_{4k+2:4k+5}$; note how the coefficients of $\mathbf{a}$ are "unaligned", i.e. the starting index is not a multiple of 4 (32 for a full-size AMX unit). The offset 2 in the example corresponds in general to n mod 4, due to the modulo $x^n - 1$ reduction when n is not a multiple of 4 (32 for a full-size AMX unit).

We now present Algorithm 4.7, a counterpart of Algorithm 4.2 for the integrated polynomial multiplication and reduction procedure. It follows Algorithm 4.2 in accumulating outer products with matching columns (i.e. the submatrices of the same color in Figure 6(a)), a step which we recall was omitted from the example for ease of exposition.

We transform Figure 6(a) into Figure 6(b) by removing struck-out partial products and shifting rows of Figure 6(a) by the row index modulo 4; indices i, j of partial products a_ib_j in each column add exactly to the column index modulo 10. As we are working modulo $x^{10} - 1$, the final result must be restricted to indices from 0 to 9; we mark this boundary with a thick vertical line in the figure. Some partial products remain in columns with index $k \geq 10$, i.e. to the left of this line. We move them to the column with index $k' = k \bmod 10$ in the same row to arrive at Figure 6(c); note that $i + j \equiv k \pmod{10}$ always holds.

Note parallelogram widths in Figure 5(c)/(d) and Figure 6(c): $2n-1$ and n coefficients, respectively, validating the earlier claim that the number of sum-reductions (shifts and flattenings) is cut approximately in half by integrating multiplication and reduction.

Algorithms 4.3 and 4.4 perform shifts and flattenings for columns 0 to 9 of the

Algorithm 4.7 AccumulateOuterProductsReduction($\mathbf{a}, \mathbf{b}, j, r, n, m$): accumulate outer products for polynomial multiplication with reduction modulo $x^n - 1$ to $\mathtt{Z}[r:2:62+r]$, where $r = 0$ or 1; only the first $m \leq 32$ columns are computed.

1: $\mathtt{X}_1,\ \mathtt{Y}_1 \leftarrow \mathtt{ldx}(\mathbf{a}[32j : 32j+31]),\ \mathtt{ldy}(\mathbf{b}[0:31])$
2: $\mathtt{Z}[r:2:62+r] \leftarrow \mathtt{mac16}(\mathtt{Y}_1^\intercal \mathtt{X}_1, \text{columns} = 0 : m-1)$
3: **for** $1 \leq l \leq j$ **do**
4: $\quad k \leftarrow j - l$ ▷ Thus: $k + l = j$
5: $\quad \mathtt{X}_1,\ \mathtt{Y}_1 \leftarrow \mathtt{ldx}(\mathbf{a}[32k : 32k+31]),\ \mathtt{ldy}(\mathbf{b}[32l : 32l+31])$
6: $\quad \mathtt{Z}[r:2:62+r] \leftarrow \mathtt{mac16}(\mathtt{Z}[r:2:62+r] + \mathtt{Y}_1^\intercal \mathtt{X}_1, \text{columns} = 0 : m-1)$
7: **for** $j < l \leq \lfloor (n-1)/32 \rfloor$ **do**
8: $\quad \mathtt{X}_1,\ \mathtt{Y}_1 \leftarrow \mathtt{ldx}(\mathbf{a}[n - 32(l-1) : n - 32(l-1) + 31]),\ \mathtt{ldy}(\mathbf{b}[32l : 32l+31])$
9: $\quad \mathtt{Z}[r:2:62+r] \leftarrow \mathtt{mac16}(\mathtt{Z}[r:2:62+r] + \mathtt{Y}_1^\intercal \mathtt{X}_1, \text{columns} = 0 : m-1)$

parallelogram; in this example, Algorithm 4.3 is applied to the first two blocks (columns 0 to 3 and 4 to 7 of Figure 6(b)), and Algorithm 4.4 to columns 8 and 9 (which also includes columns 10 and 11 in the result, an issue we must work around). Handling columns 10, 11 and 12, i.e. merging them with the main matrix as in Figure 6(c) (in this case, with columns 0, 1 and 2, respectively) requires a new procedure, formalized as Algorithm 4.8.

We leverage the algorithms of this section and of Section 4.2 in a procedure for integrated polynomial multiplication and reduction, formalized as Algorithm 4.9, which we use as the main polynomial multiplication routine in NTRU, i.e. the `poly_Rq_mul()` function.

Algorithm 4.8 MergeFirstAndLastBlocks($\mathbf{c}, j, r, n$): sum-reduction of columns from the two most significant multiplication sub-parallelograms.

Input: j (index of current sub-parallelogram)
Input: $r \in \{0, 1\}$ (indicates relative order of even and odd rows of $\mathtt{Z}$)
Input: n (reduction polynomial is $x^n - 1$)
Output: $\mathbf{c}[0:31]$ (coefficients of x^0 through x^{31} of the polynomial multiplication result)
Notes: assumes that $\mathtt{Z}[1-r:2:63-r]$ and $\mathtt{Z}[r:2:62+r]$ contain the output of Algorithm 4.7 for sub-parallelogram indices $j-1$ and j, respectively

1: $\delta \leftarrow (-n) \bmod 32$
2: $\mathtt{Z}[r] \leftarrow \mathtt{ldz}(c[0:31])$
3: $\mathtt{X}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[3-r])$
4: $\mathtt{Z}[r] \leftarrow \mathtt{mac16}(\mathtt{Z}[r] + \mathtt{X}[95-\delta : 126-\delta])$
5: $i \leftarrow 1$
6: **while** $i < 31 - \delta$ **do**
7: $\quad \mathtt{X}_2,\ \mathtt{Y}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i+3-r]),\ \mathtt{extrh}(\mathtt{Z}[2i+5-r])$
8: $\quad \mathtt{Z}[r] \leftarrow \mathtt{vecint}(\mathtt{Z}[r] + \mathtt{X}[95-\delta-i : 126-\delta-i] + \mathtt{Y}[94-\delta-i : 125-\delta-i])$
9: $\quad i \leftarrow i + 2$
10: **while** $i < 31$ **do**
11: $\quad \mathtt{X}_1,\ \mathtt{X}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i+2+r]),\ \mathtt{extrh}(\mathtt{Z}[2i+3-r])$
12: $\quad \mathtt{Y}_1,\ \mathtt{Y}_2 \leftarrow \mathtt{extrh}(\mathtt{Z}[2i+4+r]),\ \mathtt{extrh}(\mathtt{Z}[2i+5-r])$
13: $\quad \mathtt{Z}[r] \leftarrow \mathtt{vecint}(\mathtt{Z}[r] + \mathtt{X}[95-\delta-i : 126-\delta-i] + \mathtt{Y}[94-\delta-i : 125-\delta-i])$
14: $\quad i \leftarrow i + 2$
15: $\mathbf{c}[0:31] \leftarrow \mathtt{stz}(\mathtt{Z}[r])$

4.4 Working around memory access slowdowns

An initial implementation of polynomial multiplication modulo $x^n - 1$, applying all optimizations mentioned in Sections 4.2 and 4.3, outperforms the analogous routines from the

Algorithm 4.9 PolyModMul($\mathbf{c}, \mathbf{a}, \mathbf{b}, n$): modulo-$2^{16}$ multiplication of two n-coefficient polynomials using AMX, reduced modulo $x^n - 1$.

Input: $\mathbf{a}, \mathbf{b}$ (arrays of n coefficients each of $a(x)$ and $b(x)$)
Output: $\mathbf{c}$ (array of n coefficients of $c(x) = a(x)b(x)$)
Notes: since AMX stores 64 bytes (in this case, 32 coefficients of 16 bits each) at a time, output array must be allocated with $32\lceil n/32 \rceil$ coefficients

1: $n' \leftarrow \lfloor (n-1)/32 \rfloor$
2: $\mathtt{X}_0, \mathtt{X}_3, \mathtt{Y}_0, \mathtt{Y}_3 \leftarrow \mathtt{ldx}([0, 0, \dots, 0])$ ▷ load zeros to all indicated registers
3: AccumulateOuterProductsReduction($\mathbf{a}, \mathbf{b}, 0, 0, n, 32$)
4: AccumulateOuterProductsReduction($\mathbf{a}, \mathbf{b}, 1, 1, n, 32$)
5: FlattenFirstTwoBlocks($\mathbf{c}$)
6: **for** $i = 2$ **to** $n' - 2$ **do**
7: AccumulateOuterProductsReduction($\mathbf{a}, \mathbf{b}, i, i \bmod 2, n, 32$)
8: FlattenMiddleBlock($\mathbf{c}, i, i \bmod 2$)
9: AccumulateOuterProductsReduction($\mathbf{a}, \mathbf{b}, n' - 1, (n' - 1) \bmod 2, n, n \bmod 32$)
10: FlattenMiddleBlock($\mathbf{c}, n' - 1, (n' - 1) \bmod 2$)
11: MergeFirstAndLastBlocks($\mathbf{c}, n' - 1, (n' - 1) \bmod 2, n$)

implementations of [NG21] and [CCHY23] when benchmarked in isolation. However, improvements were modest for the operations of the full scheme (key generation, encapsulation and decapsulation), with [CCHY23] still outperforming our NTRU implementation.

Handley [Han23] claims that load/store queues track AMX memory accesses at a granularity of memory pages, marking them as “interesting” for AMX, and concurrent CPU and AMX accesses to the same page will block. Inspection of the `kem.c`, `owcpa.c` and `poly.c` source files shows that polynomial coefficient arrays are allocated on the stack, next to other variables used exclusively by CPU code, and thus stored in adjacent addresses in memory, very likely in the same memory page. We conjectured this was the root cause of our performance issues, we switched to allocating one polynomial per memory page using the `mmap()` function. Benchmarks using this new memory allocation scheme led to our implementation outperforming those of [NG21] and [CCHY23] in all tested scenarios.

Johnson [Joh22b] suggests using CPU prefetch instructions, which (unlike AMX instructions) are executed out of order by the CPU, triggering memory accesses in advance of AMX load/store instructions and reducing latency. However, we saw no performance improvements from this. It appears this suggestion is targeted towards typical AMX applications with working sets too large to fit CPU caches, which is not the case for NTRU.

As the CPU’s NEON units are independent from AMX, one might conjecture that splitting tasks between both can achieve even higher throughput. However, AMX/CPU memory conflicts incur performance penalties which may invalidate this strategy, unless NEON and AMX work on distinct data, stored on different memory pages.

5 Experimental results

We benchmark our implementation on a 2020 Apple MacBook Air laptop with the Apple M1 SoC with nominal P-core clock speed of 3200 MHz, and a 2023 Apple MacBook Pro laptop with the Apple M3 Max SoC with nominal P-core clock speed of 4064 MHz. All implementations are compiled with Apple clang 15 using the `-O3` optimization flag.

We compare our implementation to the state-of-the-art ones of [CCHY23], which implements only HPS2048677 and HRSS701 (and holds the current speed record for these sets), and [NG21], which implements all parameter sets. We backport optimized polynomial inversion and constant-time sorting routines from [CCHY23] to [NG21], seeking

to highlight differences in polynomial multiplication performance only; we include this modified implementation in our GitHub repository. We also optimize NIST's AES-256 CTR-DRBG `randombytes()` function with AES instructions from ARMv8-A's Cryptographic Extensions, using software tests to verify that outputs are bit-identical to the NIST version.

All of these optimizations (polynomial inversion, constant-time sorting and random number generation) are also applied to our AMX implementation. We ensure Known Answer Tests for all implementations match those provided in the NTRU reference code.

Recall from Section 4.4 that our implementation uses a specific memory allocation strategy for arrays of polynomial coefficients. To ensure a fair comparison, we create stack and `mmap()` allocation variants for every implementation and report performance figures for both, computing speedups by comparing the fastest strategy for each implementation.

Performance measurements use the same cycle-counting harness found in [NG21] and [CCHY23]. To reduce timing variabilities, each routine is run for 1,024 times in a loop, measuring the cycle counts of each run individually, and the median value is reported.

5.1 Performance measurements

We present NTRU performance results in Table 4. We omit the `tc` variant of HPS2048677 included in [CCHY23], as the `tmvp` variant always outperforms it.

Table 4: Cycle counts for NTRU KEM operations and polynomial multiplication. Speedups computed as a ratio of the best cycle counts for previous implementations and our implementation (in both cases, across different memory allocations types).

Par. set	**Mem. alloc.**	**Work**	**Operation**							
			Key gen.		**Encaps.**		**Decaps.**		**Poly. mult.**	
			M1	**M3**	**M1**	**M3**	**M1**	**M3**	**M1**	**M3**
509	`mmap`	[NG21]	218938	215086	16314	15695	29876	28667	6352	6351
		Ours	170334	163936	12620	11885	18976	17300	2281	2009
	stack	[NG21]	218436	214434	16287	15805	29870	28780	6373	6349
		Ours	181787	170648	19628	14327	26609	23810	3675	3229
	Speedup (×)		**1.28**	**1.31**	**1.29**	**1.32**	**1.57**	**1.66**	**2.78**	**3.16**
677	`mmap`	[NG21]	364048	348559	23738	22591	44977	42912	10028	10347
		[CCHY23]	307460	296025	19272	18586	31718	30620	5472	7080
		Ours	283023	266334	17324	16205	26443	24004	3781	3715
	stack	[NG21]	363999	348318	23806	22668	46566	44654	9998	10308
		[CCHY23]	307014	295602	20789	19596	35454	34061	5523	6056
		Ours	316845	291497	27777	16441	35953	32728	4299	4303
	Speedup (×)		**1.08**	**1.11**	**1.11**	**1.15**	**1.20**	**1.28**	**1.63**	**1.81**
821	`mmap`	[NG21]	497771	493264	28254	27728	56756	56294	13912	13911
		Ours	384569	371277	19520	18303	30981	28292	4789	4414
	stack	[NG21]	475626	492597	28261	27653	56874	56227	13859	13911
		Ours	610072	384194	27403	19204	40542	37827	5386	7100
	Speedup (×)		**1.29**	**1.33**	**1.45**	**1.51**	**1.83**	**1.99**	**2.89**	**3.15**
701	`mmap`	[NG21]	393734	384123	20804	20078	54265	53030	12870	12603
		[CCHY23]	323627	308902	14731	14044	37233	35423	6911	6860
		Ours	287657	269343	11659	10657	29038	26622	3708	3367
	stack	[NG21]	398678	386098	20779	20160	55835	53942	12529	12300
		[CCHY23]	323551	308247	14791	14128	37300	35544	6918	6858
		Ours	310970	273767	11729	10804	36039	31719	5431	5876
	Speedup (×)		**1.12**	**1.14**	**1.26**	**1.32**	**1.28**	**1.33**	**1.86**	**2.04**

Stack allocation of polynomials fluctuates between small slowdowns and speedups for NEON implementations (with a few outliers), but AMX consistently favors `mmap()`

allocation, more so in the M1; recall that speedup calculations pick the best performing memory allocation method for each implementation. The M3 AMX unit is faster than the M1's; in particular, microbenchmarks show that the M3 executes some vector AMX operations at a rate of 2 instructions/cycle, a phenomenon we didn't observe in the M1.

The most representative results are for HPS2048677 and HRSS701 using the TMVP algorithm [CCHY23]. Our polynomial multiplication routine achieves significant speedups of 1.63× (M1)/1.81× (M3) for the former and 1.86× (M1)/2.04× (M3) for the latter. These translate into smaller speedups for KEM operations (1.08× to 1.28× for the M1 and 1.11× to 1.33× for the M3), since the time spent in other routines (such as polynomial inversion for key generation, and constant-time sorting for key generation and encapsulation) remain identical between our implementation and previous ones. Nevertheless, our implementation consistently outperforms all others. For the remaining parameter sets, our gains are even more pronounced, but we expect the gap would shrink if TMVP were applied to them.

Scaling and subquadratic algorithms. In Figure 7(a), we provide polynomial multiplication scaling results as a function of polynomial degree, either with or without reduction modulo $x^n - 1$. We also perform a least-squares fit to an equation of the form ax^b, and note that in this range, the algorithm displays subquadratic scaling; this can be attributed to $O(n^2)$ scaling of outer product operations (cheaper) versus $O(n)$ scaling of shifts and flattenings (more expensive). We note that the version with reduction outperforms the one without, as it requires only about half as many slower shift/flattening operations.

We also implemented Karatsuba on AMX, for a single recursion level with our routine of Section 4.2 as a basecase, and compare it to the routine of Section 4.3 in Figure 7(b). Karatsuba is seen to outperfom schoolbook for $n \geq 4544$, far outside the range of cryptographic interest. We give numerical results for all NTRU parameter sets in Table 5, and also compute lower bounds (accounting for the cost of pointwise multiplications only) for single-level Karatsuba and subquadratic algorithms across all possible multi-level recursion strategies, using either schoolbook, Karatsuba, Toom-3 or Toom-4 at each level (the bounds are computed by a Jupyter notebook included in our GitHub repository). We conclude that schoolbook outperforms any subquadratic strategy for all NTRU parameter sets.

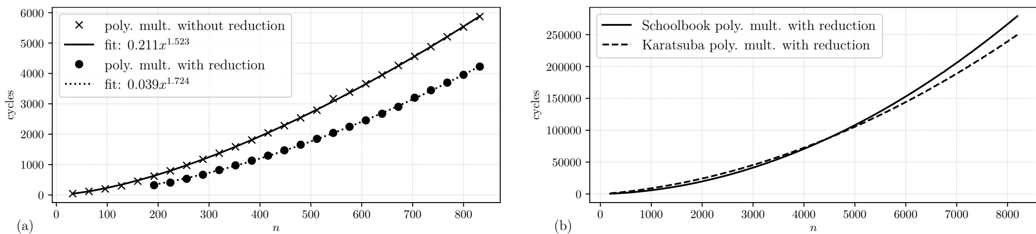

Figure 7: Cycle counts for AMX polynomial multiplication routines in the M3 as a function of n: (a) schoolbook algorithm with and without reduction modulo $x^n - 1$; (b) schoolbook and (single recursion level) Karatsuba algorithms with reduction.

Constant-time experiments. We empirically tested whether AMX instructions execute with data-independent timing, by benchmarking polynomial multiplication routines with either zero or random polynomials as inputs; the former were chosen as the most likely case for hypothetical data-dependent hardware optimizations. We also performed these experiments for the implementations of [NG21] and [CCHY23] for comparison purposes.

Our experiment consists in creating either zero polynomials or a single pair of random polynomials, and repeatedly computing polynomial multiplications with these fixed inputs. We benchmark each operation 64 times and return the median cycle count; we loop this

Table 5: Cycle counts for schoolbook and Karatsuba polynomial multiplication in the M3, and a lower bound computed across all possible subquadratic recursion strategies.

Parameter set	Schoolbook performance	Subquadratic lower bound	Single level Karatsuba	
			lower bound	performance
HPS2048509	1848	2100	2922	3506
HPS2048677/HRSS701	3202	3881	4758	5396
HPS4096821	4226	4813	6132	6793

for 1,000,000 times for zero inputs, and another 1,000,000 times for random inputs. We discard the 0.5% smallest and 0.5% highest readings and draw cycle count histograms.

As discussed in Section 4.4, allocating polynomials on the stack causes undesirable CPU-AMX memory subsystem interactions. In addition to worse performance (Table 4), our experiments exhibited performance instabilities (bimodal cycle count distributions for the same code across benchmark iterations); we conjecture they arise from CPU accesses of neighboring stack variables. Since our implementation does not index arrays based on secret data, any such instabilities cannot be exploited in timing side-channel attacks. Regardless, in choosing a representative example for Figure 8, we opted for a version using `mmap()` allocation, displaying some of the worst-case timing differences in the M3. The full dataset, including similarly-behaving M1 data, is available in our GitHub repository.

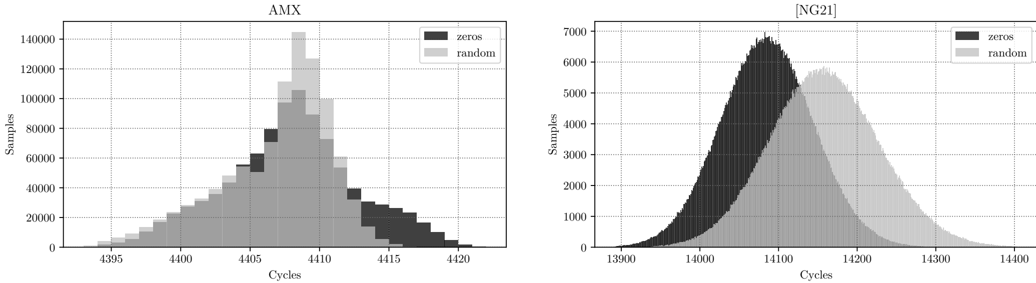

Figure 8: Histogram of cycle counts for 1,000,000 iterations of polynomial multiplication, for either zero or random inputs, for the AMX and [NG21] NEON implementation of NTRU HPS4096821 with `mmap()` memory allocation, running on the M3.

We note some timing differences between the zero and random inputs, for both AMX and NEON implementations. In the case of AMX, the outer product instruction is executed $\lceil 821/32 \rceil^2 = 676$ times, whereas the largest variations we see are in the order of 5 cycles. While not all outer product instructions in the code are latency-bound, deviations are on the order of < 0.01 cycles per outer product instruction, which seem unlikely to arise on purpose due to a variable-time ALU design. Note also that even higher deviations occur on the presumed constant-time NEON implementation. Thus, they seem more likely to be random deviations between benchmark runs; indeed, while zero inputs are marginally faster than random inputs in the NEON example, we also see occurrences of the reverse.

We also ran an experiment to measure the latency of matrix-mode `mac16`, for both zero and random inputs, as we believe this is the most likely target for data-dependent timing. We repeatedly run `mac16` and extract the results back from `Z` to `X` and `Y`, creating an inter-iteration data dependency, which serializes all operations so that we can measure their latency. Running this in the M3 for 100,000 iterations, we get 2,294,898 cycles for zero inputs and 2,294,876 cycles for random inputs, a difference of only 18 cycles or 0.0001%. A variable-time ALU would exhibit at least 1 cycle difference in latency between both cases, and thus one would expect a difference of at least 1 cycle/iteration, i.e., 100,000 cycles.

In summary, we see no evidence of data-dependent timing in AMX.

6 Conclusion

We report on an implementation of polynomial multiplication, applied to the NTRU lattice-based PQC scheme, leveraging a matrix-multiplication coprocessor which, as of 2023, is shipping on hundreds of millions of devices. Our implementation sets new speed records on two representative devices featuring this coprocessor, the Apple M1 and M3 SoCs, outperforming state-of-the-art conventional implementations for the same platforms.

CPU-coupled matrix multiplication accelerators will soon flood the market, bringing about huge leaps in processing power, but implementors must adapt to their new architectural paradigms to harness their added performance. Our most surprising conclusion is that multiplication (2D outer product) is much cheaper in AMX than vector (1D) operations, which must be applied row- or column-wise, increasing instruction count. As subquadratic algorithms trade off multiplication count for extra vector operations, AMX's architectural tradeoffs favor schoolbook for small polynomial degrees of cryptographic interest.

Future work. We suggest several avenues of future work from the results of our paper.

Other schemes based on NTT-unfriendly rings may see similar speedups from AMX, such as SABER [BMD+20] and FrodoKEM [BCD+16, ABD+21] (note the latter uses matrix-matrix multiplication). It is unclear whether NTT-based schemes such as Kyber [ABD+19] and Dilithium [BDK+21] are a good match for AMX, but it merits investigation.

While we have argued that schoolbook should outperform other subquadratic algorithms for NTRU, we assume that recursion is performed in a "black-box" fashion, calling self-contained functions to compute multiplications of smaller degree. We encourage investigating if an integrated approach, as in Sections 4.2 and 4.3, can improve performance.

Finally, we are witnessing the initial generations of matrix-multiplication accelerators. As they evolve, implementation techniques must equally adapt: faster vector operations could render subquadratic algorithms feasible, and the introduction of instructions to accelerate matrix-vector products could favor the use of TMVP. In this vein, other accelerators such as Intel AMX and ARMv9-A SME undoubtedly differ to some degree in available instructions and performance characteristics from Apple's AMX, and warrant a careful study aiming to exploit their performance potential for post-quantum cryptography.

References

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2019. https://pq-crystals.org/kyber/resources.shtml.

[ABD+21] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM learning with errors key encapsulation: Algorithm specifications and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2021. https://frodokem.org/files/FrodoKEM-specification-20210604.pdf.

[Ame17] American National Standards Institute. Lattice-based polynomial public key establishment algorithm for the financial services industry. ASC X9.98-2010 (R2017), 2017.

[App23] Apple Inc. Accelerate: Make large-scale mathematical computations and image calculations, optimized for high performance and low energy consumption, 2023. https://developer.apple.com/documentation/accelerate.

[ARM] ARM Limited. How is instruction timing affected by the FEAT_DIT architectural feature? URL: https://developer.arm.com/documentation/ddi0487/ja/.

[BCD+16] Joppe Bos, Craig Costello, Leo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2016. URL: https://doi.org/10.1145/2976749.2978425.

[BDK+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Dilithium: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2021. https://pq-crystals.org/dilithium/resources.shtml.

[BMD+20] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. Saber: Mod-LWR based KEM (round 3 submission). Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/index.html.

[Bun21] Bundesamt für Sicherheit in der Informationstechnik. Migration to post quantum cryptography: Recommendations for action by the BSI, 2021. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Migration_to_Post_Quantum_Cryptography.pdf?__blob=publicationFile&v=2.

[Caw23] Peter Cawley. Apple AMX instruction set, 2023. https://github.com/corsix/amx/.

[CCHY23] Han-Ting Chen, Yi-Hua Chung, Vincent Hwang, and Bo-Yin Yang. Algorithmic views of vectorized polynomial multipliers – NTRU. Cryptology ePrint Archive, Report 2023/1637, 2023. https://ia.cr/2023/1637. To appear at INDOCRYPT 2023.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M.Schanck, Tsunekazu Saito, Peter Schwabe, William Whyte, Keita Xagawa, Takashi Yamakawa, and Zhenfei Zhang. NTRU algorithm specifications and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020. https://ntru.org/resources.shtml.

[CHWZ17] Cong Chen, Jeffrey Hoffstein, William Whyte, and Zhenfei Zhang. NIST PQ submission: NTRUEncrypt - a lattice based encryption algorithm. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/NTRUEncrypt.zip.

[Han23] Maynard Handley. AArch64-Explore: Exploration of Apple CPUs – volume 3: SoC, 2023. https://github.com/name99-org/AArch64-Explore.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography*, pages 341–371. Springer International Publishing, 2017. doi:10.1007/978-3-319-70500-2_12.

[HP19] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, jan 2019. doi:10.1145/3282307.

[HPS96] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: a new high speed public key cryptosystem. CRYPTO '96 rump session, 1996. https://web.securityinnovation.com/hubfs/files/ntru-orig.pdf.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[HRSS17a] Andreas Hülsing, Joost Rijneveld, John Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 232–252, Cham, 2017. Springer-Verlag Berlin Heidelberg.

[HRSS17b] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. NTRU-HRSS-KEM algorithm specifications and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2017. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/NTRU_HRSS_KEM.zip.

[Ins09] Institute of Electrical and Electronics Engineers. IEEE standard specification for public key cryptographic techniques based on hard problems over lattices. IEEE Std 1363.1-2008, 2009. doi:10.1109/IEEESTD.2009.4800404.

[Int22] 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.

[Int23a] Intel Corporation. Data operand independent timing instruction set architecture (ISA) guidance, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html.

[Int23b] Intel Corporation. Data operand independent timing instructions, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[Int23c] International Organization for Standardization. FrodoKEM: Learning with errors key encapsulation preliminary draft standard, 2023. URL: https://frodokem.org/files/FrodoKEM-ISO-20230314.pdf.

[Joh22a] Dougall Johnson. Apple M1 microarchitecture research. https://dougallj.github.io/applecpu/firestorm.html, 2022.

[Joh22b] Dougall Johnson. IDA (disassembler) and Hex-Rays (decompiler) plugin for Apple AMX, 2022. https://gist.github.com/dougallj/7a75a3be1ec69ca550e7c36dc75e0d6f.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3079856.3080246.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, sep 1979. doi:10.1145/355841.355847.

[LSH+22] Wai-Kong Lee, Hwajeong Seo, Seong Oun Hwang, Ramachandra Achar, Angshuman Karmakar, and Jose Maria Bermudo Mera. DPCrypto: Acceleration of post-quantum cryptography using dot-product instructions on GPUs. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(9):3591–3604, 2022. doi:10.1109/TCSI.2022.3176966.

[LSZH22] Wai-Kong Lee, Hwajeong Seo, Zhenfei Zhang, and Seong Oun Hwang. TensorCrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU. *IEEE Access*, 10:20616–20632, 2022. doi:10.1109/ACCESS.2022.3152217.

[MBB+21] José E. Moreira, Kit Barton, Steven Battle, Peter Bergner, Ramon Bertran, Puneeth Bhat, Pedro Caldeira, David Edelsohn, Gordon C. Fossum, Brad Frey, Nemanja Ivanovic, Chip Kerchner, Vincent Lim, Shakti Kapoor, Tulio Machado Filho, Silvia Melitta Mueller, Brett Olsson, Satish Sadasivam, Baptiste Saleil, Bill Schmidt, Rajalakshmi Srinivasaraghavan, Shricharan Srivatsan, Brian W. Thompto, Andreas Wagner, and Nelson Wu. A matrix math facility for Power ISA(TM) processors, 2021. https://arxiv.org/abs/2104.03142.

[MCL+18] S. Markidis, S. Chien, E. Laure, I. Peng, and J. S. Vetter. NVIDIA tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, Los Alamitos, CA, USA, may 2018. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPSW.2018.00091, doi:10.1109/IPDPSW.2018.00091.

[Nat17] National Institute of Standards and Technology. Post-Quantum Cryptography, 2017. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

[NG21] Duc Tri Nguyen and Kris Gaj. Fast NEON-based multiplication for lattice-based NIST post-quantum cryptography finalists. In Jung Hee Cheon and

Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 234–254, Cham, 2021. Springer International Publishing.

[Rod20] Andres Rodriguez. *Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, Oct. 2020.

[SBG+16] Ali Sazegari, Eric Bainville, Jeffry E. Gonion, III Gerard R. Williams, and Andrew J. Beaumont-Smith. Outer product engine. US patent US2018/074824 A1, 2016. URL: https://patents.google.com/patent/US20180074824A1/en.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997. doi:10.1137/S0097539795293172.

[WMS22] Finn Wilkinson and Simon 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)*, pages 135–140, 2022. doi:10.1109/PMBS56514.2022.00018.

[WZF+22] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuewu Wang, Jingqiang Lin, and Jiankuo Dong. A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 514–534, Cham, 2022. Springer Nature Switzerland.