Polynomial multiplication on embedded vector architectures

Hanno Becker[1], Jose Maria Bermudo Mera[2], Angshuman Karmakar[2], Joseph Yiu[1] and Ingrid Verbauwhede[2]

[1] Arm Ltd, Cambridge, UK, {firstname.lastname}@arm.com
[2] imec-COSIC, KU Leuven Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium {firstname.lastname}@esat.kuleuven.be

Abstract. High-degree, low-precision polynomial arithmetic is a fundamental computational primitive underlying structured lattice based cryptography. Its algorithmic properties and suitability for implementation on different compute platforms is an active area of research, and this article contributes to this line of work: Firstly, we present memory-efficiency and performance improvements for the Toom-Cook/Karatsuba polynomial multiplication strategy. Secondly, we provide implementations of those improvements on Arm® Cortex®-M4 CPU, as well as the newer Cortex-M55 processor, the first M-profile core implementing the M-profile Vector Extension (MVE), also known as Arm® Helium™ technology. We also implement the Number Theoretic Transform (NTT) on the Cortex-M55 processor. We show that despite being single-issue, in-order and offering only 8 vector registers compared to 32 on A-profile SIMD architectures like Arm® Neon™ technology and the Scalable Vector Extension (SVE), by careful register management and instruction scheduling, we can obtain a 3× to 5× performance improvement over already highly optimized implementations on Cortex-M4, while maintaining a low area and energy profile necessary for use in embedded market. Finally, as a real-world application we integrate our multiplication techniques to post-quantum key-encapsulation mechanism Saber.

Keywords: Post-Quantum Cryptography · Polynomial multiplication · IoT · Cortex-M55 · Cortex-M4 · M-profile Vector Extension (MVE) · Helium vector extension · Number Theoretic Transform (NTT) · Toom-Cook · Karatsuba

1 Introduction

The rapidly expanding Internet of Things (IoT) has an unprecedented impact on our digital ecosystem, so much that it is often termed the fourth industrial revolution. However, it also poses significant challenges. Firstly (sadly, often rather lastly), security: Absence or insufficient quality of security measures on IoT devices is a frequent headline. Secondly, performance: Embedded devices have much less compute resources than high-end consumer devices such as mobile phones or desktops. The dichotomy of the situation is that tight resource constraints need to be imposed on these devices to allow them to be cost-effective, but equally they limit performance and, importantly, often impede incorporating secure cryptographic protocols. Both aspects are in fact closely tied because (public-key) cryptography is computationally demanding. It is therefore a constant tussle of cryptographers and embedded engineers alike to fit secure and most advanced cryptographic protocols into constrained devices, *and* to have them perform sufficiently well to enable a wide range of applications. Unsurprisingly, an important criterion to assess a new cryptographic protocol is to evaluate its suitability for constrained devices.

The rise of Post-Quantum Cryptography (PQC) exacerbates the problem. Recalling the context first: Large-scale quantum computers are a threat to our current digital security infrastructure, especially public-key cryptography (PKC). The reason is that the security of today's predominant public-key cryptography – RSA and Elliptic Curve Cryptography (ECC) – rests on the hardness of computational problems which are known [Sho97, PZ03] to be solvable in polynomial time using a large quantum computer. Therefore, the National Institute of Standards and Technology (NIST) started a standardization procedure [NISa] aimed at the standardization of key-encapsulation mechanisms (KEM), public-key encryption (PKE) and signature schemes which resist the increased computational abilities of quantum computers – those schemes form the field of Post-Quantum Cryptography (PQC). These schemes will eventually replace their counterparts RSA and ECC used in our current public-key infrastructure. We refer to [NISb, Arm20] for more detailed introductions.

Post-Quantum Cryptography is highly relevant to the problem of balancing cost, performance and security in the IoT since it tends to have a higher resource footprint than classical public-key cryptography. It is thus vital to understand the demands and performance of PQC on embedded devices. Reflecting this, the Cortex-M4 processor has been designated by NIST as the reference platform for PQC on embedded systems.

The most prominent class of PQC is that of schemes based on structured lattices, and the underlying computational workload is the multiplication of polynomials of large degree and low coefficient precision. This problem has been studied extensively, and two approaches prevailed: Multiplication via the Toom-Cook-Karatsuba algorithms [Too63, Coo66, KO62], and multiplication via the Number Theoretic Transform (NTT). We are thus interested in the constraints and performance of these algorithms on embedded systems. Structurally, both approaches are very similar and amenable to the same tradeoffs [KMRV18, KRS19, MKV20]. The main benefit of Toom-Cook is the absence of modular arithmetic, which the NTT heavily relies on. Its main drawback, and the primary factor determining its memory-usage, is that it expands the input both before ("evaluation") and during ("point multiplication") the core of the multiplication routine, while the NTT operates in-place.

On high-end processors, performance improvements can often be obtained by leveraging Single Instruction Multiple Data (SIMD) instruction set extensions, such as Arm's Neon instruction set on the R- and A-profiles of the Arm architecture, or the Intel® Advanced Vector Extension (AVX). On embedded systems, however, SIMD instruction extensions are challenging within the usually tight power, cost and area constraints. The M-profile Vector Extension (MVE), or Helium vector extension, is a rather recent addition to the M-profile of the Arm architecture which promises to introduce the performance benefits of vector extensions to embedded systems while maintaining a low gate and power profile. The Cortex-M55 processor is the first implementation of the Helium instruction set. The primary motivation of this work is to evaluate how this future generation of low-cost and low-energy processors fare in the context of the next generation of public key cryptography.

Contributions We contribute to the study of viability and performance of PQC on embedded devices. We study two themes: Firstly, how to reduce the resource demands of PQC for low-end embedded devices such as the Cortex-M4 CPU. Secondly, how to leverage the Helium instruction set on embedded devices implementing the Helium vector extension, focusing on the Cortex-M55 processor. Our specific contributions are as follows:

- We revive a known but little used variant of Toom-Cook and Karatsuba called *striding Toom-Cook/Karatsuba* to reduce the memory-usage of polynomial multiplication. In a nutshell, while expansion during evaluation remains, a suitable input reordering allows to avoid size-doubling on the point-multiplication, thereby saving almost 50% in memory and also accelerating the interpolation step. See Table 1. We implement the striding variant of Toom-Cook/Karatsuba on the Cortex-M4 processor.

- We implement the striding Toom-Cook/Karatsuba and a 32-bit degree-256 negacyclic NTT on the Cortex-M55 processor, based on the Helium instruction set. We report an ≈ 5× speedup of our striding Toom-Cook/Karatsuba implementation compared to previous Cortex-M4 implementations, and an ≈ 3.5× speedup of our implementation of the NTT compared to the fastest NTT on Cortex-M4 [CHK+21]. NTT-based multiplication remains faster, albeit by a smaller margin than on Cortex-M4.
- We provide an introduction to the M-profile Vector Extension and to the Cortex-M55 processor, highlight its similarities and differences compared to other vector extensions, and share our experience on what to look out for to get the most out of the instructions. We hope that this will enable researchers to build on our work and study the use of the Helium vector extension for other workloads, esp. PQC schemes.

Table 1 Comparing features of multiplication based on FFT/NTT and Toom-Cook

	NTT [CHK+21]	Toom-Cook
Lazy interpolation	✓	✓ [MKV20]
No expansion during evaluation	✓	✗
No expansion during inner multiplication	✓	✓[**This work**]

Code The code generation tooling and resulting Helium assembly for Cortex-M55 are available at `https://gitlab.com/arm-research/security/pqmx`. The repository also contains detailed instructions for how to explore Helium using a freely available functional model of the Cortex-M55. The Cortex-M4 code will be made available on the Saber repository `https://github.com/KULeuven-COSIC/SABER`.

2 Preliminaries

We denote $\mathbb{Z}_n$ the ring of integers modulo $n \in \mathbb{Z}$. If $n = q$ is a prime, we also write $\mathbb{F}_q$. Unless stated otherwise, R denotes a commutative ring. We denote $R[X]$ the polynomial ring over R, and for a monic polynomial $P \in R[X]$ we denote $R[X]/(P)$ the quotient of $R[X]$ obtained by identifying polynomials with the same remainder modulo P. We will mostly work with $\mathbb{F}_q[X]/(X^n + 1)$ or $\mathbb{Z}_{2^k}[X]/(X^n + 1)$.

2.1 Polynomial multiplication

In this section we survey the most prominent sub-quadratic polynomial multiplication strategies: Toom-Cook/Karatsuba and the Number Theoretic Transform (NTT). Good expositions exist in the literature, so we will be brief. See e.g. the excellent [Ber01].

2.2 Multiplication by evaluation

The idea behind Toom-Cook/Karatsuba and NTT-based multiplication is the same: Multiplication via evaluation. Given a ring R, two polynomials $f, g \in R[X]$ and fixed elements $\alpha_1, \ldots, \alpha_n \in R$, we have $(fg)(\alpha_i) = f(\alpha_i)g(\alpha_i)$. Each equation puts a constraint on fg, and by choosing n large enough, we hope to be able to recover fg from $(fg)(\alpha_i)$.

Formally, consider the evaluation homomorphism

$$\widetilde{\mathrm{ev}}_\alpha : R[X] \xrightarrow{f \longmapsto (f(\alpha_1),\ldots,f(\alpha_n))} R \times R \times \cdots \times R.$$

It factors through the quotient $R[X]/\prod_i (X - \alpha_i)$, which as a set is canonically identified with $R[X]^{<n}$, the set of polynomials of degree $< n$. Depending on the context, we can view the evaluation homomorphism either as $\mathrm{ev}_\alpha : R[X]/\prod_i(X - \alpha_i) \to R^n$, or as $\mathrm{ev}_\alpha : R[X]^{<n} \to R^n$. It is an isomorphism if and only if for all $i \neq j$, $\alpha_i - \alpha_j$ is invertible in R. For example, this holds if R is a field and the α_i are pairwise distinct. If ev_α is bijective, we can thus fully recover a polynomial $f \in R[X]^{<n}$ of degree $< n$ from its evaluations $f(\alpha_i)$; equally, we can recover the residue of $f \in R[X]$ modulo $\prod_i(X - \alpha_i)$ from $f(\alpha_i)$.

This technique can be used to calculate products fg if $f, g \in R[X]/\prod_i(X - \alpha_i)$ or if $f \in R[X]^{\leq a}, g \in R[X]^{\leq b}$ with $a + b < n$: In both cases, we first compute $f(\alpha_i), g(\alpha_i)$ – the "evaluation step" – then the products $f(\alpha_i)g(\alpha_i)$ – the "point multiplication step" – and finally recover fg from its evaluations at the α_i – the "interpolation step".

It is common to add the "evaluation at ∞", giving $\mathrm{ev}_\alpha^\infty : R[X]^{\leq n} \to R^{n+1}$ defined by $f \mapsto (f(\alpha_1), \ldots, f(\alpha_n), f(\infty))$, where $f(\infty)$ is the coefficient of X^n. This extended evaluation $\mathrm{ev}_\alpha^\infty$ is bijective if and only if ev_α is. In this case, we can therefore a polynomial of degree $\leq n$ from its evaluations at $\alpha_1, \ldots, \alpha_n$ plus its n-th coefficient.

2.2.1 Toom-Cook/Karatsuba

Karatsuba's algorithm instantiates the "multiplication by evaluation" blueprint with $\alpha_0 = 0, \alpha_1 = 1, \alpha_2 = \infty$, thereby allowing to compute the product of two degree-1 polynomials $f = a + bX$ and $g = c + dX$ in terms of their evaluations at $0, 1, \infty$. Concretely,

$$f(0) = a, f(1) = a + b,\ f(\infty) = b,\ g(0) = c,\ g(1) = c + d,\ g(\infty) = d,$$

and we recover fg from the point multiplications $f(0)g(0)$, $f(1)g(1)$, and $f(\infty)g(\infty)$ via

$$\begin{aligned} fg &= (a + bX)(c + dX) = ac + (ad + bc)X + bdX^2 \\ &= f(0)g(0) + (f(1)g(1) - f(0)g(0) - f(\infty)g(\infty))X + f(\infty)g(\infty)X^2. \end{aligned}$$

This works for any ring R since $\alpha_0 - \alpha_1 = -1$ is invertible.

The Toom-Cook algorithm generalizes Karatsuba's. Concretely, we are interested in 4-way Toom-Cook defined by the points $0, \infty, \pm 1, \pm\frac{1}{2}, 2$ and applied over a ring R where $2^k = 0$ for some k – we will mainly be looking at $\mathbb{Z}_{2^{13}}$. This setting has been introduced and studied in [KMRV18], and a few subtle points should be noted – see [KMRV18] for details: Firstly, to make sense of the fractional evaluations in $\mathbb{Z}$, one scales by $\alpha^{\deg(f)}$, so that for $f = a_0 + a_1X + a_2X^2 + a_3X^3$ we have $f\left(\frac{1}{2}\right) := 8a_0 + 4a_1 + 2a_2 + a_3$. Secondly, ev_α is not an isomorphism: In fact, if $2^k = 0$ in R but $2^{k-1} \neq 0$, the polynomial $f := 2^{k-1}X(X + 1)$ satisfies $f(\alpha) = 0$ for all $\alpha \in \{0, \infty, \pm 1, \pm\frac{1}{2}, 2\}$. However, [KMRV18] explains that multiplication by evaluation still works if we temporarily operate with $k + 3$ bits of precision – that is, we lift representatives from $\mathbb{Z}_{2^k}$ to $\mathbb{Z}_{2^{k+3}}$ below 2^k, compute evaluation, multiplication and interpolation in $\mathbb{Z}_{2^{k+3}}$, and reduce to $\mathbb{Z}_{2^k}$ in the end.

2.2.2 Iterating Toom-Cook/Karatsuba

Toom-Cook/Karatsuba multiplication is usually applied iteratively, with R being a polynomial ring itself, transforming a product of two polynomials of large degree into numerous products of polynomials of smaller degree, which can then be computed by more direct means such as schoolbook multiplication. We explain this for k-way Toom-Cook.

Given a polynomial $f \in R[X]^{<n}$ with $n = kr$, we can group its monomials into the blocks $\{\frac{in}{k}, \frac{in}{k} + 1, \ldots \frac{(i+1)n}{k} - 1\}, i = 0, \ldots, r - 1$, and thereby express f as

$$\tilde{f} = f_0(X) + f_1(X)Y + \cdots + f_{k-1}(X)Y^{k-1}, \quad \deg_X(f_i) < n/k, \quad Y = X^{\frac{n}{k}}.$$

Formally, we consider lifts under $R[X]^{<n/k}[Y]^{<k} \to R[X]^{<n}, Y \mapsto X^{n/k}$. Treating $R' := R[X]$ as the coefficient ring, we then apply Toom-Cook to compute $R'[Y]^{<k} \times R'[Y]^{<k} \to$

$R'[Y]^{<2k-1}$ via $2k-1$ multiplications of elements in R' and substitute $Y = X^{n/k}$ afterwards. Ultimately, this computes a degree-n product via $2k-1$ products of degree n/k.

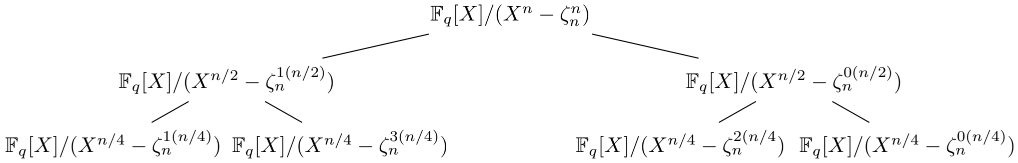
Toom-Cook and Karatsuba can be combined. E.g., one layer of 4-way Toom-Cook and two layers of Karatsuba reduce a degree-$16n$ product to 63 degree-n products.

2.2.3 Number Theoretic Transform

The *Number Theoretic Transform* (NTT) instantiates the "multiplication by evaluation" blueprint with $R = \mathbb{F}_q$ for a prime q with $n|q-1$. Then, it is known that $\mathbb{F}_q$ has a primitive n-th root of unity ζ_n, and that $X^n - 1 = \prod_i (X - \zeta_n^i)$, and ev_α is an isomorphism

$$\mathrm{NTT} : \mathbb{F}_q[X]/(X^n - 1) \xrightarrow{f \mapsto (f(1), f(\zeta), \ldots, f(\zeta^{n-1}))} \mathbb{F}_q^n.$$

If $n = 2^k$, then NTT can be computed iteratively as indicated in the following figure:


$\mathbb{F}_q[X]/(X^n - \zeta_n^n)$
$\mathbb{F}_q[X]/(X^{n/2} - \zeta_n^{1(n/2)})$
$\mathbb{F}_q[X]/(X^{n/2} - \zeta_n^{0(n/2)})$
$\mathbb{F}_q[X]/(X^{n/4} - \zeta_n^{1(n/4)})$
$\mathbb{F}_q[X]/(X^{n/4} - \zeta_n^{3(n/4)})$
$\mathbb{F}_q[X]/(X^{n/4} - \zeta_n^{2(n/4})$
$\mathbb{F}_q[X]/(X^{n/4} - \zeta_n^{0(n/4)})$

Here, the k-th entry from the right in the i-th layer is $\mathbb{F}_q[X]/(X^{n/2^i} - \zeta_n^{\mathrm{rev}_i(k)(n/2^i)})$, where rev_i is the bit-reversal on $i+1$ bits. For example, $\mathrm{rev}_1(2) = \mathrm{rev}_1(\texttt{0b10}) = \texttt{0b01} = 1$.

This strategy is akin to the Cooley-Tukey algorithm for the FFT and achieves complexity $n \log_2(n)$. Since NTT is a ring isomorphism, we can compute the product of $a, b \in \mathbb{F}_q[X]/(X^n - 1)$ as $\mathrm{NTT}^{-1}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b))$, where $\circ$ is pointwise multiplication in $\mathbb{F}_q^n$.

We consider the NTT from an implementation view. Every transition is of the form

$$R[X]/(X^{2n} - \alpha^2) \to R[X]/(X^n - \alpha) \times R[X]/(X^n + \alpha).$$

In terms of the standard monomial basis, $R[X]/(X^{2n} - \alpha^2) \to R[X]/(X^n - \alpha)$ is given by $f_0 + X^n f_1 \mapsto f_0 + \alpha f_1$, while $R[X]/(X^{2n} - \alpha^2) \to R[X]/(X^n - \alpha)$ is given by $f_0 + X^n f_1 \mapsto f_0 - \alpha f_1$. The transformation $(a, b) \mapsto (a + \zeta b, a - \zeta b)$ is called a *Cooley-Tukey* (CT) butterfly. The *Gentleman-Sande* (GS) butterfly $(a, b) \mapsto (a + b, \zeta^{-1}(a - b))$ reverses the Cooley-Tukey butterfly up to $\frac{1}{2}$. The CT and GS are central primitives to implement for an NTT-based polynomial multiplication; the third is point-wise multiplication in $\mathbb{F}_q$.

2.3 Modular Arithmetic in $\mathbb{F}_q$

We briefly recall two prominent algorithms in the context of modular multiplication: Barrett reduction and Montgomery multiplication.

The central problem is as follows: Given a modulus $n \in \mathbb{Z}$ and a "large" $a \in \mathbb{Z}$, find "small" representatives of a modulo n. Two choices stand out: Firstly, the canonical unsigned representative $a \bmod^+ n$ is the unique representative of a modulo n in the interval $\{0, 1, \ldots, n-1\}$. Secondly, the canonical signed representative $a \bmod^\pm n$ is the unique representative of a modulo n in the interval $\{-\lceil \frac{n}{2} \rceil, -\lceil \frac{n}{2} \rceil + 1, \ldots, \lfloor \frac{n}{2} \rfloor\}$. These can naïvely be calculated as $a \bmod^\pm n = a - n \lfloor \frac{a}{n} \rceil$ and $a \bmod^+ n = a - n \lfloor \frac{a}{n} \rfloor$, but such an approach is inacceptable from a performance perspective since division by n is expensive in general. Barrett and Montgomery reduction are two ways around this by trading the division by n for a division by a 2-power, which can be cheaply implemented as a bitshift.

2.3.1 Barrett reduction

The idea behind Barrett reduction is to pick ℓ such that $2^\ell > n$ and approximate $\lfloor \frac{a}{n} \rceil = \left\lfloor a\frac{2^\ell}{n}/2^\ell \right\rceil \approx \left\lfloor a \left\lfloor \frac{2^\ell}{n} \right\rceil /2^\ell \right\rceil$, where $C := \left\lfloor \frac{2^\ell}{n} \right\rceil$ can be precomputed. See Algorithm 1. For $a < 2^\ell$ and ℓ bounded by the machine word size, the resulting modular reduction $a - n \cdot \lfloor \frac{aC}{2^\ell} \rceil$ is amenable for direct mapping to many instruction sets: It is a combination of a high-multiply with rounding for $x \mapsto \lfloor \frac{x \cdot C}{2^\ell} \rceil$ and a low-multiply-accumulate for $x, a \mapsto a - n \cdot x$. We will discuss the details in the case of the Helium vector extension in Section 6.4.1.

2.3.2 Montgomery reduction and multiplication

Montgomery reduction is a way to compute a representative a' s.t. $2^\ell a' \equiv a$ modulo n if n is odd; in other words, a' is a small representative of the quotient $\frac{a}{2^\ell}$ *computed in* $\mathbb{Z}_n$ – note that the latter expression makes sense because n is odd, so 2^ℓ is invertible modulo n.

If a is already divisible by 2^ℓ in $\mathbb{Z}$, we can set $a' := \frac{a}{2^\ell}$, computed in $\mathbb{Z}$. Otherwise, if we pick $k := a \cdot n^{-1} \bmod^{\pm} 2^\ell$, where $a \cdot n^{-1}$ is computed in $\mathbb{Z}_{2^\ell}$, we have $n \cdot k \equiv a$ modulo 2^ℓ by construction, and so $a - n \cdot k$ is divisible by 2^ℓ. We can thus set $a' := (a - n \cdot k)/2^\ell$.

We are interested in Montgomery reduction for ℓ bounded by the machine word size, and $a < 2^{2\ell}$ a product of two single-width values $x, y < 2^\ell$. In this case, it can be expressed in single-width operations: two high-multiplications $x, y \mapsto \frac{xy}{2^\ell}$, $k \mapsto \frac{n \cdot k}{2^\ell}$, and two low multiplications $x, y \mapsto xy \bmod^{+} 2^\ell$, $a \mapsto a \cdot n^{-1}$ in $\mathbb{Z}_{2^\ell}$. See Algorithm 2 for both the double-width and the single-width description of Montgomery multiplication. We will discuss the details in the case of the Helium vector extension in Section 6.4.1.

Alg. 1: Barrett reduction

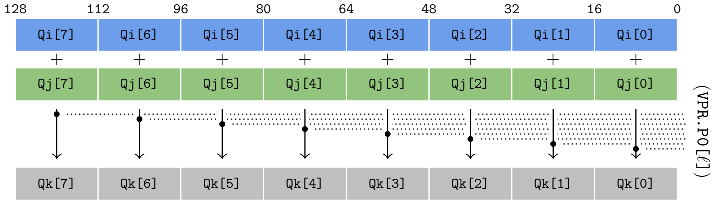
```
In: $n$ modulus, $2^\ell > n$, $z < 2^\ell$.
Out: Barrett reduction of $z$
$t \leftarrow \left\lfloor z \left\lfloor \frac{2^\ell}{n} \right\rceil /2^\ell \right\rceil$
$c \leftarrow nt$
return $z - c$
```

Alg. 2: Montgomery multiplication

```
In: $n$ odd modulus, $2^\ell > n$, $a, b < 2^\ell$.
In: $\omega$ modular inverse of $n$ w.r.t. $2^\ell$.
Out: Montgomery multiplication of $a, b$
$h \leftarrow ab$                      $h \leftarrow \lfloor \frac{ab}{2^\ell} \rfloor$, $l \leftarrow ab \bmod^{+} 2^\ell$
$r \leftarrow h\omega \bmod^{\pm} 2^\ell$       $r \leftarrow l\omega \bmod^{\pm} 2^\ell$
$c \leftarrow r \cdot n$                  $c \leftarrow \lfloor \frac{r \cdot n}{2^\ell} \rfloor$
return $(h - c)/2^\ell$         return $h - c$
```

3 M-profile Vector Extension (MVE)

In this section we give an introduction to the M-profile Vector Extension (MVE), also referred to as Arm® Helium™ Technology. We will focus on aspects of MVE that are relevant to the algorithms studied in this paper, and refer to [Arme, Mar21] more exhaustive introductions to the Helium instruction set, as well as to [Armb] for the full reference. We also found the blog series [Armc, Part 1-4] very useful.



```
(VPST)
VADD(T).u16 Qk, Qi, Qj
```

Figure 1 Parallel 16-bit addition using `VADD.u16`, with/without predication

3.1 A primer on vector architectures

Vector architectures promise to accelerate computation through the introduction of *Single Instruction Multiple Data* (SIMD) operations: They add a set of wide registers that are viewed as vectors of equal-sized blocks called lanes, and provide instructions that operate on those lanes in parallel. For example, a 128-bit vector register can be viewed as a length-8 vector of 16-bit elements, and a 16-bit vector add instruction (e.g. `VADD.u16` in Helium, see Figure 1) adds the lanes of two such vectors, storing the results in the lanes of a third vector – thus the term “Single Instruction, Multiple Data”.

It is intuitively clear that vectorization holds great potential for accelerating computational workloads. However, the exact order of speedup is influenced by a large number of architectural and microarchitectural parameters, which tend to also have implications on both hardware cost and software constraints. Choosing a specific set of parameters is therefore a careful tradeoff tailored to the target domain. We provide some details in the rest of this section to help the reader put the design choices of Helium and Cortex-M55 and their implications on hardware and software into perspective.

Architecturally, two parameters impacting the speedup potential of a SIMD architecture are the expressiveness of its instruction set and the size of the vector registers: A SIMD architecture with 256-bit vectors, such as provided by AVX2 or 256-bit implementations of Arm’s Scalable Vector Extension (SVE), performs twice as many operations per instruction compared to SIMD on 128-bit vectors, such as provided by Helium or Neon. Also, specialized instructions for common operations or operation sequences (such as multiply-accumulate) lower the number of instructions required to express a computation. It is clear, however, that tailoring those parameters for performance comes at a significant cost in hardware.

Another architectural parameter of interest is the number of architectural vector registers, which constitutes a less obvious tradeoff: To begin, fewer vector registers clearly imply a lower hardware cost but larger constraints on software. However, the extent to which those constraints imply a performance loss is less obvious, and very much depends on the workload under consideration.

Microarchitecturally, primary parameters influencing the potential of vectorization are the number of SIMD execution units, the latency and throughput of vector instructions, and out-of-order execution capabilities. For example, the Cortex-X1 CPU has four Neon execution units and therefore twice the vectorization throughput potential compared to a Cortex-A78 CPU with two Neon execution units. Whether such potential can be realized, however, heavily depends on the nature of the workload (e.g. data flow, instruction distribution) and on the scheduling of instructions: In the case of high-end out-of-order CPUs, the latter is mostly taken care of by the CPU itself – even to the extent that multiple iterations of a loop can be executed in parallel to create enough independent data streams to keep all SIMD units busy. Out-of-order execution also relaxes constraints on software since instruction scheduling happens dynamically, but the downside of those powerful microarchitectural features is a significant hardware cost. In-order execution, such as found in all M-profile processors as well as low-end A-profile processors, considerably improves cost and efficiency, but there it is the responsibility of the programmer/compiler to keep the SIMD unit(s) busy through careful instruction scheduling.

To summarize: Within the design space of SIMD architectures and microarchitectures, there are tradeoffs between hardware cost and performance potential, and hardware cost and software constraints towards reaching this potential. Whether those constraints can be met is a workload-specific problem. We will next describe the tradeoffs and constraints in the case of the Helium vector architecture and the Cortex-M55 microarchitecture. In Section 6, we will discuss how to solve the software constraints in the case of the PQC workloads under consideration, thereby realizing the full performance potential of Helium + Cortex-M55 while maintaining a low gate and power profile.

3.2 Introduction to MVE

The M-Profile Vector Extension (MVE) is a feature of the Armv8.1-M architecture [Armb, Arme], primarily for signal processing and machine learning applications. MVE is also referred to as the Helium vector extension, in alignment with the Arm® Neon™ Technology architecture extension for A-profile processors [Arma, Section C.3.5]. However, while there are similarities between the Helium instruction set and the Neon instruction set, the Helium vector extension is a new ground-up architecture design specifically tailored for the extreme area and energy efficiency required in typical Cortex-M applications.

The vector file Like the Neon instruction set, the Helium vector extension uses 128-bit vector registers. There are 8 vector registers, compared to 32 in the Neon instruction set, a design choice which can be approached from multiple angles: Firstly, it results in a lower gate count and power profile. Secondly, the lower vector count is compensated for by multiple features, including the presence of numerous instructions utilizing both vector and scalar registers. The suitability of scalar-vector operations is a main difference between M- and A-profile: On A-profile implementations, the physical distance between vector and scalar register file is too large to allow for low-latency scalar-vector operations. The smaller scale of M-profile CPUs means that these operations become feasible.

Implication: Algorithm design A common technique for vectorization of computational workloads is batching, whereby multiple instances of a higher-level construct are computed in parallel, rather than the computation of a single instance of the construction being vectorized and accelerated.

Since batching tends to use a large number of vector registers but a low number of general purpose registers (GPR), we found it not suitable for use with the Helium vector extension except for very small computations. Instead, we found that the Helium instruction set works well for single-instance speedup, using a mix of vector and GPR file. We will see this in the example of schoolbook multiplications below.

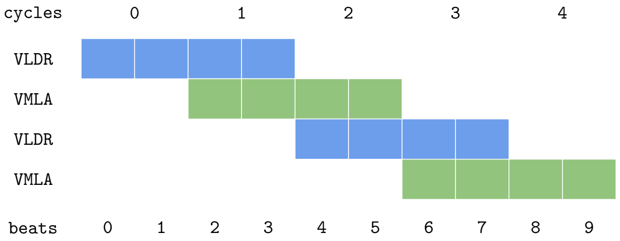


Figure 2 Instruction overlapping on a dual-beat implementation of Helium

Instruction Overlapping The idea of overlapping the execution of instructions is fundamental to CPU microarchitecture and at the heart of the concept of a pipeline. However, it is commonly hidden from the programmers. The Helium architecture extension deviates from this by making instruction overlapping part of the *architecture*.

The execution of each vector instruction is architecturally subdivided into four 32-bit parts called *beats*. The operation of each beat may affect multiple lanes: For example, the `VADD.u16` operation operates on two 16-bit lanes per beat. The Helium architecture extension requires that each beat of an instruction is executed in-order, but it allows implementations to execute beats from the following instruction whilst still executing beats from the previous instruction [Armb]. For example, a vector load `VLDR` instruction can be overlapped with a vector multiply accumulate `VMLA` as shown in Figure 2.

Because the load/store units and arithmetic units can both be kept busy at the same time, even on a single issue CPU, the architected instruction overlap can significantly improve performance at minimal area overheads. Moreover, implementations benefit from the fact that instruction overlapping is architectural because they do not need logic to maintain the illusion of atomic execution: The Helium architecture extension allows instructions to be interrupted between architectural beats.

Implication: Instruction design Support for instruction overlapping has to be built into the design of the architecture: For example, consider a data dependency between two consecutive instructions `A,B` on a dual-beat system. In this case, beats `B0,B1` of `B` can only commence if the inputs (produced by beats `A0,A1` of `A`) are available. Generally, this is supported by instructions whose beats describe the same amount of work and operate on lanes independently, and from right to left. This explains why there is a 128-bit left shift Helium instruction `VSHLC` (see Figure 3), but no 128-bit right shift. Long multiplies are another example: The inputs to `VMULL` in the Neon instruction set are lower and upper halves of 128-bit vectors. This would not fare well with beat-wise execution, and instead there are instructions `VMULL{B,T}` in Helium operating on even or odd parts of the inputs, which is beat-friendly. We again refer to [Armc, Part 1] for details and more examples.

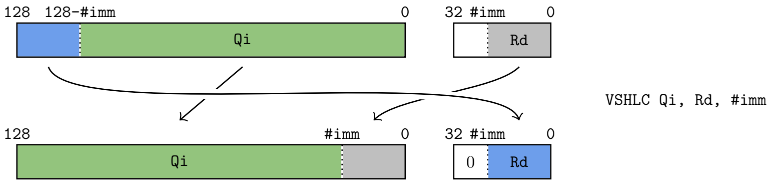


Figure 3 The semantics of the beat-friendly long left shift with carry instruction `VSHLC`

Implication: Programming discipline There are three main ways to develop code for the Helium instruction set: Auto-vectorization, intrinsics and handwritten assembly. In the former cases, the compiler has the responsibility of instruction scheduling. When handwriting Helium instructions, however, it is important to carefully study the ordering of instructions in order to make optimal use of instruction overlapping and the available computational resources. For example, the sequence in Figure 2 would run 50% slower on a system with 64-bit data paths if we'd batch the multiplies and load operations. Generally, alternating instructions of different nature (load/store, add/sub, multiply) is preferred.

Low overhead loops Armv8.1-M adds the Low Overhead Branch Extension, allowing software to indicate how many iterations of a loop will be executed and let the hardware do counter modification and branching. Importantly, the loop-end instruction `LE` may cache details of the loop and allow processors to optimize future iterations, skipping subsequent `LE` instructions and overlapping instructions across loop iterations. The following iterations of the loop can therefore perform as if the loop had been unrolled at compile-time.

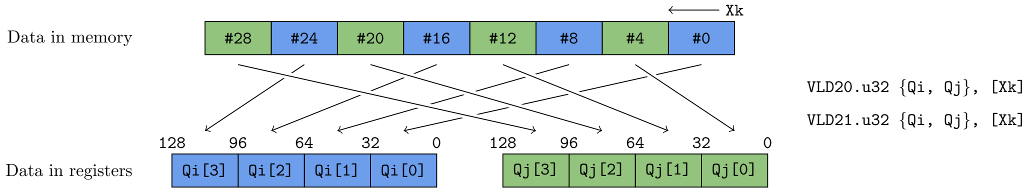


Figure 4 32-bit deinterleaving load at stride 2

Interleaving memory operations The Helium instruction set adds support for deinterleaving data during loads via the instruction sequences `VLD2{0-1}` and `VLD4{0-3}`, and interleaving data during stores via `VST2{0-1}` and `VST4{0-3}`. Logically, those correspond to a load of $\{2, 4\} \cdot 16$ bytes of data, combined with $2 \times \_ \leftrightarrow \_ \times 2$ and $4 \times \_ \leftrightarrow \_ \times 4$ transpositions. It should be noted that the feasibility of such instructions in an embedded architecture is a non-trivial question, as the required interleaving logic has to be implementable at low cost. We refer to [Armc, Part 2] for details on how this is achieved. See Figure 4 for an illustration of `VLD2{0,1}`. The instructions can appear in any order and be interleaved with other operations in support of instruction overlapping.

```
VPTE.s32  LT, Qi, Qj // If Qi[r] < Qj[r]
VSUBT.s32 Qk, Qj, Qi // Then Qk[r] := Qj[r] - Qi[r]
VSUBE.s32 Qk, Qi, Qj // Else Qk[r] := Qi[r] - Qj[r]
```

Listing 1 Computing the lane-wise absolute difference via lane predication

Lane predication The Helium vector architecture supports restricting the effect of a vector instruction to a subset of lanes through *lane predication*. Here, the subset of "active" lanes is determined by a dedicated register `VPR.P0` (see e.g. Figure 1) which can be set directly or defined via a "predicate" over contents of vector registers. Moreover, the predicate can be inverted, thereby allowing for the construction of `if-then-else` style instruction sequences using predication. For example, Listing 1 shows how to compute the absolute difference between two vectors using predication. See also [Mar21, Section 4.4].

3.3 Cortex-M55

The Cortex-M55 processor is the first implementation of the Arm v8.1-M architecture, including optional support for the Helium vector extension. Cortex-M55r1 is the most recent revision of the Cortex-M55 processor, providing additional features and performance optimizations. We give a very brief introduction here and refer to [Armd] for details.

Pipeline and memory The Cortex-M55 processor has a 5-stage in-order pipeline when Helium is included, and instruction scheduling is single-issue with a few exceptions. In addition to cache-able main memory, the Cortex-M55 processor supports tightly coupled memory (TCM) interfaces for instructions and data, allowing fast and deterministic memory access which is useful for real-time or performance critical applications. The total D-TCM bandwidth is 128-bit/cycle, allowing 64-bit/cycle bandwidth for processing such as vector processing, plus 64-bit/cycle for DMA transfers to/from TCM running in the background. We locate all our code and data in TCM for our measurements.

Vector processing The Cortex-M55 processor is a dual-beat implementation of the Helium instruction set which supports instruction overlapping. This means that it takes two cycles to perform the work of a vector instruction, and that parallel execution of the next instruction can commence in the second cycle, computing resources permitting. Figure 2 shows how a sequence of alternating `VMLA`, `VLDR` would perform on the Cortex-M55 processor, absent of other hazards and assuming data and code in TCM.

The Cortex-M55 processor has separate units for load/store (e.g. `VLDR,VSTR`), additive integer operations (e.g. `VADD,VSUB`), and multiplicative integer as well as floating point operations (e.g. `VMUL`). We therefore tried to achieve an alternation of those three kinds of operations as a first step towards good instruction overlapping. The assembly snippets below reflect the different instruction types through their color.

4 Memory efficient striding Toom-Cook

In this section, we describe "striding" Toom-Cook multiplication. While not new – see e.g. [Ber01] – it does not appear to have been used in lattice-based cryptography before.

Recall from Section 2.2.2 that using classical Toom-Cook k-way for large-degree multiplication, the inputs in $R[X]$ are lifted to the isomorphic ring $R[X][Y]/(X^k - Y)$ via

$$\begin{aligned} &a_0 + a_1 X + \cdots + a_{n-2}X^{n-2} + a_{n-1}X^{n-1} \\ &= (a_0 \cdots + a_{k-1}X^{k-1}) + (a_k \cdots + a_{2k-1}X^{k-1})Y \cdots + (a_{n-k} \cdots + a_{n-1}X^{k-1})Y^{r-1} \end{aligned}$$

where $r := \frac{n}{k}$, before k-way Toom-Cook is applied to $R'[Y]^{<r}$ over the base ring $R' = R[X]$. The problem with this approach is that even if, ultimately, we care about size-preserving multiplication in $R[X]/(X^n + 1)$, the base multiplication in $R[X]^{<k}$ is length-doubling: The polynomial degree is too small for wraparound. This size-doubling during point multiplication in turn raises the memory usage.

This can be improved by using a different ring isomorphism, effectively changing the arrangement of X and Y: We lift the polynomials from $R[X]$ to $R[Y][X]/(X^k - Y)$ as

$$\begin{aligned} &(a_0 + a_k Y + \cdots + a_{(r-1)k}Y^{r-1})X^0 + (a_1 + a_{k+1}Y + \cdots + a_{(r-1)k+1}Y^{r-1})X^1 + \cdots \\ =&\sum_{i<k}(a_i + a_{k+i}Y + \cdots + a_{(r-1)k+i}Y^{r-1})X^i \end{aligned}$$

and apply k-way Toom-Cook to $R'[X]^{<k}$ and base ring $R' = R[Y]^{<r}$. The point is: If our target ring is $R[X]/(X^n + 1) = R[X]/(X^{rk} + 1)$, then because $Y = X^k$, our new base $R' = R[Y]$ is in fact $R[Y]/(X^r + 1)$, another negacyclic polynomial ring with size-preserving multiplication. In this way, the polynomial reduction can be moved to the point multiplication, significantly reducing memory usage of Toom-Cook multiplication. Another advantage is that the interpolation operates on smaller polynomials and is thus faster.

Difference in evaluation: Striding Toom-Cook/Karatsuba evaluation differs from classical evaluation only through the memory access pattern, which is no longer contiguous but at stride k. In Section 5 and Section 6 we discuss how to overcome this with low performance-penalty on the Cortex-M4 and Cortex-M55 processors. Alternatively, one can permute the input upfront and thereby reduce striding evaluation to classical evaluation. It depends on the context whether this is feasible – we comment on the case of Saber below.

Differences in interpolation: While classical interpolation gives $ab = \sum_{i<2k-1} c_i(X)Y^k$ with $\deg_X(c_i) < 2(n/k)$ and $Y = X^{n/k}$, striding interpolation gives $ab = \sum_{i<2k-1} c_i(Y)X^k$ with $\deg_Y(c_i) < n/k$ and $Y = X^k$. Resolving the substitution $Y = X^{n/k}$ in the classical context means overlapping the upper half of c_i with the lower half of c_{i+1}. Resolving the substitution $Y = X^k$ in the striding context gives $ab = \sum_{i<k}(c_i(X^k) + X^k c_{k+i}(X^k))X^i$ – computing this thus involves a negacyclic rotation of c_{k+i} and addition onto c_k.

4.1 Application: Saber

Our primary interest in striding Toom-Cook/Karatsuba multiplication lies in its use for the Saber PQC scheme. We refer to [BMD+20] or the section pertaining to Toom-Cook multiplication in the supplementary material for details, but the important point is that the ring $\mathbb{Z}_{2^{13}}[X]/(X^{256}+1)$ underlying Saber is amenable to iterated application of the striding method. Concretely, a striding Toom-Cook/Karatsuba based multiplication for Saber would use a hybrid of one layer of 4-way Toom-Cook to reduce from $\mathbb{Z}_{2^{13}}[X]/(X^{256} + 1)$ to $\mathbb{Z}_{2^{13}}[X]/(X^{64} + 1)$, one or two layers of Karatsuba to reduce from $\mathbb{Z}_{2^{13}}[X]/(X^{64} + 1)$ to $\mathbb{Z}_{2^{13}}[X]/(X^{\{16,32\}} + 1)$, and schoolbook multiplication.

The polynomials used in Saber are generated from packed data on a per-coefficients level. We expect that it possible to merge the upfront permutation of the input into the key generation process with small negligible penalty, thereby reducing the evaluation step to classical Toom-Cook/Karatsuba, but leave exploring the details for future work.

5 Implementation: Cortex-M4

In this section, we describe the implementation aspects of the striding version of Toom-Cook and Karatsuba on Cortex-M4 processors.

5.1 Toom-Cook/Karatsuba

As explained in Section 4, there are two fundamental differences between the classical and the striding variants of Toom-Cook multiplication: The access pattern to the coefficients of the operands and the memory expansion of the products. The sequence of arithmetic operations performed during evaluation and interpolation is the same for both variants.

The read accesses occur during the evaluation and are performed with offset 64 for the classical Toom-Cook whereas 4 consecutive coefficients at a time are required for the striding version. Although the access pattern of the striding variant is more regular, this is an issue for exploiting the DSP extensions to carry out two operations on halfword registers in parallel. To circumvent this problem, the polynomials can be stored with a custom memory layout, in which case the complexity is moved to the packing operation, or the coefficients of the polynomial can be packed with an offset of 4 after being loaded into the register. Since we implement the entire multiplication we opt for the latter. The instruction overhead is equal to the number of word loads, effectively as if we only performed halfword loads.

The write operations happen during interpolation, when the result is computed from the weighted polynomials. Here, the striding version has an advantage since it iterates half the times of the classical due to the non expansion of the products. Additionally, since the coefficients of the result are generated consecutively, the write operations can be simplified to 4 per iteration instead of 7. In addition to this simplification of the interpolation, the non expansion of the products in the striding version allows for a halving in the memory requirements to store all the weighted polynomials, i.e., a saving of 896 bytes.

The differences in the implementation of the striding version of Karatsuba with respect to the classical Karatsuba are equivalent to Toom-Cook. There is an overhead in the instruction count due to the sequential access pattern to the coefficients of the polynomial during evaluation, and the counterpart during interpolation if one wants to exploit the single instruction multiple data capabilities of Cortex-M4 processors. However, the non expansion of the polynomial degree after the multiplication is more beneficial than in the case of Toom-Cook. In particular, if Karatsuba is applied recursively the memory utilization can be kept constant by re-utilizing the memory allocated to store the operands after the evaluation is performed.

5.2 Schoolbook multiplication

It has been shown in [KMRV18, KRS19] that schoolbook multiplication has the highest impact in the performance of polynomial multiplication. We follow the divide and conquer approach of [KRS19] to decompose the schoolbook multiplication into smaller multiplications that can be performed without fetching extra coefficients from memory and aiming to use the multiply and accumulate instructions. Moreover, to avoid length doubling, we need to perform a negacyclic convolution in-place. This is a challenge because the DSP extensions of Cortex-M4 processors offer a wide range of multiply and accumulate instructions but not their multiply and subtract counterparts. Additionally, extra load instructions are required due to the negatively wrapped accumulation which further increases the register pressure, which is key to performance [KMRV18, KRS19]. To circumvent this issue, we create a custom memory layout where both operands are in consecutive addresses and, therefore, only one pointer needs to be kept in a register while coefficients of different operands can be loaded using immediate offsets.

The other implementation decision for the polynomial multiplication is the cut-off degree from which Karatsuba is not applied anymore and the multiplication is performed using the schoolbook algorithm. Due to the availability of registers to pre-load coefficients and the absence of multiply and subtract instructions, we find that the optimal cut-off is degree 16. For higher cut-offs, the performance will be determined by the dominant term between the extra load operations required in the striding version of Karatsuba and the overhead introduced by the negacyclic convolution during schoolbook multiplication. We have verified that the optimal cut-off is degree 16, with $34,884$ clock cycles for a 256 coefficient polynomial multiplication, while for degree 32 the execution time increases to $36,340$ clock cycles. This choice was expected since 16 was also the optimal cut-off for other multiplications combining Toom-Cook and Karatsuba [KMRV18, KRS19].

In Section 7 we discuss the performance and memory figures of our proposed multiplication and compare the impact on the full Saber operation to other implementations in Cortex-M4 processors as well as in the new Cortex-M55 processor.

6 Implementation: Cortex-M55

In this section, we describe our implementations of large-degree, low-precision polynomial multiplication on the Cortex-M55 CPU, leveraging the Helium vector extension.

6.1 NTT vs. Toom-Cook/Karatsuba on vector architectures

[CHK+21] demonstrates that for Cortex-M4, polynomial multiplication via NTTs can outperform multiplication via Toom-Cook even for coefficient rings not tailored to the NTT, such as $\mathbb{Z}_{2^{13}}$. For vectorized implementations, however, the passage to the NTT comes at a larger cost than for non-vectorized implementations. We provide some details.

Setting expectations Since the Helium vector extension operates on 128-bit vectors and the Cortex-M55 CPU is single-issue for vector instructions, we consider a 4× speedup a theoretical limit for a vectorized 32-bit NTT. However, we do not expect to meet this limit, because the NTT relies on long multiplications for modular arithmetic, and few Helium instructions can perform more than two $32 \times 32 \rightarrow 64$ long multiplications. For Toom-Cook/Karatsuba based multiplication, we operate on 16-bit lanes, hence get a theoretical speedup of 8× over scalar code. However, optimized implementations on the Cortex-M4 CPU already leverage the DSP extension to treat 32-bit registers as 2×16-bit vectors and thus operate on two 16-bit values at once. We thus expect a speedup somewhere between 4× and 8×. Considering that NTT-based polynomial and matrix-vector multiplication for the Saber rings is around 2× as fast as Toom-Cook/Karatsuba based routines [CHK+21, Table 5], it is not clear which approach will be faster when implemented using Helium.

6.2 Vectorizing striding Toom-Cook/Karatsuba

Recall that 4-way Toom-Cook evaluation is a matrix-vector multiplication

$$R^{7\times 4} \times R^{4\times(n/4)} \longrightarrow R^{7\times(n/4)}, \quad R = \mathbb{Z}_{2^{16}},$$

with every sub-matrix-vector multiplication $R^{7\times 4} \times R^4 \to R^7$ corresponding to one evaluation transformation. We vectorize this by computing one sub-matrix-matrix multiplication $R^{7\times 4} \times R^{4\times \ell} \to R^{7\times \ell}$ a time, storing the four rows of the input $R^{4\times \ell}$ and the 7 rows of the output $R^{7\times \ell}$ in one vector register each. We find that by overwriting input registers as soon as they are no longer needed, the 8 vector registers available are sufficient for the transformation. With 128-bit vectors, we compute $\ell = \frac{128}{16} = 8$ evaluations at once.

The above vectorization approach works for both classical and striding Toom-Cook. However, in the case of striding Toom-Cook, we need to de-interleave the input at stride 4 first, for which we leverage the de-interleaving load instruction `VLD4{0-3}`.

The interleaving of memory operations and arithmetic is crucial to leverage instruction overlapping. We achieve this by optimizing the evaluation across loops, using preloading and late storing of input and results. The preloading becomes more challenging for striding Toom-Cook, where we can use `VLD4{0-3}` only once we have four free vector registers available. Listing 2 shows one loop of our vectorized Toom-Cook evaluation.

```
vld43.u16 {Q0,Q1,Q2,Q3}, [r0]
vadd.u16 Q7, Q0, Q2
vstrw.u32 Q6, [r14,#(-32)]
vadd.u16 Q6, Q1, Q3
vstrw.u32 Q5, [r0,#(-32)]
vsub.u16 Q5, Q7, Q6
vstrw.u32 Q4, [r0,#(-48)]
vmla.u16 Q7, Q0, r10
vstrw.u32 Q5, [r0,#(48)]
vmla.u16 Q5, Q6, r11
vstrw.u32 Q0, [r0], #64
vmla.u16 Q6, Q1, r10
vstrw.u32 Q3, [r14,#(32)]
vadd.u16 Q4, Q5, Q1
vstrw.u32 Q5, [r14], #48
vmla.u16 Q4, Q2, r10
vmla.u16 Q4, Q3, r9
```

Listing 2 Striding Toom-Cook evaluation

```
ldrd r9, r7, [r1, #48]
vmla.u16 Q3, Q6, r10
vshlc Q2, r12, #16
vmla.u16 Q2, Q4, r5
vshlc Q3, r12, #16
vmla.u16 Q3, Q4, r7
vshlc Q5, r12, #16
vmla.u16 Q2, Q7, r7
ldrd r10, r8, [r1, #8]
vmla.u16 Q3, Q6, r5
vshlc Q2, r12, #16
vmla.u16 Q2, Q4, r6
vshlc Q3, r12, #16
vmla.u16 Q3, Q4, r9
vshlc Q5, r12, #16
vmla.u16 Q2, Q7, r9
```

Listing 3 Schoolbook multiplication

We turn to the implementation of Toom-Cook interpolation: As for evaluation, we vectorize interpolation by computing 8 interpolations at once, keeping the 7 input/output rows in one vector each and leveraging the interleaving-store `VST4{0-3}` for the striding. We're using the interpolation sequence from [KMRV18].

At the end of the interpolation, we have $f_0, \dots f_6 \in \mathbb{Z}_{2^{13}}[X^4]/(X^n+1)$ such that

$$\sum\nolimits_i f_i X^i = (f_0 + X^4 f_4) + (f_1 + X^4 f_5)X + (f_2 + X^4 f_5)X^2 + (f_3 + X^4 f_6)X^3$$

is the polynomial we're looking for. Each $X^4 \cdot f_{4+k}$ amounts to a negacyclic rotation of f_{4+k}, which we compute gradually within the interpolation loop, using `VSHLC` to shift one 128-bit block after each iteration, and remembering the carry-out as the carry-in for the next iteration. This requires the use of three more general purpose registers for carries.

The interpolation sequence from [KMRV18] is very add/sub-heavy, leading to inevitable stalls on a dual-beat system when implemented as written. We experimented with replacing `VADD`, `VSUB` by `VMLA` with constants ±1, but were limited by the pressure on the GPR file. It is left for future work to optimize this further.

6.3 Schoolbook multiplication

We consider multiplication in $\mathbb{Z}_{2^{16}}[X]/(X^k+1)$. While previous vectorized implementations use batch-multiplication, our algorithm vectorizes a single schoolbook multiplication. To our knowledge, the approach is novel, and rests on the shift-with-carry instruction VSHLC illustrated in Figure 3.

Outline We outline the algorithm for $k = 16$. We can write

$$a = (a_0 + a_8X^8)X^0 + (a_1 + a_9X^8)X^1 + \ldots + (a_7 + a_{15}X^8)X^7 = \sum_i (a_i + a_{8+i}X^8)X^i,$$

hence $ab = \sum_i \left[(a_i + a_{8+i}X^8)b\right] X^i$. We will now separately discuss the following:

- How to efficiently compute one subproduct $(a_i + a_{8+i}X^8)b$ in the sum. We will express this as an 8-fold batch multiplication over $\mathbb{Z}_{2^{16}}[X]/(X^2 + 1)$.
- How to shift + accumulate in the sum. We will express this as an iteration of addition and negacyclic rotation, similar to a linear feedback shift register.

Subproduct computation We write $b = \sum_i(b_i + b_{8+i}X^8)X^i$ as before, and hence obtain $(a_i + a_{8+i}X^8)b = \sum_j(a_i + a_{8+i}X^8)(b_j + b_{8+j}X^8)X^j$. Now, note that $\mathbb{Z}_{2^{16}}[X^8]/(X^{16} + 1)$ is a subring of $\mathbb{Z}_{2^{16}}[X]/(X^{16} + 1)$ isomorphic to the "complex numbers" $\mathbb{Z}_{2^{16}}[Y]/(Y^2 + 1)$ – we therefore have to compute a batch multiplication in $\mathbb{Z}_{2^{16}}[Y]/(Y^2 + 1)$ of the fixed $a_i + a_{8+i}Y$ with the varying $b_j + b_{8+j}Y$, $j = 0, 1, \ldots, 7$.

This batch multiplication is easy to vectorize: If a_i and a_{8+i} are in GPRs $\mathtt{a}_0$, $\mathtt{a}_1$, and $b_0, \ldots, b_7$ and $b_8, \ldots, b_{15}$ are in vectors $\mathtt{b}_0$ and $\mathtt{b}_1$, we only have to calculate a scalar-vector 2×2 schoolbook multiplication $(\mathtt{a}_0+\mathtt{a}_1Y)(\mathtt{b}_0+\mathtt{b}_1Y) = (\mathtt{a}_0\mathtt{b}_0-\mathtt{a}_1\mathtt{b}_1)+Y(\mathtt{a}_0\mathtt{b}_1+\mathtt{a}_1\mathtt{b}_0)$. This uses 2 GPRs for the a, 2 vector registers for the b, and 2 vector registers for the output. Multiplication is done via VMUL.u16 / VMLA.u16. Since there is no multiply-subtract variant of VMLA.u16, the computation of $-\mathtt{a}_1\mathtt{b}_1$ requires flipping the sign of $\mathtt{a}_1$ or $\mathtt{b}_1$.

Accumulation We need to compute the sum $ab = \sum_i \left[(a_i + a_{8+i}X^8)b\right] X^i$. and discussed how to compute an individual product $(a_i + a_{8+i}X^8)b$. We now consider how to sum them.

Since $\cdot X$ is a negacyclic shift $(z_0, \ldots, z_{k-1}) \mapsto (-z_{k-1}, z_0, \ldots, z_{k-2})$, computing $[(a_i + a_{8+i}X^8)b]X^i$ from $(a_i + a_{8+i}X^8)b$ means a negacyclic shift of i positions. This does not map well to MVE because the maximum shift amount of VSHLC is 32 bits, hence two 16-bit coefficients. Instead, setting $\mathbf{a}_i := a_i + a_{8+i}X^8$, we can rewrite the sum as follows:

$$ab = \sum_i \left[\mathbf{a}_i b\right] X^i = \mathbf{a}_0 b + (\mathbf{a}_1 b + (\ldots + (\mathbf{a}_6 b + (\mathbf{a}_7 b + 0)X) \cdots X)X)$$

Here, we only shift by X, corresponding to a VSHLC Qd, Ra, #16. Moreover, we can handle the sum through multiply-accumulates in the schoolbook subroutine computing $\mathbf{a}_i b + \_$.

Implementation considerations We describe how to map the above algorithm to the Helium instruction set and the microarchitecture of the Cortex-M55 processor.

First, consider a rotation $(b_0, \ldots, b_{15}) \mapsto (-b_{15}, b_0, \ldots, b_{14})$, where b is stored in vector registers $\mathtt{b}_0 = (b_0, \ldots, b_7)$ and $\mathtt{b}_1 = (b_8, \ldots, b_{15})$: To begin, two chained invocations of VSHLC with carry GPR $\mathtt{c}$ yield $\mathtt{b}_0' = (\mathtt{c}_{\text{old}}, b_0, \ldots, b_6)$ and $\mathtt{b}_1' = (b_7, \ldots, b_{14})$, with $\mathtt{c}_{\text{new}} = b_{15}$ (in-carry and out-carry in VSHLC use the same GPR). After that, $-\mathtt{c}_{\text{new}} = -b_{15}$ has to be fed back into $\mathtt{b}_0'$. We implement this lazily by buffering $\mathtt{c}_{\text{new}}$ in a vector $\mathtt{C} = (0, \ldots, 0, \mathtt{c})$ and deferring correcting $\mathtt{b}_0'$. If $\mathtt{C}$ is initially 0, we can simultaneously store $\mathtt{c}$ in $\mathtt{C}$ and clear it via VSHLC. We only need to correct $\mathtt{b}_0'$ once after the full loop.

A standalone implementation of the anticyclic shift would not perform well on a dual-beat system like the Cortex-M55 processor, because consecutive invocations of VSHLC would stall. We will revisit this after studying the nature of each subproduct computation.

Each subproduct involves the following: Firstly, loading of a-input in two GPRs, ideally using LDRD to load a pair of GPRs at once. Secondly, 4× multiply via VMUL.u16 or

VMLA.u16. Thirdly, a sign flip in a or b. The lowest instruction count is offered by storing $-\mathtt{b}_1$ in another vector throughout the loop. Flipping a adds a 1-instruction overhead, but releases pressure on the vector file. At best, we get 1 × LDRD and 3 × VMLA per subproduct.

As for the negacyclic shift, a naïve implementation would perform poorly on a dual-beat system because consecutive invocations of VMLA would stall. However, we found that interleaving subproduct and shift leads to good overlapping and resource utilization on the Cortex-M55 processor. We show an iteration of the core loop in Listing 3.

Variants There is a variant computing the rotation abX instead of ab: We write

$$\begin{aligned} aX &= (a_0 + a_8X^8)X^1 + (a_1 + a_9X^8)X^2 + \ldots + (a_7 + a_{15}X^8)X^8 \\ &= (-a_{15} + a_7X^8) + (a_0 + a_8X^8)X^1 + \ldots + (a_6 + a_{14}X^8)X^7 \end{aligned}$$

and apply the same multiply-accumulate strategy as before. The rotation in the a-input is easy to express through modified immediate offsets in the GPR loads.

Secondly, consider what happens when we accumulate onto the destination vector in the first iteration. Since the result of first iteration gets shifted 7 times subsequently, this computes $ab + cX^7$, where c is the polynomial previously stored in the destination vectors.

The previous points allow for efficient integration of 16 → 32 Karatsuba interpolation with degree-16 schoolbook multiplication: Recall that we have to compute $(f_\mathrm{e}g_\mathrm{e} + Xf_\mathrm{o}g_\mathrm{o}) + X(f_\mathrm{s}g_\mathrm{s} - f_\mathrm{e}g_\mathrm{e} - f_\mathrm{o}g_\mathrm{o})$. We avoid any manual shift of $f_\mathrm{o}g_\mathrm{o}$ here by computing $Xf_\mathrm{o}g_\mathrm{o}$ first, and then using the multiply-accumulate variant for $f_\mathrm{s}g_\mathrm{s} - f_\mathrm{o}g_\mathrm{o} = f_\mathrm{s}g_\mathrm{s} + X^7(X^8 \cdot (Xf_\mathrm{o}g_\mathrm{o}))$: Here, $Xf_\mathrm{o}g_\mathrm{o}$ is known, and $\cdot X^8$ is a single sign-flip and some re-indexing.

6.3.1 Integrating Toom-Cook, Karatsuba, and Schoolbook

We implement a degree 256 negacyclic polynomial through a combination of one layer of 4-way Toom-Cook for degree 256 → 64 reduction, one layer of Karatsuba for degree 32 → 16 reduction, and an integrated 32 ↔ 16 followed by 16 × 16 schoolbook. We found this variant slightly more efficient than an implementation of degree-32 schoolbook.

6.4 Number Theoretic Transform

Alg. 3: Barrett reduction

```
In: Modulus $n < 2^{32}$, $|z| < 2^{31}$
Out: Barrett reduction of $z$ w.r.t. $n$.
VQRDMULH.s32 $t$, $z$, $\left\lfloor \frac{2^\ell}{2n} \right\rceil$
VMLA.s32 $z$, $t$, $-n$
return $z$
```

Alg. 4: Montgomery multiplication

```
In: $n < 2^{32}$, $|a|, |b| < 2^{31}$, $\omega \equiv n^{-1} \bmod 2^{32}$
Out: Representative $h$ of $ab/2^{32}$ modulo $n$
V[QD]MULH.s32 $h$, $a$, $b$
VMUL.u32 $l$, $a$, $b$
VMUL.u32 $l$, $l$, $\omega$
V[QD]MULH.s32 $c$, $l$, $n$
V[H]SUB.s32 $h$, $h$, $c$
```

6.4.1 Modular Arithmetic

Montgomery multiplication Algorithm 4 shows two implementations of single-width Montgomery multiplication (Algorithm 2), one using the multiply-high instruction VMULH.s32, whose functional effect is $a, b \mapsto \left\lfloor \frac{ab}{2^{32}} \right\rfloor$, the other the *doubling* multiply-high instruction VQDMULH.s32 from fixed-point arithmetic, whose functional effect is $a, b \mapsto \left\lfloor \frac{2ab}{2^{32}} \right\rfloor$. In the latter case, the doubling can be corrected through a *halving* subtraction VHSUB.s32 in the last instruction. We mention the fixed-point variant for two reasons: Firstly, because VQDMULH has a scalar-vector variant where one vector is constant. This is useful for the

VQDMULH.s32 c, l, n step, where n is a constant which we'd ideally store in a GPR rather than a vector. Secondly, the fixed-point variant can be improved to 4 and 3 instruction Montgomery multiplication variants, which we discuss now.

Alg. 5: Montgomery via rounding

```
In: Odd $n < 2^{32}$, $|a|, |b| < 2^{31}$, $b$ odd
In: $\omega \equiv -n^{-1}$ modulo $2^{32}$ precomputed
Out: Representative $h$ of $ab/2^{31}$ modulo $n$
VQRDMULH.s32 $h$, $a$, $b$
VMUL.u32 $l$, $a$, $b$
VMUL.u32 $l$, $l$, $\omega$
VQRDMLAH.s32 $h$, $l$, $n$
```

Alg. 6: Montgomery via rounding, constant

```
In: Odd $n < 2^{32}$, $|a|, |b| < 2^{31}$, $b$ odd
In: $b' \equiv -bn^{-1}$ modulo $2^{32}$ precomputed
Out: Representative $h$ of $ab/2^{31}$ modulo $n$
VQRDMULH.s32 $h$, $a$, $b$
VMUL.u32 $l$, $a$, $b'$
VQRDMLAH.s32 $h$, $l$, $n$
```

Algorithm 5 shows a variant of Algorithm 4 where the subtraction step VHSUB has been merged with the high multiply VQDMULH.s32 into a single doubling multiply-high-rounding with accumulate instruction VQRDMLAH.s32, leveraging $\frac{v+w}{2^l} = \lfloor \frac{v}{2^\ell} \rceil + \lfloor \frac{w}{2^\ell} \rceil$ for any two $v, w < 2^\ell$ s.t. $v \equiv -w \not\equiv 2^{\ell-1}$ modulo 2^ℓ, the case $2^{\ell-1}$ being excluded by the important new assumption that b is odd. This variant of Montgomery multiplication has been communicated with the authors of [BHK+], and is explored in greater detail in loc.cit. alongside other improvements to modular arithmetic.

Barrett reduction Algorithm 3 implements Barrett reduction (Algorithm 1). Two things are noteworthy: Firstly, we have merged the multiplication $c \leftarrow n \cdot t$ and the subtraction $z - c$ into a single multiply-accummulate VMLA. Secondly, since VQRDMULH.s32 computes $\lfloor \frac{2ab}{2^{32}} \rceil$, not $\lfloor \frac{ab}{2^{32}} \rceil$, we use $2 \left\lfloor \frac{2^\ell}{2n} \right\rceil$ as the approximation to $\frac{2^\ell}{n}$ instead of $\left\lfloor \frac{2^\ell}{n} \right\rceil$ in Algorithm 1. The theoretical implications of this modification are studied in greater detail in [BHK+].

Central reduction After the inverse NTT, we need to normalize to signed canonical representatives in $\{-\lfloor \frac{q}{2} \rfloor, \ldots, \lfloor \frac{q}{2} \rfloor\}$. Assuming we are already given a signed representative in the range $\{-q+1, \ldots, q-1\}$, this can be achieved by two conditional additions of $\pm q$. In Helium, we implement central reduction using lane predication. See Algorithm 7.

Alg. 7: Central reduction

```
In: $q$ 32-bit modulus, $v \in \mathbb{Z}^4$, $|v_i| < q$.
Out: $v'$ with $|v'_i| < q/2$ and $v'_i \equiv v_i$ mod $q$.
VPT.s32 LT, $v$, #0            Predicate next operation on $v_i < 0$
VADDT.s32 $v$, $v$, $q$        Predicated correction by $q$
VPT.s32 GE, $v$, #$\frac{q}{2}$    Predicate next operation on $v_i \geq \frac{q}{2}$
VSUBT.s32 $v'$, $v$, $q$       Predicated correction by $q$
```

6.4.2 Forward NTT

We compute two layers of radix-2 butterflies at once, using 4 vectors for the butterfly and leaving 4 for intermediate results and loop optimization (pre-load, late-store). Merging three layers would require register spilling, so we have not implemented it. See Algorithm 8.

We implement Montgomery multiplication via the 3-instruction rounding strategy Algorithm 6, choosing odd representatives for the ζ_i and their twists ζ'_i. For NTT layers 0 to 5, we keep the modulus and root constants in 7 GPRs, reducing the pressure on the vector file and taking advantage of the scalar-vector instructions. For the last two NTT layers, where each size-4 butterfly operates on consecutive 32-bit elements, we use the de-interleaving load `VLD4{0-3}` and compute four size-4 butterflies at once. In this case, we require 6 *vectors* for the root constants, in addition to the 4 input vectors. However, we find that with careful register management, the total of 8 vector registers is sufficient.

Alg. 8: Size-4 NTT Cooley-Tukey butterfly

```
In: $a_0, a_1, a_2, a_3, \zeta_0, \zeta_1, \zeta_2 \in \mathbb{F}_q$
Out: Radix-4 butterfly of $(a_i)$ w.r.t. $\zeta_j$.
  $a_2 \leftarrow \zeta_0 a_2$,                              $a_3 \leftarrow \zeta_0 a_3$                              Montgomery
  $(a_0, a_2) \leftarrow (a_0 + a_2, a_0 - a_2)$,    $(a_1, a_3) \leftarrow (a_1 + a_3, a_1 - a_3)$    Add/Sub
  $a_1 \leftarrow \zeta_1 a_1$,                              $a_3 \leftarrow \zeta_2 a_3$                              Montgomery
  $(a_0, a_1) \leftarrow (a_0 + a_1, a_0 - a_1)$,    $(a_2, a_3) \leftarrow (a_2 + a_3, a_2 - a_3)$    Add/Sub
```

To leverage instruction overlapping, it is vital to interleave the Montgomery multiplications – consisting of multiplications only – and the add/sub steps – consisting of addition operations only. Moreover, since there are 6 multiplications but only 4 addition/subtraction operations, we also interleave loads/stores in order to achieve a stall-free execution.

6.4.3 Point multiplication for full NTT

For the point multiplication in $\mathbb{F}_q^n$, our measured code uses Algorithm 4. We comment on the possibility of using the shorter Algorithm 5: This algorithm is only applicable if one of the inputs is odd. Here, the following trick can be applied:

Proposition 1. *Assume a Montgomery multiplication routine for the NTT which produces only even representatives. Then, if (x_s) is an input vector to the NTT, all entries of its NTT transform have the same parity, which agrees with the parity of the representative x_0.*

In particular, if x_0 is odd, then all *entries in the NTT transform of (x_s) are odd.*

Proof. Consider the first layer: If n is the size of the NTT, each pair $(a = x_r, b = x_{r+n/2})$ of elements in the lower and upper half is transformed via $(a, b) \mapsto (a + \zeta b, a - \zeta b)$. By assumption, ζb is represented by an even integer, so at the end of layer 0, the parity of the elements in the lower half has not changed, and the parity of elements in the upper half is the same as the parity of the corresponding lower half element. Continue inductively. □

As long we ensure that x_0 is odd initially, we can therefore force all outputs of the NTT to have only odd entries, and thus be suitable for our accelerated point multiplication via Algorithm 5. We leave it for future work to explore the use of this trick further.

6.4.4 Point multiplication for partial NTT

When implementing multiplication in $\mathbb{F}_q[X]/(X^{256} + 1)$ using a 6-layer incomplete NTT, the base multiplication is in $\mathbb{F}_q[X]/(X^4 - \zeta)$. For this, [CHK+21] relies on `SMLAL` to reduce the number of Montgomery reductions by operating in double-width values as long as possible, as does [BHK+], which moreover batches the base multiplication.

As mentioned in Section 3.2, batched implementations are difficult to realize in the Helium instruction set due to the lower number of vectors. A batched multiplication in $\mathbb{F}_q[X]/(X^4 - \zeta)$ would need at least $4 + 4 + 4$ vectors for the two inputs and the output. Instead, we vectorize a single multiplication in $\mathbb{F}_q[X]/(X^4 - \zeta)$ as follows: For inputs $a = a_0 + a_1 X + a_2 X^2 + a_3 X^3$ and $b = b_0 + b_1 X + b_2 X^2 + b_3 X^3$, we first prepare the reversed and expanded array of 32-bit values $b_3, b_2, b_1, b_0, \zeta b_3, \zeta b_2, \zeta b_1$ in memory. We can then compute the coefficients of ab as dot products of $[a_0, a_1, a_2, a_3]$ with length-4 subvectors of the expanded array, each of which can be computed with a single invocation of the long-multiply-accumulate-across instruction `VMLALDAV.s32`. For example, the X^2-coefficient of

ab is $[a_0, a_1, a_2, a_3] \cdot [b_2, b_1, \zeta b_3, \zeta b_2]$. Like `UMLAL`, `VMLALDAV.s32` accumulates into a pair of GPRs. We compute all coefficients of ab in pairs of GPRs each, before moving them into a vector using `VMOV Qx[i], Qx[j], Ra, Rb` and applying a single Montgomery reduction.

The approach has some drawbacks: Firstly, we need four loads for b; a batched implementation would require only one. This can be amortized by simultaneously computing multiple $a_i b$; we have not yet implemented this approach, yet. Secondly, the cost of preparing $b_3, b_2, b_1, b_0, \zeta b_3, \zeta b_2, \zeta b_1$. Following an approach similar to [BHK$^+$] and adapted to this context, this can be partly amortized in the application to matrix-vector multiplication by precomputing $b_3, b_2, b_1, b_0, \zeta b_3, \zeta b_2, \zeta b_1$ as part of the forward NTT. In Section 7 below, this size-doubling version of the NTT is called "expanded NTT". We find that the above schoolbook multiplication strategy offers a good mix of different kinds of operations which can be interleaved to leverage instruction overlapping.

6.4.5 Inverse NTT

We use Gentleman-Sande butterflies $(a, b) \mapsto (a + b, \zeta(a - b))$, which we compute using interleaved add/sub sequences and the 3-instruction Montgomery multiplication as in the forward-NTT. Overflow is prevented through the addition of selected Barrett reductions.

The GS butterflies are inverse to the CT butterflies only up to a factor of 2, which we need to compensate through a modular division in/after the inverse NTT. We also need to account for the Montgomery twist by $2^{-31} \in \mathbb{F}_q$ in the point multiplication. We merge half of the required scalings with the multiplications in the last-layer GS butterflies. For the other half, we add explicit Montgomery multiplications. One could integrate the scalings into the GS butterflies for all but the first coefficient with a trick similar to Proposition 1; however, we would need explicit Barrett reductions at the end of the last layer in this case, which would only be one instruction shorter than the Montgomery multiplications.

6.5 Side-Channel resistance

All of our code is resistant against timing side-channels on Cortex-M4 and Cortex-M55: The control-flow is secret-independent, and all instructions we used have data-independent timing, including predication. We do not attempt resistance against other side-channels.

6.6 Hashing

We have not attempted to develop hashing implementations based on the Helium vector extension: Both the smaller scale nature of Cortex-M CPUs and the fact that software executes directly in the physical address space (without virtual to physical translation) mean that the overhead associated with offloading computation to accelerators is very low. As a result, many Cortex-M microcontrollers already include accelerators for symmetric cryptography and hashing. It is expected that these accelerators will be used for the hashing in PQC schemes, eliminating the need for high performance software-based hashing.

"90's versions" Numerous PQC schemes offer a "90's version" replacing the use of SHA-3 by AES and SHA-2, and it has been discussed whether NIST should standardize those versions. We support NIST standarizing "90's versions": Firstly, many existing MCUs already have hardware acceleration for AES and SHA-2. Secondly, AES and SHA-2 will not go away anytime soon, and limited gate budget may prevent vendors from shipping MCUs with both SHA-2 and SHA-3 acceleration. Finally, SHA-2 is a faster software-fallback for systems which do not have have any hardware acceleration for hashing.

Table 2 Comparison of Toom-Cook and NTT-based implementations of Ind-CPA and Ind-CCA versions of the NIST PQC finalist Saber. SW-based hashing waters down optimizations for polynomial multiplication, but is expected to be HW-accelerated in practice, see Section 6.6. Numbers are for the SHA3-based Saber, not the 90s version.

		Operation	Cortex-M4			Cortex-M55	
			NTT [CHK+21]	TC†	TC [This]	TC [This]	NTT [This]
LightSaber	CPA	KeyGen	294k	377k	367k	244k	232k
		Encaps	330k	440k	425k	250k	233k
		Decaps	58k	101k	96k	22k	20k
	CCA	KeyGen	360k	443k	433k	308k	296k
		Encaps	513k	622k	607k	429k	412k
		Decaps	498k	631k	612k	378k	358k
Saber	CPA	KeyGen	554k	759k	737k	466k	436k
		Encaps	606k	858k	829k	480k	442k
		Decaps	79k	142k	135k	30k	27k
	CCA	KeyGen	658k	862k	840k	568k	538k
		Encaps	864k	1115k	1086k	735k	697k
		Decaps	835k	1128k	1091k	655k	615k
Firesaber	CPA	KeyGen	879k	1260k	1222k	750k	693k
		Encaps	947k	1395k	1346k	772k	706k
		Decaps	101k	184k	174k	40k	36k
	CCA	KeyGen	1008k	1389k	1350k	877k	820k
		Encaps	1255k	1703k	1654k	1077k	1011k
		Decaps	1227k	1732k	1674k	985k	914k

† Saber implementation from [KRSS] using Toom-Cook for polynomial multiplication instead of NTT.

7 Results

7.1 Development setup

Benchmarking for the Cortex-M4 processor We have developed our code based on the pqm4 framework [KRSS]. We have focused our development on polynomial multiplication only without optimizing other operations in Saber. To measure the performance, we have executed the benchmark environment provided by pqm4 on a STM32F4 Discovery board featuring a 32-bit Arm Cortex-M4 with FPU core, 1MB of Flash, and 192KB of RAM.

Benchmarking for the Cortex-M55 processor We developed our code using the Fixed Virtual Platform (FVP) for the Arm® Corstone®-300 MPS2[1], containing a functional model of the Cortex-M55. To measure performance, we have used an FPGA model for the Cortex-M55r1 CPU which is expected to be publicly released by the end of 2021. An image including the Cortex-M55r0 CPU is already available[2]. Researchers can get access to Cortex-M55 RTL and cycle-accurate models through the Arm Academic Access program[3].

Code generation Most of our Helium and Cortex-M4 assembly was developed with the help of a small Python-based code-generation framework, similar to [KRS19]. The main tasks the framework performs are register management and address offset calculations when

[1] Freely available at https://developer.arm.com/tools-and-software/open-source-software/arm-platforms-software/arm-ecosystem-fvps

[2] https://developer.arm.com/tools-and-software/development-boards/fpga-prototyping-boards contains more information on prototyping boards and FPGA images.

[3] https://www.arm.com/academicaccess

Table 3 Comparing different stages of NTT multiplication for Saber. "Expand" is the NTT-variant discussed in Section 6.4.4.

	NTT-based polynomial multiplication								
	NTT			inv-NTT		Base multiplication		Poly-Mul	
	Full	Partial	Expand	Full	Partial	Point	deg-4	Full	Partial
Cortex-M4 [CHK+21]†	–	5794	–	–	7791	–	3886	–	–
Cortex-M55 [This work]	2027	1473	1835	2559	1940	569	1151	7194	6411

† [CHK+21] provides batched implementations; numbers reported here are averaged.

Table 4 Performance of polynomial multiplication based on Toom-Cook/Karatsuba. Sum-decompositions refer to shares of Toom-Cook/Karatsuba during interpolation/evaluation.

	Toom-Cook/Karatsuba based polynomial multiplication			
	Evaluation 256 → 32	Interpolation 32 → 256	Schoolbook (32x32)	Poly-Mul
Cortex-M55 [This work]	289+330 = 619	370+531 = 901	267	7790
	Evaluation 256 → 16	Interpolation 16 → 256	Schoolbook (16x16)	Poly-Mul
Cortex-M4 [This work]	2188+7x(3x128+260) = 6696	7x(3x91+179)+3187 = 6351	302	34884

loading contiguous or scattered buffers. While the final assembly can possibly be written by hand, we found the tool useful for quick experimentation with different approaches. For very simple code-sequences, such as the Karatsuba algorithm or NTT base multiplication, we have written the assembly by hand.

There are alternatives to handwritten assembly, e.g. auto-vectorization and intrinsics. The benefit is that the programmer operates at the level of C, while giving up assembly-level control potentially unlocking the final bit of performance. We explore handwritten assembly in this work to understand the capabilities of the Cortex-M55 processor and the Helium instruction set, without wondering how much penalty we pay from using C.

7.2 Polynomial and matrix-vector multiplication

Table 5 and Table 4 show the performance of our matrix-vector multiplication routines, running on the Cortex-M55 CPU, compared to prior implementations on the Cortex-M4 CPU. We found that the incomplete NTT with expansion (see Section 6.4.4) outperforms a full NTT, and is about 3.4× faster than the incomplete NTT implementation from [CHK+21]. For Toom-Cook, we obtain a $\approx 5\times$ speedup compared to [MKV20]. The results are in line with the expectations established in Section 6.1. Overall, we confirm the finding of [CHK+21] that NTT-based polynomial multiplication outperforms Toom-Cook-based multiplication, but we note that the difference is smaller than on the Cortex-M4 CPU.

Table 3 and Table 4 provide a more detailed breakdown of the performance of various components of NTT and Toom-Cook/Karatsuba based polynomial multiplication.

7.3 Saber

We have integrated our polynomial and matrix-vector multiplication routines into a top-level implementation of the NIST PQC finalist Saber. Table 2 shows the results for both Toom-Cook/Karatsuba and NTT-based implementations in comparison with prior art.

As observed and explained above, the difference between Toom-Cook/Karatsuba and NTT on the Cortex-M55 processor is smaller than on the Cortex-M4 processor. We also

Table 5 Comparison of different approaches for Matrix-Vector multiplication.

	Matrix-Vector Multiplication					
	NTT			Toom-Cook + Karatsuba		
Dim	Cortex-M4 [CHK+21]	Cortex-M55r1 (this work)	Speedup	Cortex-M4 [MKV20]	Cortex-M55r1 (this work)	Speedup
l=2	66k	18.4k	3.5x	159k	28.1k	5.6x
l=3	125k	35.9k	3.4x	317k	61.6k	5.1x
l=4	205k	58.9k	3.4x	528k	107.7k	4.9x

Table 6 Comparison of the memory utilization of Saber on Cortex-M4 processor when implementing different polynomial multiplication algorithms.

	NTT [CHK+21]‡	TC classical†	TC striding [This]
	KeyGen/Encaps/Decaps	KeyGen/Encaps/Decaps	KeyGen/Encaps/Decaps
LightSaber	5,3 / 5,3 / 5,3 KB	6,1 / 6,0 / 6,0 KB	4,3 / 4,3 / 4,3 KB
Saber	6,4 / 6,3 / 6,3 KB	6,6 / 6,5 / 6,6 KB	4,8 / 4,8 / 4,8 KB
FireSaber	7,4 / 7,3 / 7,4 KB	7,1 / 7,1 / 7,1 KB	5,3 / 5,3 / 5,3 KB

† Saber implementation from [KRSS] using Toom-Cook for polynomial multiplication instead of NTT.

‡ Stack-optimized implementation of [CHK+21] from PQM4 [KRSS], commit https://github.com/mupq/pqm4/commit/992f0f226503d43b6d33278ecb60a9168ed8d787.

see that implementations based on the striding technique are slightly faster than classical Toom-Cook/Karatsuba. Additionally, the main benefit of the striding approach is the reduced memory usage. We show this in Table 6, where the memory footprint of different Saber implementations on Cortex-M4 is compared. The striding Toom-Cook approach provides a memory compact alternative to the NTT based implementation.

We repeat the common observation that arithmetic underlying Saber, and many other PQC schemes, is outweighed by the computational complexity of hashing; see Section 6.6.

8 Conclusion

In this work, we have introduced the Helium vector extension for the M-profile of the Arm architecture and studied its use for structured lattice-based cryptography in the example of the Cortex-M55 processor and the NIST PQC finalist Saber. We have demonstrated that even within the tight constraints of the embedded market, features like instruction overlapping, scalar-vector operations and careful instruction design allow a speedup close to the theoretical optimum for the core mathematical primitives – despite only 8 vector registers and an in-order, single-issue pipeline of the Cortex-M55 processor. We have also introduced and implemented a memory-efficient variant of the Toom-Cook multiplication suitable for highly constrained devices. Though we have performed numerous optimizations, there is still scope for further improvements and extension: We hope that our results will motivate further research into the capabilities of the Helium vector extension, esp. within the realm of PQC, and to explore further schemes and optimizations.

9 Acknowledgements

This work was supported partially by the Research Council KU Leuven: C16/15/058 and by CyberSecurity Research Flanders with reference number VR20192203. In addition, Angshuman Karmakar is funded by FWO (Research Foundation – Flanders) as junior post-doctoral fellow (contract number 203056 / 1241722N LV).

We thank Jack Andrew, Francois Botman, Tom Grocutt, Fabien Klein and Jun Mendoza for sharing their insights into the Helium vector extension and the Cortex-M55 processor.

References

[AA16] Jacob Alperin-Sheriff and Daniel Apon. Dimension-preserving reductions from LWE to LWR. *IACR Cryptol. ePrint Arch.*, 2016:589, 2016.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A New Hope. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016.

[Arma] Arm Limited. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/latest.

[Armb] Arm Limited. Armv8-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0553/latest.

[Armc] Arm Limited. Blog series: “making Helium: Why not just add Neon?”. Part 1: https://community.arm.com/developer/research/b/articles/posts/making-helium-why-not-just-add-neon, Part 2: https://community.arm.com/developer/research/b/articles/posts/making-helium-sudoku-registers-and-rabbits, Part 3: https://community.arm.com/developer/research/b/articles/posts/making-helium-going-around-in-circles, Part 4: https://community.arm.com/developer/research/b/articles/posts/making-helium-going-around-in-circles.

[Armd] Arm Limited. Whitepaper: Introducing Cortex-M55. https://www.arm.com/resources/white-paper/cortex-m55-introduction.

[Arme] Arm Limited. Whitepaper: Introduction to the Armv8.1-M Architecture. https://www.arm.com/resources/white-paper/intro-armv8-1-m-architecture.

[Arm20] Arm Research. Whitepaper: Post-Quantum Cryptography, 2020. https://community.arm.com/developer/research/m/resources/1002.

[BDK+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367. IEEE, 2018.

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians, 2001. http://cr.yp.to/papers/m3.pdf.

[BGM+16] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, volume 9562 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2016.

[BGML+18] Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: KEM and PKE based on GLWR. Cryptology ePrint Archive, Report 2018/725, 2018. https://eprint.iacr.org/2018/725.

[BHK$^+$] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. https://eprint.iacr.org/2021/986.

[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). https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf, 2020. [Online; accessed 3-July-2021].

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *EUROCRYPT 2012*, pages 719–737, 2012.

[CHK$^+$21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, Feb. 2021.

[CKLS18] Jung Hee Cheon, Duhyeong Kim, Joohee Lee, and Yongsoo Song. Lizard: Cut off the tail! A practical post-quantum public-key encryption from LWE and LWR. In Dario Catalano and Roberto De Prisco, editors, *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*, volume 11035 of *Lecture Notes in Computer Science*, pages 160–177. Springer, 2018.

[Coo66] S. A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966. pp. 51-77.

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*, volume 10831 of *Lecture Notes in Computer Science*, pages 282–305. Springer, 2018.

[DXL12] Jintai Ding, Xiang Xie, and Xiaodong Lin. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive, Report 2012/688, 2012. https://eprint.iacr.org/2012/688.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *J. Cryptol.*, 26(1):80–101, 2013.

[JZC$^+$18] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. IND-CCA-secure key encapsulation mechanism in the quantum random oracle model, revisited. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 96–125. Springer, 2018.

[KMRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM: CCA-secure module lattice-based key encapsulation on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):243–266, Aug. 2018.

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of USSR Academy of Sciences*, 145(7):293–294, 1962.

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in \mathbb z_2ˆm[x] on cortex-m4 to speed up NIST PQC candidates. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, volume 11464 of *Lecture Notes in Computer Science*, pages 281–301. Springer, 2019.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[Mar21] John Marsh. *Arm Helium Technology M-Profile Vector Extension (MVE) for Arm Cortex-M Processors Reference Book*. Arm Education Media, 2021. Freely available at https://www.arm.com/resources/ebook/helium-mve-reference-book.

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020, Issue 2:222–244, 2020.

[NISa] NIST, National Institute for Standards in Technology. Post-Quantum Cryptography Project. https://csrc.nist.gov/projects/post-quantum-cryptography (last accessed May 2021).

[NISb] NIST, National Institute for Standards in Technology. Post-Quantum Cryptography Project, round 3 candidates. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions (last accessed May 2021).

[Pei14] Chris Peikert. Lattice cryptography for the internet. In Michele Mosca, editor, *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, volume 8772 of *Lecture Notes in Computer Science*, pages 197–219. Springer, 2014.

[PZ03] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *eprint arXiv:quant-ph/0301141*, January 2003.

[Reg04] Oded Regev. *New Lattice-based Cryptographic Constructions*, volume 51-6, pages 899–942. ACM, New York, NY, USA, November 2004.

[Sho97] Peter W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Sci. Statist. Comput.*, 26:1484, 1997.

[Too63] A.L Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics-Doklady*, volume 7, pages 714–716, 1963. http://toomandre.com/my-articles/engmat/MULT-E.PDF.

Appendix A Classical vs. striding Toom-Cook 4-way

This section compares the algorithms for the evaluation and interpolation techniques of classical Toom-Cook 4-way used in earlier implementations of Saber [DKRV18, KMRV18, MKV20, KRS19] with the corresponding algorithms for striding Toom-Cook 4-way multiplication proposed in this paper. The changes in the algorithms have been highlighted with bold text. Additionally, there is a change on the output of the evaluation and the input of the interpolation with respect to the previous methods.

Alg. 9: Toom-Cook 4-way evaluation with vertical scanning from [KMRV18]

```
Input: Two arrays $A$ and $B$ with the $n = 256$ coefficients of the polynomials
Output: Seven arrays $w_1$ to $w_7$ with 127 coefficients of the intermediate products each
for $j = 0$ to 63 do
    $\mathbf{r_0} = \mathbf{A_0}[\mathbf{j}]$;
    $\mathbf{r_1} = \mathbf{A_{64}}[\mathbf{j}]$;
    $\mathbf{r_2} = \mathbf{A_{128}}[\mathbf{j}]$;
    $\mathbf{r_3} = \mathbf{A_{192}}[\mathbf{j}]$;
    $r_4 = r_0 + r_2$;
    $r_5 = r_1 + r_3$;
    $r_6 = r_4 + r_5$;
    $r_7 = r_4 - r_5$;
    $aws_3[j] = r_6$;
    $aws_4[j] = r_7$;
    $r_4 = 2 * (r_0 * 4 + r_2)$;
    $r_5 = r_1 * 4 + r_3$;
    $r_6 = r_4 + r_5$;
    $r_7 = r_4 - r_5$;
    $aws_5[j] = r_6$;
    $aws_6[j] = r_7$;
    $r_4 = 8 * r_3 + 4 * r_2 + 2 * r_1 + r_0$;
    $aws_2[j] = r_4$;
    $aws_7[j] = r_0$;
    $aws_1[j] = r_3$;
Repeat the above steps to generate the weighted polynomials $bws_1$ to $bws_7$;
for $i = 1$ to 7 do
    $w_i = aws_i * bws_i$;
return $w_1$ to $w_7$;
```

Alg. 10: Striding Toom-Cook 4-way evaluation with vertical scanning of the coefficients[**This work**]

```
Input: Two arrays $A$ and $B$ with the $n = 256$ coefficients of the polynomials
Output: Seven arrays $w_1$ to $w_7$ with 64 coefficients of the intermediate products each
for $j = 0$ to 63 do
    $\mathbf{r_0} = \mathbf{A_0}[\mathbf{j}]$;
    $\mathbf{r_1} = \mathbf{A_1}[\mathbf{j}]$;
    $\mathbf{r_2} = \mathbf{A_2}[\mathbf{j}]$;
    $\mathbf{r_3} = \mathbf{A_3}[\mathbf{j}]$;
    $r_4 = r_0 + r_2$;
    $r_5 = r_1 + r_3$;
    $r_6 = r_4 + r_5$;
    $r_7 = r_4 - r_5$;
    $aws_3[j] = r_6$;
    $aws_4[j] = r_7$;
    $r_4 = 2 * (r_0 * 4 + r_2)$;
    $r_5 = r_1 * 4 + r_3$;
    $r_6 = r_4 + r_5$;
    $r_7 = r_4 - r_5$;
    $aws_5[j] = r_6$;
    $aws_6[j] = r_7$;
    $r_4 = 8 * r_3 + 4 * r_2 + 2 * r_1 + r_0$;
    $aws_2[j] = r_4$;
    $aws_7[j] = r_0$;
    $aws_1[j] = r_3$;
Repeat the above steps to generate the weighted polynomials $bws_1$ to $bws_7$;
for $i = 1$ to 7 do
    $w_i = aws_i * bws_i$;
return $w_1$ to $w_7$;
```

Alg. 11: Toom-Cook 4-way interpolation [KMRV18]

```
Input: Seven arrays $w_1$ to $w_7$ with 127 coefficients of the intermediate products each
Output: An array $C$ containing the coefficients of $C(x) = A(x) * B(x)$
$\mathbf{C \leftarrow 0}$;
for $j = 0$ to 126 do
    $r_1 = w_2[i]$;
    $r_4 = w_5[i]$;
    $r_5 = w_6[i]$;
    $r_0 = w_1[i]$;
    $r_2 = w_3[i]$;
    $r_3 = w_4[i]$;
    $r_6 = w_7[i]$;
    $r_1 = r_1 + r_4$;
    $r_5 = r_5 - r_4$;
    $r_3 = (r_3 - r_2)/2$;
    $r_4 = r_4 - r_0$;
    $r_8 = 64 \cdot r_6$;
    $r_4 = r_4 - r_8$;
    $r_4 = 2 \cdot r_4 + r_5$;
    $r_2 = r_2 + r_3$;
    $r_1 = r_1 - 65 \cdot r_2$;
    $r_2 = r_2 - r_6$;
    $r_2 = r_2 - r_0$;
    $r_1 = r_1 + 45 \cdot r_2$;
    $r_4 = (r_4 - 8 \cdot r_2)/_{24}$;
    $r_5 = r_5 + r_1$;
    $r_1 = (r_1 + 16 \cdot r_3)/_{18}$;
    $r_3 = -(r_3 + r_1)$;
    $r_5 = (30 \cdot r_1 - r_5)/_{60}$;
    $r_2 = r_2 - r_4$;
    $r_1 = r_1 - r_5$;
    $\mathbf{C[i] = (C[i] + r_6)}$;
    $\mathbf{C[i + 64] = (C[i + 64] + r_5)}$;
    $\mathbf{C[i + 128] = (C[i + 128] + r_4)}$;
    $\mathbf{C[i + 192] = (C[i + 192] + r_3)}$;
    $\mathbf{C[i + 256] = (C[i + 256] + r_2)}$;
    $\mathbf{C[i + 320] = (C[i + 320] + r_1)}$;
    $\mathbf{C[i + 384] = (C[i + 384] + r_0)}$;
return $C$;
```

Alg. 12: Striding Toom-Cook 4-way interpolation[**This work**]

```
Input: Seven arrays $w_1$ to $w_7$ with 64 coefficients of the intermediate products each
Output: An array $C$ containing the coefficients of $C(x) = A(x) * B(x)$
for $\mathbf{j = 0}$ to $\mathbf{63}$ do
    $\mathbf{r_7 = r_0}$;   $\mathbf{r_8 = r_1}$;   $\mathbf{r_9 = r_2}$;
    $r_1 = w_2[i]$;
    $r_4 = w_5[i]$;
    $r_5 = w_6[i]$;
    $r_0 = w_1[i]$;
    $r_2 = w_3[i]$;
    $r_3 = w_4[i]$;
    $r_6 = w_7[i]$;
    $r_1 = r_1 + r_4$;
    $r_5 = r_5 - r_4$;
    $r_3 = (r_3 - r_2)/2$;
    $r_4 = r_4 - r_0$;
    $r_8 = 64 \cdot r_6$;
    $r_4 = r_4 - r_8$;
    $r_4 = 2 \cdot r_4 + r_5$;
    $r_2 = r_2 + r_3$;
    $r_1 = r_1 - 65 \cdot r_2$;
    $r_2 = r_2 - r_6$;
    $r_2 = r_2 - r_0$;
    $r_1 = r_1 + 45 \cdot r_2$;
    $r_4 = (r_4 - 8 \cdot r_2)/24$;
    $r_5 = r_5 + r_1$;
    $r_1 = (r_1 + 16 \cdot r_3)/_{18}$;
    $r_3 = -(r_3 + r_1)$;
    $r_5 = (30 \cdot r_1 - r_5)/_{60}$;
    $r_2 = r_2 - r_4$;
    $r_1 = r_1 - r_5$;
    $\mathbf{C[4 \cdot i + 3] = r_3}$;
    if $i == 0$ then
        $\mathbf{C[4 \cdot i] = r_6}$;
        $\mathbf{C[4 \cdot i + 1] = r_5}$;
        $\mathbf{C[4 \cdot i + 2] = r_4}$;
    else
        $\mathbf{C[4 \cdot i] = (r_6 + r_9)}$;
        $\mathbf{C[4 \cdot i + 1] = (r_5 + r_8)}$;
        $\mathbf{C[4 \cdot i + 2] = (r_4 + r_7)}$;
$\mathbf{C[0] = (C[0] - r_2)}$;
$\mathbf{C[1] = (C[1] - r_1)}$;
$\mathbf{C[2] = (C[2] - r_0)}$;
return $C$;
```

Appendix B Saber – A post-quantum KEM

Saber [DKRV18, BMD+20] is a key encapsulation mechanism (KEM) secure against a post-quantum chosen-ciphertext attack (CCA). It is one of the 4 finalists [NISb] in the NIST's PQC standardization procedure. We describe Saber briefly here. We refer to the NIST submission of Saber [BMD+20] for more details.

The hardness of Saber is based on the module learning with rounding (LWR) [BPR12] problem, a variant of the learning with errors (LWE) problem introduced in [Reg04]. An LWE sample is of the form $(\mathbf{A}, \mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m$ whereas an LWR sample has the form $(\mathbf{A}, \mathbf{b} = \lfloor \frac{p}{q} \mathbf{A} \cdot \mathbf{s} \rceil = \lfloor \mathbf{A} \cdot \mathbf{s} \rceil_p) \in \mathbb{Z}_q^{m \times n} \times \mathbb{Z}_p^m$. In the latter case, the explicit error e is replaced by the error inherent in rounding from $\mathbb{Z}_q$ to $\mathbb{Z}_p$. The connection between LWR and LWE is well-studied, see e.g. LWE [BPR12, AA16, BGM+16]. Schemes using LWR need to use multiple moduli [BGML+18, CKLS18] unlike LWE based schemes [ADPS16, BDK+18]. Saber uses 2-power moduli (p, q, t) such that $p|q|t$, where p and q is fixed at 2^{13} and 2^{10}, respectively, and t is $2^3, 2^4$ or 2^6 depending on the security parameter.

Saber uses module-lattices, which means that the public key $\mathbf{A} \in (\mathcal{R}_{2^k}^n)^{l \times l}$ is a *matrix of polynomials*, the secret $\mathbf{s} \in (\mathcal{R}_{2^k}^n)^l$ is a *vector of polynomials*. Here and below, $\mathcal{R}_{2^k}^n$ abbreviates $\mathbb{Z}_{2^k}[X]/(X^n + 1)$, and l is a security parameter which for Saber can be either of 2 (LightSaber), 3 (Saber), or 4 (FireSaber). This explains the importance of polynomial matrix-vector multiplication for implementations of Saber.

The construction of Saber follows the general structure of a LWE based *noisy Diffie-Hellman* key-exchange construction as shown by Chris Peikert [Pei14]. To reduce the failure probability of such schemes a reconciliation mechanism [DXL12] is used. This basic key-exchange scheme is then transformed into a chosen-plaintext attack (CPA) secure public-key encryption (PKE) [BMD+20] in the similar way ElGamal encryption is obtained from Diffie-Hellman key-exchange. Finally, this CPA secure PKE is transformed to a CCA secure KEM by Fujisaki-Okamoto (FO) transform [JZC+18, FO99, FO13].

This section lists the PKE (Algorithm 13-15) and KEM algorithms of Saber. As described in Section 2, the Saber PKE is a CPA secure scheme which is then converted into a CCA-secure KEM (Algorithm 16-18) scheme using a variant of FO transform [JZC+18, FO99, FO13].

Here, XOF stands for extended output function, β_μ denotes centered binomial distribution with standard deviation $\sqrt{\mu/2}$ ($\mu = 5, 4, 3$ for $l = 2, 3, 4$ respectively), $\mathcal{M}$ denotes the message space $\{0, 1\}^{256}$, and $\epsilon_p, \epsilon_q, \epsilon_t$ denotes $\log_2(p), \log_2(q), \log_2(t)$ respectively.

Alg. 13: Saber.PKE.KeyGen

```
input  : Security level $l$
output : pk=($seed_A$, $b$),sk=($s$)
$\text{seed}_A \leftarrow_\$ \{0,1\}^{256}$;
$r \leftarrow_\$ \{0,1\}^{256}$;
$A \leftarrow_\$ \text{Gen}(\text{XOF}(\text{seed}_A)) \in (\mathcal{R}_q^n)^{l \times l}$;
$s \leftarrow_\$ \beta_\mu(\text{XOF}(r)) \in (\mathcal{R}_q^n)^{l \times l}$;
$b = (As + h) \gg (\epsilon_q - \epsilon_p) \in (\mathcal{R}_p^n)^{l \times l}$;
return $\texttt{pk} = (\texttt{seed}_\texttt{A},\ \texttt{b}), \texttt{sk} = (\texttt{s})$;
```

Alg. 14: Saber.PKE.Dec

```
input  : sk=($s$), $ct = (c, b')$
output : $m \in \mathcal{M}$
$v = b'^T(s\ mod\ p) \in \mathcal{R}_p^n$;
$m = ((v - 2^{\epsilon_p - \epsilon_t - 1}c + h_2)) \gg (\epsilon_p - 1) \in \mathcal{R}_2^n$;
return $\texttt{m}$;
```

Alg. 15: Saber.PKE.Enc

```
input  : pk=($\text{seed}_A$, $b$), $m \in \mathcal{M}$, optional $r'$
output : $ct = (c, b')$
$A \leftarrow_\$ \text{Gen}(\text{XOF}(\text{seed}_A)) \in (\mathcal{R}_q^n)^{l \times l}$;
if $r'$ not specified then
    $r' \leftarrow_\$ \{0,1\}^{256}$;
$s' \leftarrow_\$ \beta_\mu(\text{XOF}(r')) \in (\mathcal{R}_q^n)^{l \times 1}$;
$b' = (A^T s' + h) \gg (\epsilon_q - \epsilon_p) \in (\mathcal{R}_p^n)^{l \times 1}$;
$v' = b^T(s'\ mod\ p) \in \mathcal{R}_p^n$;
$c = (v' + h_1 - 2^{\epsilon_p - 1}m) \gg (\epsilon_p - \epsilon_t - 1) \in \mathcal{R}_{2t}^n$;

return $\texttt{ct} = (\texttt{c}, \texttt{b}')$;
```

Alg. 16: Saber.KEM.KeyGen

```
input  : Security level $l$
output : $pk = (seed_A, b), sk = (z, pkh, pk, s)$
$((seed_A, b), s) =$ Saber.PKE.KeyGen(l);
$pk = (seed_A, b)$;
$pkh = \mathcal{F}(pk)$;
$z = \mathcal{U}(\{0,1\}^{256})$;
return $pk = (seed_A, b), sk = (z, pkh, pk, s)$;
```

Alg. 17: Saber.KEM.Encaps

```
input  : $pk = (seed_A, b)$
output : shared secret:= $K$
$m = \mathcal{U}(\{0,1\}^{256})$;
$((\hat{K}), r) = \mathcal{G}(\mathcal{F}(pk), m)$;
$c =$ Saber.PKE.Enc(pk, m, r);
$K = \mathcal{H}(\hat{K}, c)$;
return $ct = c$,$secret$= $K$;
```

Alg. 18: Saber.KEM.Decaps

```
input  : $sk = (z, pkh, pk, s)$, $ct = c$
output : shared secret= $K$
$m' =$ Saber.PKE.Dec($s, c$);
$((\hat{K}'), r') = \mathcal{G}(pkh, m')$;
$c' =$ Saber.PKE.Enc(pk, m', r');
if $c == c'$ then
    return $K = \mathcal{H}(\hat{K}', c)$ ;
else
    return $K = \mathcal{H}(z, c)$ ;
```