Post-Quantum Cryptography with Contemporary Co-Processors
Beyond Kronecker, Schönhage-Strassen & Nussbaumer

Joppe W. Bos
NXP Semiconductors

Joost Renes
NXP Semiconductors

Christine van Vredendaal
NXP Semiconductors

Abstract

There are currently over 30 billion IoT (Internet of Things) devices installed worldwide. To secure these devices from various threats one often relies on public-key cryptographic primitives whose operations can be costly to compute on resource-constrained IoT devices. To support such operations these devices often include a dedicated co-processor for cryptographic procedures, typically in the form of a big integer arithmetic unit. Such existing arithmetic co-processors do not offer the functionality that is expected by upcoming post-quantum cryptographic primitives. Regardless, contemporary systems may exist in the field for many years to come. In this paper we propose the Kronecker+ algorithm for polynomial multiplication in rings of the form $\mathbb{Z}[X]/(X^n+1)$: the arithmetic foundation of many lattice-based cryptographic schemes. We discuss how Kronecker+ allows for re-use of existing co-processors for post-quantum cryptography, and in particular directly applies to the various finalists in the post-quantum standardization effort led by NIST. We provide a detailed implementation analysis which highlights the potential of the Kronecker+ technique for commonly available multiplier lengths on contemporary co-processors. We validate this approach with an implementation of the algorithm running on an ARM Cortex-M4 core: the recommended embedded target platform by NIST.

1 Introduction

The number of IoT devices has steadily outgrown the number of humans living on this planet and is expected to keep increasing [22]. To secure these and many other devices, Elliptic Curve Cryptography (ECC) [19, 23] and the Rivest–Shamir–Adleman (RSA) [31] algorithm are vital components in our public-key infrastructure based on secure key exchange and digital signatures. On these embedded (IoT) devices speed is a key performance indicator. To enable them to securely and efficiently execute the complex cryptographic algorithms, many have access to dedicated hardware accelerators or so-called *co-processors*. Typically, for ECC and RSA these co-processors consist of a hardware-supported instruction set that enables the device to compute large-integer arithmetic routines efficiently and securely.

With the steady progress in the development of a quantum computer, the security of our public-key infrastructure is being threatened. When a full-scale quantum computer would become a reality, Shor's algorithm [35] is able to recover private keys used in ECC/RSA in polynomial time. To prepare for this threat, alternative cryptographic algorithms are necessary; these are called *post-quantum*, or *quantum-safe*, cryptographic algorithms. In an effort to standardize such algorithms the US National Institute of Standards and Technology (NIST) put out a call for proposals [25] for cryptographers to submit candidate algorithms in 2016. As of July 2020, seven out of fifteen remaining candidates are marked as finalists of which a subset is expected to be standardized in the upcoming three years.

For embedded devices the migration to completely new public-key cryptography algorithms results in multiple practical challenges. None of the seven finalists require large integer arithmetic, which is the computationally expensive operation in both ECC and RSA and exactly what is offered by existing public-key co-processors. Adding new dedicated hardware support means a significant investment for new generations of devices that cannot be started yet as it is not clear which candidate schemes will be standardized. Additionally, the design, testing and (most prominently) the migration time for these co-processors is expected to span many years if not decades.

Five of the finalists do have in common that they are so called ring-based lattice schemes. For these schemes the primary bottleneck in terms of performance is to multiply polynomials with integer coefficients: a typical example is to work with polynomials from $\mathbb{Z}[X]/(X^n+1)$ where n is a power of two. At CHES 2019, Albrecht, Hanser, Hoeller, Pöppelmann, Virdia and Wallner [1] proposed to apply Harvey's improvements [15] to Kronecker substitution [20, 33] to convert polynomial multiplication to large integer multiplication and thereby unlock the potential of the already existing

co-processors. Subsequently, this approach was explored by Wang, Gu and Yang for application to post-quantum crypto scheme Saber [39].

In this paper we expand on this line of research and present a new method to realize fast polynomial arithmetic implementations on embedded devices which have access to commonly used arithmetic co-processors. We show how one can generalize Harvey's negated-evaluation-points technique such that it works in polynomial rings as frequently used in post-quantum cryptography. Our new method can also be viewed as a *variant* of Nussbaumer polynomial multiplication [26] combined with Kronecker substitution, which we call *Kronecker+*. It opens up the possibility to compute a *symbolic* Number Theoretic Transform (NTT) in the polynomial rings used in the post-quantum cryptographic submissions, which in turn can be computed by multiple smaller integer multiplications using Kronecker substitution in a clever way. On contemporary co-processors this results in a more efficient polynomial multiplication, compared to existing approaches such as Schönhage-Strassen [32] and Nussbaumer [26]. More concretely, although the overhead of the forward and backward transforms is similar to that of Nussbaumer and Schönhage-Strassen, Kronecker+ halves the number or the bit size, respectively, of the required multiplications when compared to the aforementioned methods. In Table 1 an overview is presented of combining different approaches with Kronecker, including the new approach from this paper: Kronecker+.

In the setting of the NIST finalist Kyber [34] (which is almost identical to Saber [12] for our purposes), and more specifically for the parameters of Kyber-768 ($n = 256$ and $\ell = 32$), in [1] it is shown that one polynomial multiplication can be computed using the standard Kronecker substitution approach with a single multiplication of 8197 bits, and using Harvey's negated-evaluation-points technique with two integer multiplications of 4097 bits each. Kronecker+ enables further division into exactly $t = 2^\tau$ integer multiplications of $8196/t + 1$ bits each, where $\tau < 6$ is a positive integer, where the exact optimal choice of τ will strongly depend on platform-specific details.

Of course one could use asymptotically faster multiplication methods for the one or two large integer multiplications. For example, Karatsuba [18] replaces one multiplication of b bits with three multiplications of $b/2$ bits plus some overhead in the form of additions (or subtractions) and can be applied recursively. Moreover, r-way Toom-Cook [10, 37] generalizes this multiplication approach and replaces one b-bit multiplication with $2r-1$ multiplications of approximately b/r bits plus some overhead for the evaluation and interpolation formula used ($r = 2$-way Toom-Cook is approximately equivalent to one layer of Karatsuba). This immediately highlights the potential of our new approach: while r-way Toom-Cook can reduce one b-bit multiplication to $2r-1$ multiplications this can be done with r multiplications using Kronecker+ (where both approaches reduce to approximately b/r-bit multiplications). A high-level overview on how to perceive our contribution in light of the many other available multiplication approaches one could try with Kronecker (something which was not done or considered in case of Schönhage-Strassen and Nussbaumer in previous work) is shown in Table 1.

Algorithm	# Muls	# Bits
Kronecker (KS1)	1	$\ell n+1$
Harvey (KS2/KS3)	2	$\ell n/2+1$
Harvey (KS4)	4	$\ell n/4+1$
Kronecker + Karatsuba	$3^{\log t}$	$(\ell n+1)/t$
Kronecker + Toom-Cook-t	$2t-1$	$(\ell n+1)/t$
Kronecker + Schön.-Strassen	t	$2\ell n/t+t+1$
Nussbaumer + Kronecker	$2t$	$\ell n/t+1$
Kronecker+ (this work)	t	$\ell n/t+1$

Table 1: Comparison of number of multiplications of certain bit length required for multiplying two polynomials with n coefficients each. Here ℓ is the parameter for Kronecker substitution (i.e., evaluating at 2^ℓ) and t specifies the depth (if applicable) of the algorithm.

It should be noted that the number of multiplications of course does not tell the full story since each of the methods is accompanied by transformational overhead, usually in the form of additions or multiplications by small constants. However, the overhead of Kronecker+ is comparable to that of a regular NTT and of complexity $O(t \log t)$ by employing the Cooley-Tukey butterfly approach [11]. In fact, because of the smaller sizes of integers that we operate on, the overhead will be smaller than for Schönhage-Strassen and Nussbaumer. Especially for small values of τ, the overhead of the transformations is small even when compared to layers of Karatsuba or Toom-Cook. See Table 4 for a detailed comparison of Kronecker+ against Karatsuba.

In short, we summarize our contributions as follows:

1. We design Kronecker+, a polynomial multiplication method that generalizes the techniques of Harvey and Nussbaumer to reduce polynomial to integer multiplications, enabling the use of contemporary co-processors.

2. We provide a detailed performance analysis of Kronecker+. In doing so, we also analyze the efficiency of the existing multipoint Kronecker substitution whose runtime was only computed asymptotically.

3. We implement the algorithm on an ARM Cortex-M4 core to validate the theoretical runtime analysis, and integrate it into the NIST finalist Saber to demonstrate its applicability to leading post-quantum cryptography schemes. Based on this, we provide performance estimates for Saber with the aid of co-processors and show that Kronecker+ has huge potential to improve its efficiency on a wide variety of co-processors.

For the benefit of the reader, we include high-level Sage [36] code to simplify verification of the correctness of Kronecker+.[1] We shall also release the Cortex-M4 implementation in C and assembly that was used to generate the various cycle counts.

2 Preliminaries

2.1 NIST PQC Candidates

Of the fifteen remaining candidates in the third round of the NIST standardization effort, seven are lattice-based. These are CRYSTALS-Kyber [34], NTRU [40], Saber [12] (KEM finalists), CRYSTALS-Dilithium [21], Falcon [29] (digital signature finalists), FrodoKEM [24] and NTRU Prime [4] (KEM alternates). For our purposes the most interesting candidates are Kyber, Saber and Dilithium whose main ring operations are performed in $\mathbb{Z}_q[X]/(X^n+1)$ for some choice of q and n fixed across all parameter sets. Many of our results can also be applied to NTRU and NTRU Prime, though one would have to change rings which in practice means doubling the size of the involved polynomials. A similar statement holds for the non-FFT operations in Falcon, while FrodoKEM does not contain any polynomial multiplication at all.

2.2 Polynomial Multiplication

Let $f=\sum_{i=0}^{n-1} f_i X^i$ and $g=\sum_{i=0}^{n-1} g_i X^i \in \mathbb{Z}[X]$ be two polynomials of degree less than n. In this section we describe various methods that exist in the literature to compute the multiplication $h=(f\cdot g) \bmod (X^n+1)$. For many algorithms the reduction modulo X^n+1 has little effect, as it is only applied (in a straightforward manner) after the more involved multiplication in $\mathbb{Z}[X]$. However, we include it here as it is crucial for some of the polynomial multiplication algorithms under discussion (e.g., Nussbaumer and Schönhage-Strassen) and is relevant for cryptographic schemes that apply them (e.g., Kyber and Saber). For these algorithms we introduce an additional parameter t, which is a positive integer dividing n.

Note that in this section we are assuming that the polynomial multiplications are performed in (quotient rings of) $\mathbb{Z}[X]$. Most algorithms however can be applied more generally over other rings $\mathcal{R}$. In this case there is an extra assumption that multiplication by t and $2t$ are injective maps, and hence that they can be inverted. This holds of course for $\mathbb{Z}$, but does not necessarily hold for general rings $\mathcal{R}$.

All algorithms that we describe are well known, so one could argue that a detailed description is not necessary. We have opted to include it here for completeness, as they appear scattered over the literature with varying notation and level of detail. We add the relevant references in the respective sections. Moreover, presenting them in a single framework with unified notation will make it easier to introduce our own contributions (and, hopefully, highlight the elegance of their simplicity). Finally, inspired by Harvey [15], we set up a small running example to ease the comparison between different (relevant) multiplication methods.

Example 1. *Let f and g be the two arbitrarily chosen polynomials $f(X)=-3-3X^2-X^3-2X^5+X^6-3X^7$ and $g(X)=3+X+3X^2+2X^3-2X^4+3X^5+3X^6$, whose coefficients can be represented with 3 bits in the interval $[-3,3]$. The goal is to compute the product $h=(f\cdot g) \bmod X^8+1$, which is easily checked to be the polynomial $h(X)=7+3X-4X^2-15X^3+2X^4-15X^5-4X^6-21X^7$.*

2.2.1 Karatsuba and Toom-Cook

Karatsuba [18] and its generalization Toom-Cook [10, 37] are multiplication methods which are asymptotically faster compared to the schoolbook algorithm, which runs in $O(N^2)$ for $N\times N \to 2N$ bit multiplication. The idea behind k-way Toom-Cook (where $k=1$ is equal to schoolbook and $k=2$ essentially to Karatsuba) is to split the single N-bit multiplication into $2k-1$ multiplications of approximately N/k bits such that the run-time is $O(N^{\log(2k-1)/\log(k)})$. This is done by evaluating the polynomials at $2k-1$ distinct points, and performing an interpolation after having performed $2k-1$ smaller multiplications. See [6] for more details on how to optimally compute the Toom-Cook multiplication.

The 2-way ($O(N^{1.585})$), 3-way ($O(N^{1.465})$) or 4-way ($O(N^{1.404})$) version of Toom-Cook are popular approaches to multiply medium-sized integers and have been applied in a variety of settings in cryptography.

2.2.2 Fast Fourier Transform in a Finite Field

Pollard showed how to define a transform in the finite field $\mathbb{Z}_q$ of integers modulo a prime q, analogous to the discrete Fourier transform, which can be computed using a Fast Fourier Transform (FTT) algorithm [28]. In cryptography this is often referred to as the Number Theoretic Transform (NTT). In this case we want to compute a polynomial product of $f,g\in\mathbb{Z}_q[X]/(X^n+1)$, where $2n \mid (q-1)$, so that the multiplicative group $\mathbb{Z}_q^*$ contains a principal $2n$-th root of unity ζ. We can then use the Chinese remainder theorem to construct the isomorphism

$$\begin{aligned}\mathbb{Z}_q[X]/(X^n+1) &\cong \textstyle\prod_{i=0}^{n-1}\mathbb{Z}_q[X]/(X-\zeta^{2i+1})\\ f &\mapsto (f(\zeta^1),f(\zeta^3),\ldots,f(\zeta^{2n-1}))\,.\end{aligned}$$

By applying this isomorphism to f and g, their product can be reduced to n multiplications in $\mathbb{Z}_q$. As q is typically fairly small (e.g., 12 and 23 bits for the latest versions of Kyber and Dilithium, respectively), this is not interesting for modern co-processors which are aimed at hundreds or thousands of bits for ECC or RSA. Note that similar constructions can be

[1] https://joostrenes.nl/software/kroneckerplus.tar.gz

made with n-th principal roots of unity, requiring only that $n \mid (q-1)$, which is done for instance by Kyber. We do not elaborate on this further here.

2.2.3 Nussbaumer

This algorithm was designed in 1980 and is named after its creator [26]. We base our description on those of Bernstein [5, §9] and the bachelor thesis of van der Lubbe [38, §3.1]. The first step is to apply the transformation

$$\Psi : \mathbb{Z}[X]/(X^n+1) \to (\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t), \quad (1)$$
$$f = \sum_{i=0}^{n-1} f_i X^i \mapsto \Psi(f) = \sum_{i=0}^{t-1} \left(\sum_{j=0}^{n/t-1} f_{i+jt} Y^j\right) X^i.$$

As the polynomial $\Psi(f)$ has degree less than t in X, we can trivially lift it to $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]$ and view it as a polynomial in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(X^{2t}-1)$ where the coefficients of $\Psi(f)$ for the monomials X^i for $i = t, \ldots, 2t-1$ are 0. Similarly we obtain $\Psi(g)$.

As the coefficient ring $\mathbb{Z}[Y]/(Y^{n/t}+1)$ contains the $2t$-th principal root of unity $\zeta = Y^{n/t^2}$, we can apply a cyclic NTT with respect to ζ. For this we require the additional restriction on t that $t^2 \mid n$, as opposed to only $t \mid n$. More concretely, we compute

$$\Psi(h)(\zeta^i) = \Psi(f)(\zeta^i) \cdot \Psi(g)(\zeta^i), \quad \text{for } i = 0, \ldots, 2t-1, \quad (2)$$

from which we can compute $2t \cdot H_j = \sum_{i=0}^{2t-1} \Psi(h)(\zeta^i)\zeta^{-ji}$. As $\Psi(h) = \sum_{j=0}^{2t-1} H_j(Y)X^j$, we can recover h by dividing by $2t$ (recall that multiplication by $2t$ was assumed to be injective), reducing modulo $Y - X^t$ and inverting Ψ.

The main cost of the algorithm is in Equation (2). The simultaneous evaluation of $\Psi(f)$ at all the roots of unity ζ^i can be computed with a Cooley-Tukey butterfly algorithm [11] with complexity $O(t \log t)$. The multiplications are in the ring $\mathbb{Z}[Y]/(Y^{n/t}+1)$, so consist of $2t$ multiplications of polynomials with n/t coefficients each. Of course there is a clear possibility for recursion, but such an analysis is not of interest for our purposes.

This Nussbaumer algorithm was applied [38] to to the post-quantum crypto scheme NewHope [2, 7] and in the setting of homomorphic encryption [8].

Example 2. *Let f and g be as chosen in Example 1. As Nussbaumer requires that $t^2 \mid n$, the largest choice for $n = 8$ we can make is $t = 2$. Splitting the ring, we find*

$$\Psi(f) = Y^3 - 3Y - 3 + (-3Y^3 - 2Y^2 - Y) \cdot X,$$
$$\Psi(g) = 3Y^3 - 2Y^2 + 3Y + 3 + (3Y^2 + 2Y + 1) \cdot X,$$

as polynomials modulo $Y - X^2$. Lifting to $(Z[Y]/(Y^4+1))[X]$ and viewing them as polynomials modulo $X^4 - 1$ by setting the coefficients of X^2 and X^3 to be zero, we can apply the cyclic NTT with respect to 4-th root of unity Y^2. In other words,

$$[\Psi(f)(1), \Psi(f)(Y^2), \Psi(f)(Y^4), \Psi(f)(Y^6)] = [-2Y^3 - 2Y^2 - 4Y - 3, -1, 4Y^3 + 2Y^2 - 2Y - 3, 2Y^3 - 6Y - 5],$$
$$[\Psi(g)(1), \Psi(g)(Y^2), \Psi(g)(Y^4), \Psi(g)(Y^6)] = [3Y^3 + Y^2 + 5Y + 4, 5Y^3 - Y^2 + 3Y, 3Y^3 - 5Y^2 + Y + 2, Y^3 - 3Y^2 + 3Y + 6].$$

Multiplying pairwise modulo $Y^4 + 1$, we get $[\Psi(h)(1), \Psi(h)(Y^2), \Psi(h)(Y^4), \Psi(h)(Y^6)]$ as $-31Y^3 - 25Y^2 - 23Y + 12, -5Y^3 + Y^2 - 3Y, 11Y^3 + 5Y^2 + 7Y + 6, 25Y^3 - 5Y^2 - 45Y - 30]$. Inverting the transform with respect to Y^2 (i.e., applying the same NTT with Y^6), we obtain $[-6Y^2 - 16Y - 3, -21Y^3 - 15Y^2 - 15Y + 3, -10Y^3 - 4Y^2 + 8Y + 12, 0]$. Finally setting $Y = X^2$, viewing the tuple as the coefficients of a polynomial in X (where 0 is the coefficient of X^3) and reducing modulo $X^8 + 1$, we obtain h.

2.3 Utilizing Integer Multipliers

The problem of multiplying polynomials and the problem of integer multiplication are extremely related. The two can be linked by way of Kronecker substitution, which we first expand on in §2.3.1. This method has been further investigated by Harvey and afterwards been applied to Kyber and Saber, which we explain in §2.3.2. Finally we consider the Schönhage-Strassen algorithm in §2.3.3.

2.3.1 Kronecker Substitution

In 1882, Kronecker introduced a method to reduce computational problems related to multivariate polynomials to those related to univariate polynomials [20]. A hundred years later, a similar technique was introduced by Schönhage to reduce polynomial multiplications in $\mathbb{Z}[X]$ to integer multiplication (multiplication in $\mathbb{Z}$) [33]. This approach is known as the *Kronecker substitution method*.

The idea behind the method is to evaluate the polynomials at a sufficiently high two-power 2^ℓ for a positive integer ℓ, and use the resulting integers as input for a regular integer multiplication by computing $h(2^\ell) = f(2^\ell) \cdot g(2^\ell)$. Finally, the resulting integer $h(2^\ell)$ is converted back to its polynomial representation h. The result is correct if the coefficients of the resulting polynomial did not "mix" with each other, i.e. if the parameter $\ell \in \mathbb{Z}$ is sufficiently large.

The main advantage of this approach, computing a polynomial multiplication by way of an integer multiplication, is that well-studied and fast implementations of asymptotic integer multiplication methods can be used. It allows contemporary co-processors containing integer-multiplication acceleration for speeding up "classical" cryptography to be re-used for the polynomial multiplications that appear in post-quantum cryptographic primitives. This was first investigated by Albrecht, Hanser, Hoeller, Pöppelmann, Virdia and Wallner [1], who used an RSA co-processor for the implementation of

Kyber-768, and subsequently applied by Wang, Gu and Yang to an implementation of Saber [39].

Note that [1] applies this technique to polynomial multiplication modulo X^n+1, as opposed to generic multiplication. Interestingly, although the coefficients of $f \cdot g$ and $(f \cdot g) \bmod X^n+1$ differ, their *upper and lower bound* do not. Indeed, a coefficient of $f \cdot g$ can be the sum of *at most* n products of coefficients of f and g, while a coefficient of $(f \cdot g) \bmod X^n+1$ is the sum of *exactly* n such coefficient products. Therefore the choice of ℓ remains the same regardless of reduction modulo X^n+1. In particular, this implies that reduction modulo X^n+1 can also be done as an intermediate step in the Kronecker domain as reduction modulo $2^{n\ell}+1$.

Example 3. *Let f and g be as chosen in Example 1. As they have (at most) 8 coefficients and they lie in the interval $[-3,3]$, the coefficients of h (modulo X^n+1) lie in the interval $[-8\cdot 3^2, 8\cdot 3^2]$ and can therefore be represented with $\ell = 8$ bits. Therefore, we find*

$$f(2^8) = -215893506177302531, g(2^8) = 847714908832003,$$

and compute the product $h(2^8) \equiv 16932392214669820680 \bmod 2^{64}+1$. Notice that here we apply the intermediate reduction modulo X^8+1 in the Kronecker domain as reduction modulo $2^{64}+1$. We retrieve the coefficients of h by converting to a base-256 representation. As f and g have signed coefficients in $[-3,3]$, it is important to also take the signed *representation*

$$h(2^8) = 7+3\cdot 2^8 - 4\cdot 2^{16} - 15\cdot 2^{24} + 2\cdot 2^{32} - 15\cdot 2^{40} - 4\cdot 2^{48} - 21\cdot 2^{56}$$

with 8-bit limbs in $[-8\cdot 3^2, 8\cdot 3^2]$. Interestingly, one can also apply Kronecker substitution to the intermediate multiplication in Nussbaumer (see Example 2). Evaluating $\Psi(f)(Y^i)$ and $\Psi(g)(Y^i)$ for $i = 0,2,4,6$ at 2^8 leads to the tuples $[-33686531,-1,67239421,33552891]$, $[50398468,83821312,50004226,16581382]$, which are pairwise multiplied to $[-521737972,-83821312,184878854,419091170]$. The multiplications are to be reduced modulo Y^4+1, and hence modulo $2^{32}+1$ in the Kronecker domain. From here the regular Nussbaumer algorithm can be followed, with an additional final recovery to polynomial representation. These are 4 multiplications of (approximately) 32 bits each, as opposed to 1 multiplication of 64 bits for regular Kronecker.

2.3.2 Multipoint Kronecker Substitution

The size of the integers that are multiplied when applying Kronecker substitution, which impacts the efficiency of the algorithm, is strongly related to the size of ℓ. Simply put, the larger ℓ, the larger the integers and the slower the multiplication. On the other hand, ℓ needs to be at least as large as the maximum bit length of the coefficients of h in order to recover the polynomial h from $h(2^\ell)$ correctly.

One of the main observations made by Harvey [15, §3.3] was that the size of ℓ can be reduced by splitting up the polynomial evaluation into two parts. Assuming for simplicity that ℓ is even, Harvey computes

$$h(2^{\ell_2}) = f(2^{\ell_2})g(2^{\ell_2}), \quad h(-2^{\ell_2}) = f(-2^{\ell_2})g(-2^{\ell_2}),$$

where $\ell_2 = \ell/2$. He then observes that

$$h^{(0)}(2^\ell) = (h(2^{\ell_2}) + h(-2^{\ell_2}))/2,$$
$$h^{(1)}(2^\ell) = (h(2^{\ell_2}) - h(-2^{\ell_2}))/(2\cdot 2^{\ell_2}),$$

where $h^{(i)}$ denotes the polynomial whose j-th coefficient equals the $(2j+i)$-th coefficient of h. In other words,

$$h^{(0)}(2^\ell) = \textstyle\sum_{j=0}^{n/2-1} h_{2j} 2^{j\ell}, \text{ and } h^{(1)}(2^\ell) = \sum_{j=0}^{n/2-1} h_{2j+1} 2^{j\ell}.$$

The coefficients of h can therefore be recovered as the ℓ-bit limbs $h^{(0)}(2^\ell)$ and $h^{(1)}(2^\ell)$.

Denoting by $M(b)$ the cost of multiplying two b-bit integers, this approach changes the cost of the polynomial multiplication in $\mathbb{Z}[X]$ from $M(\ell n) + O(\ell n)$ in the case of standard Kronecker substitution, to $2\cdot M(\ell n/2) + O(\ell n)$. Here the big-$O$ terms incorporate the cost of packing and unpacking. This can lead to a significant speedup whenever the cost of multiplying is (relatively) expensive. In particular, this approach was used in the ring $\mathbb{Z}[X]/(X^{256}+1)$ with applications to Kyber (see [1]) and Saber (see [39]) to speedup their respective implementations based on co-processors.

In fact, Harvey considers a second approach to split up the evaluation into *four* parts by also evaluating at the *reciprocal* $f(2^{-\ell})$, that gives rise to multiplication with a cost of $4\cdot M(\ell n/4) + O(\ell n)$. We omit the details as we do not discuss it further. In particular, it was not considered practical in the previously mentioned implementations of Kyber and Saber.

Example 4. *Let f and g be as chosen in Example 1 and choose $\ell = 8$ as in Example 3. We compute*

$$[f(2^4), f(-2^4)] = [3504336126, 824184061],$$
$$[g(2^4), g(-2^4)] = [53355283, 47047411],$$

from which we obtain $[h(2^4), h(-2^4)] \equiv [2870021177, 1290988502]$ with two multiplications modulo $2^{32}+1$. It follows that

$$h^{(0)}(2^8) = 8 + 252\cdot 2^8 + 1\cdot 2^{16} + 252\cdot 2^{24},$$
$$h^{(1)}(2^8) = 4 + 241\cdot 2^8 + 240\cdot 2^{16} + 234\cdot 2^{24},$$

or, in signed representation, that

$$h^{(0)}(2^8) \equiv 7 - 4\cdot 2^8 + 2\cdot 2^{16} - 4\cdot 2^{24},$$
$$h^{(1)}(2^8) \equiv 3 - 15\cdot 2^8 - 15\cdot 2^{16} - 21\cdot 2^{24}$$

modulo $2^{32}+1$. The coefficients from h can now simply be read off. Note that this requires only 2 multiplications of (about) 32 bits each, compared to 4 for Nussbaumer combined with Kronecker.

2.3.3 Schönhage-Strassen

For the description of the Schönhage-Strassen algorithm [32], we base ourselves on the nice exposition of the implementation in the GMP library [14, §1] and Bernstein's paper [5, §9]. We assume that the integers we multiply are outputs of Kronecker substitution of the form $F = f(2^\ell)$ and $G = g(2^\ell)$, and we want to compute their product $H = h(2^\ell)$ in $\mathbb{Z}[X]/(2^{\ell n}+1)$, i.e., modulo the polynomial modulus (X^n+1) evaluated at 2^ℓ. Interestingly, we begin by viewing the integers as polynomials by applying the map

$$\Phi : \mathbb{Z}/(2^{\ell n}+1) \to \mathbb{Z}[X]/(X^t+1)$$
$$F = \textstyle\sum_{i=0}^{t-1} F_i \cdot 2^{\ell n/t} \mapsto \Phi(F) = \sum_{i=0}^{t-1} F_i \cdot X^i,$$

in other words viewing the $\ell n/t$-bit limbs as coefficients of a polynomial of degree (at most) $t-1$. Note that here we can assume that $F_t = 0$, as $F = f(2^\ell)$ is a polynomial with degree at most $n-1$ evaluated at 2^ℓ, and hence is strictly smaller than $2^{n\ell}$ (and similarly for G). It can be shown that the coefficients of $\Phi(F)\Phi(G)$ can be represented with $2\ell n/t + t$ bits [14, §1], implying that it can be recovered as the unique representative of the product of $\Phi(F)$ and $\Phi(G)$ embedded in $\mathbb{Z}/(2^{2\ell n/t+t}+1)[X]/(X^t+1)$. The main observation now is that the coefficient ring for this multiplication is $\mathbb{Z}/(2^{2\ell n/t+t}+1)$, which contains a principal $2t$-th root of unity $\zeta_t = 2^{2\ell n/t^2+1}$, under the additional assumption that $t^2 \mid 2\ell n$ (note that this is weaker than Nussbaumer, which requires $t^2 \mid n$). We can use the (principal) t-th root of unity ζ_t^2 to construct a *negacyclic* NTT to reduce this multiplication to t multiplications in $\mathbb{Z}/(2^{2\ell n/t+t}+1)$, of approximately $2\ell n/t$ bits each (assuming $n \gg t$), after which we can invert Φ to recover H.

Example 5. *Let f and g be as chosen in Example 1. As we require that $t^2 \mid 2\ell n = 128$, the largest choice we can take is $t = 8$. In that case*

$$\Phi(F) = [254, 255, 252, 254, 255, 253, 0, 253],$$
$$\Phi(G) = [3, 1, 3, 2, 254, 2, 3, 0],$$

representing polynomials in $\mathbb{Z}[X]/(X^8+1)$. Applying a negacyclic NTT with 8-th root of unity $\zeta_8^2 = 64$, we obtain the tuples [3191766, 12617514, 13706294, 6361802, 15707175, 16751308, 5128135, 10424123], [1893579, 12329652, 12869428, 336707, 1760443, 3940051, 4415315, 12786500]*, that are multiplied pairwise to* [2864000, 10297389, 10680185, 15308322, 14753371, 6584086, 650929, 5442338]*, modulo $2^{24}+1$ (note that 24 here is the first multiple of 8 larger than 16). Inverting the negacyclic NTT gives the tuple* [−66031, −65274, −4, −62734, 66295, 66799, 66806, 67813]*. Viewing these as the 8-bit limbs of an integer, we obtain the product 16932392214669820680 of F and G in $\mathbb{Z}/(2^{64}+1)$. This can be reverted to polynomial representation by inverting the Kronecker map (see Example 3).*

2.4 Public-key Hardware Co-processors

A typical hardware accelerator or cryptographic co-processor enhances the security and performance of hash-functions, random number generation, symmetric key or public-key cryptography. In the last category, the core of this accelerator is typically dedicated to multiplication and accumulation of large integers. One possible way of thinking about such hardware-supported instructions used to construct arbitrary length multiplication routines, is that given w-bit inputs a, b, and c it computes

$$(a \oslash c_1) \times (b \ominus c_2) + c \odot d$$

where $\oslash$, $\ominus$, and $\odot$ are optional operations with optional inputs c_1, c_2 and d. Concrete examples include (1) the multiply-and-accumulate instruction present on many modern computer architectures (omitting all optional operations), (2) the multiply-and-accumulate-accumulate (where $\odot$ equals the "+" operation) as present on the ARMv6 and above, and (3) the ARM barrel shifter where $\ominus$ could be a shift or rotate instruction. The multiply-and-accumulate-accumulate instruction can be used as a building block for arbitrary length multiplication and therefore also Montgomery multiplication: making this an essential building block for the most time-consuming operation in both RSA and ECC. Given the word size w, the instruction then computes $d = a \cdot b + c + d$ where all inputs are $< 2^w$ and the output is $\leq (w-1)^2 + 2(w-1) < 2^{2w}$.

Although the exact internal bit size of these co-processors is often kept secret, the word size is expected to be larger than the native word size on the embedded device (which is typically 8, 16, or even 32 bits). Typical examples of such co-processors are NXP's P71D321 [27], Infineon's SLE78 [16], or Espressif's ESP32 [13]. The accompanying technical document often state that these co-processors can be used to compute RSA (often up to 4096 bits) and ECC. It should be noted that the upper bound on the number of supported bits is often due to a restriction on the available memory.

3 Kronecker+

In this section we discuss a new multiplication technique that can be viewed as a generalization of the negated-evaluation-points idea by Harvey [15, §2.3] and as a variant of Nussbaumer when combined with Kronecker substitution. Its main improvement with respect to [15] is that there is less limitation on the depth: whereas Harvey's method reduces a polynomial multiplication to two integer multiplications that are half the length compared to Kronecker substitution, we allow reducing to t multiplications of fraction $(1/t)$ of the length. For this we require that $t \mid n$ and $t \mid \ell$, which in particular implies that that $t^2 \mid 2\ell n$ (as was needed for Schönhage-Strassen). Hence t cannot be chosen completely freely, but the degree of freedom is much larger than for Harvey.

Compared to Nussbaumer we reduce the number of required multiplications. As can be seen in §2.2.3, Nussbaumer

requires $2t$ multiplications of polynomials with n/t coefficients each. Applying Kronecker (i.e., evaluating at 2^ℓ) we would compute $2t$ multiplications of approximately $\ell n/t$ bits each. Instead, Kronecker+ requires only t such multiplications. The overhead of the forward and backward transformations is comparable.

3.1 An Alternative Transformation

We begin the description by revisiting the Nussbaumer algorithm, and proposing an alternative version. Initially, this will seem to serve no purpose as it does not lead to a reduced number of operations for polynomial multiplication. However, we show in §3.2 that this variant combines much better with Kronecker substitution and that Harvey's negated evaluation points technique can be considered a special case of our algorithm.

As usual, we assume that f and g are polynomials of degree (at most) $n-1$ in $\mathbb{Z}[X]/(X^n+1)$. Our alternative transformation starts identical to Nussbaumer by applying the map Ψ from Equation (1), obtaining $\Psi(f)$ and $\Psi(g)$ in the ring

$$(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t)\,.$$

This map is also used by [1, §4], which they refer to as "splitting the ring". They view $\Psi(f)$ and $\Psi(g)$ as degree $t-1$ polynomials in X, and multiply them through the schoolbook or Karatsuba algorithm, leading to t^2 or $3^{\log t}$ multiplications in $\mathbb{Z}[Y]/(Y^{n/t}+1)$ respectively. Alternatively, the strategy of Nussbaumer could be taken: canonically lift to $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]$ and embed in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(X^{2t}-1)$ to apply a *cyclic* NTT to $[F_0,\ldots,F_{2t-1}]$ and $[G_0,\ldots,G_{2t-1}]$ with respect to $2t$-th principal root of unity $\zeta_{2t}=Y^{n/t^2}$, where $\Psi(f)=\sum_{i=0}^{t-1}F_i(Y)X^i$ and $\Psi(g)=\sum_{i=0}^{t-1}G_i(Y)X^i$. This leads to the $2t$ multiplications

$$\Psi(h)(\zeta_{2t}^i)=\Psi(f)(\zeta_{2t}^i)\cdot\Psi(g)(\zeta_{2t}^i)\,,\quad \text{for } i=0,\ldots,2t-1\,,$$

in $\mathbb{Z}[Y]/(Y^{n/t}+1)$.

However, for Kronecker+ we deviate from both these approaches. As opposed to Nussbaumer, we only consider the length-t tuples $[F_0,\ldots,F_{t-1}]$ and $[G_0,\ldots,G_{t-1}]$ and take the principal t-th root of unity $\zeta_t=Y^{2n/t^2}$. Further, we apply weight factors X^i to the i-th element, i.e., apply a *cyclic* NTT with respect to ζ_t to the length-t tuples

$$[F_0\cdot X^0,\ldots,F_{t-1}\cdot X^{t-1}]\,,\ [G_0\cdot X^0,\ldots,G_{t-1}\cdot X^{t-1}]\,.$$

This results in the tuples

$$\left[\textstyle\sum_{i=0}^{t-1}\zeta_t^{ij}F_iX^i\right]_j,\text{ and }\left[\textstyle\sum_{i=0}^{t-1}\zeta_t^{ij}G_iX^i\right]_j.$$

An interesting observation at this point is that we can combine the application of Ψ and the NTT (including weight factors) in a single step, showing that the latter tuples are simply equal to $[f(\zeta_t^j\cdot X)]_j$ and $[g(\zeta_t^j\cdot X)]_j$ respectively. Although this is nice conceptually, we expect that an implementation of this algorithm would most likely separate Ψ from the NTT, making it easier to apply Cooley-Tukey-style butterflies [11] to the computation (see §3.3.2).

Next we perform the t multiplications

$$h(\zeta_t^i\cdot X)=f(\zeta_t^i\cdot X)\cdot g(\zeta_t^i\cdot X)\,,\text{ for } i=0,\ldots,t-1\,. \quad (3)$$

Inverting the NTT with respect to ζ_t (including dividing by t), removing the weight factors, and possibly performing an explicit reduction modulo $Y-X^t$, gives the result $\Psi(h)$. From this we can recover h by inverting Ψ.

It should be noted at this point that the polynomials $f(\zeta_t^i\cdot X)$ and $g(\zeta_t^i\cdot X)$ do not actually lie in $\mathbb{Z}[Y]/(Y^{n/t}+1)$, but instead still in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t)$. Therefore we have reduced a single multiplication in $(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t)$ to t of them. This does not make any sense from a performance perspective, and we do not suggest to use this method as described here for polynomial multiplication. However, in the next section we show that this approach has significant advantages in combination with Kronecker substitution.

3.2 Applying Kronecker

The true strength of reducing to the multiplications in Equation (3) comes from applying the (slightly modified) Kronecker substitution

$$K:(\mathbb{Z}[Y]/(Y^{n/t}+1))[X]/(Y-X^t)\to\mathbb{Z}/(2^{\ell n/t}+1)$$
$$F=\textstyle\sum_{i=0}^{t-1}F_i(Y)\cdot X^i\mapsto\sum_{i=0}^{t-1}F_i(2^\ell)\cdot 2^{i\ell/t}\,.$$

The mapping K maps $Y\mapsto 2^\ell$ and $X\mapsto 2^{\ell/t}$ and ensures that the map is well-defined modulo $Y-X^t$. In particular, this maps

$$\zeta_t\mapsto 2^{2\ell n/t^2}\,,$$
$$f(\zeta_t^i\cdot X)\mapsto f(2^{2\ell n/t^2}\cdot 2^{\ell/t})\,,\ g(\zeta_t^i\cdot X)\mapsto g(2^{2\ell n/t^2}\cdot 2^{\ell/t})\,.$$

Hence, the multiplications in Equation (3) can be reduced to t multiplications in $\mathbb{Z}/(2^{\ell n/t}+1)$. This means computing t multiplications of $\ell n/t+1$ bits each instead of a single multiplication of ℓn bits. Recall that combining Nussbaumer with Kronecker substitution leads to $2t$ such multiplications. For completeness, we summarize the proposed method in Algorithm 1 which we refer to as Kronecker+.

We can now see that Algorithm 1 is a generalization of the method of Harvey [15, §3.3]; setting $t=2$ and $\zeta_2=2^{\ell n/2}\equiv -1 \bmod 2^{\ell n/2}+1$ in Algorithm 1 is the same as applying Harvey's approach. In fact, we generalize his method by also considering the case $t>2$, whereas Harvey does not go beyond $t=2$ (at least not for negated evaluation points). This generalization is made possible by the existence of t-th roots of unity in $\mathbb{Z}[Y]/(Y^{n/t}+1)$ via the map Ψ on $\mathbb{Z}[X]/(X^n+1)$,

Algorithm 1 Pseudo-algorithmic *simple* description of Kronecker+.

Input: $f, g \in \mathbb{Z}[X]/(X^n+1)$ for a positive integer n, the Kronecker parameter ℓ and a positive integer t such that $t \mid \ell$ and $t \mid n$, and $M_i = 2^{2i\ell n/t^2} \cdot 2^{\ell/t}$ for $0 \le i < t$

Output: $h = \sum_{i=0}^{n-1} h_i X^i = f \cdot g \bmod X^n + 1$

1: Compute $f(M_i)$ and $g(M_i)$ for $i = 0, \ldots, t-1$.

2: Compute $h(M_i) = f(M_i) \cdot g(M_i) \bmod 2^{\ell n/t} + 1$ for $i = 0, \ldots, t-1$.

3: Compute

$$h^{(i)} = \frac{\sum_{j=0}^{t-1} 2^{2i(t-j)\ell n/t} h(M_i)}{t \cdot 2^{i\ell/t}} \bmod 2^{\ell n/t} + 1$$

for $i = 0, \ldots, t-1$.

4: Recover h_{i+tj} as the j-th ℓ-bit limb of $h^{(i)}$ for $0 \le i < t$ and $0 \le j < n/t$.

which do not exist for generic integer polynomial multiplication in $\mathbb{Z}[X]$. Of course we can always embed any integer polynomial of degree (at most) $n-1$ into a ring of the form $\mathbb{Z}[X]/(X^{2n}+1)$ and apply Ψ to reduce to t multiplications in $\mathbb{Z}[Y]/(Y^{2n/t}+1)$. This comes at the cost of approximately doubling the bit size for the multiplications.

To illustrate our algorithm, we provide an example. Note for comparison that in Example 6 we reduce the polynomial multiplication to 8 multiplications, each of 9 bits, using a transformation very similar to Schönhage-Strassen and Nussbaumer. However, Nussbaumer (see Example 2) requires that $t^2 \mid n$ and therefore only allows $t = 2$, while even in that case needing 4 multiplications (whereas Kronecker+ only uses 2). On the other hand, Schönhage-Strassen (see Example 5) allows for $t = 8$, but reduces to 8 multiplications of 25 bits each.

Example 6. *Let f and g be as chosen in Example 1, where $n = 8$, and choose $\ell = 8$ as in Example 3. Therefore we can choose $t = 8$ as well, as it divides both n and ℓ. As $n = t$, splitting the ring simply gives $\Psi(f) = f$ and $\Psi(g) = g$. Therefore, multiplying by the weights X^i we get tuples $[-3, 0, -3X^2, -X^3, 0, -2X^5, X^6, -3X^7]$, $[3, X, 3X^2, 2X^3, -2X^4, 3X^5, 3X^6, 0]$, respectively. Applying the map K that maps $X \mapsto 2^{\ell/t} = 2$ gives $[254, 0, 245, 249, 0, 193, 64, 130]$, $[3, 2, 12, 16, 225, 96, 192, 0]$, where each entry is taken modulo $2^{\ell n/t} + 1 = 2^8 + 1$. We now take the cyclic NTT with respect to the t-th root of unity $\zeta_t \mapsto 2^{2\ell n/t^2} = 4$ modulo $2^8 + 1$, leading to the tuples $[107, 228, 53, 131, 248, 161, 94, 239]$, $[32, 116, 51, 47, 61, 105, 254, 129]$. These are multiplied pairwise to give $[83, 234, 133, 246, 222, 200, 232, 248]$. Inverting the NTT leads to $[7, 6, 241, 137, 32, 34, 1, 139]$, and undoing the weights $X^i \mapsto 2^{i\ell/t} = 2^i$ in the Kronecker domain gives $[7, 3, 253, 242, 2, 242, 253, 236]$. Finally, converting to a signed representation gives the tuple $[7, 3, -4, -15, 2, -15, -4, -21]$, whose entries correspond to the coefficients of h.*

3.3 Algorithmics

Although the description in Algorithm 1 is nice and compact, it is not immediately clear how efficiently it can be implemented. Indeed, polynomial evaluations $f(M_i)$ and $g(M_i)$ initially appear to be of quadratic complexity in t, while multiplications by roots of unity and divisions by t and $2^{i\ell/t}$ could be costly. In this section we make some comments on algorithmic choices to implement Kronecker+ most optimally on modern (embedded) platforms.

3.3.1 Parameter choices

Even though it is not technically necessary for Kronecker+, division by t is most efficient to implement whenever it is a power of two $t = 2^\tau$. This means it can simply be computed as a bit shift by τ bits. The same holds for division by the weights $2^{i\ell/t}$. As it is very common to set ℓ to a power of two (since it makes Kronecker substitution easy to implement on modern platforms), this is not serious restriction. The requirement on Kronecker+ that $t \mid \ell$ then can be rewritten as the equivalent condition $\tau \le \log(\ell)$. Finally, since we must satisfy $t \mid n$, the choices of n most suitable for Kronecker+ are those that are divisible by a power of two.

These constraints appear very naturally in the context of cryptographic primitives. For example, the NIST KEM finalists Kyber ($q = 3329$) and Saber ($q = 2^{13}$) perform polynomial multiplication in $\mathbb{Z}_q[X]/(X^{256}+1)$. In that case $n = 256$ and one can show that $\ell = 32$ suffices (see §4.1), in which case we will have $\tau \le 5$. We emphasize that in this case applying $\tau = 2$ reduces the 8192-bit multiplication (by Kronecker substitution) to 4 multiplications of approximately 2048 bits each, which can be handled by the RSA co-processor in [1]. Instead, in their work they resort to Harvey's method with an additional layer of schoolbook multiplication, resulting in 8 multiplications of 2048 bits each. In this case Kronecker+ therefore halves the number of required multiplications: already showing its strength compared to alternatives.

Although the description in §3.2 is nice and compact, it is not immediately clear that this can be efficiently implemented. Indeed, polynomial evaluations $f(M_i)$ and $g(M_i)$ initially appear to be of quadratic complexity in t, while multiplications by roots of unity and divisions by t and $2^{i\ell/t}$ could be costly. Therefore we comment here on the circumstances in which Kronecker+ could best be used.

3.3.2 Butterfly operations

There is further reason to set t to be a power of two. Line 1 of Algorithm 1 can be decomposed into three steps:

1. Compute $F_i(2^\ell)$ and $G_i(2^\ell)$ for $i = 0, \ldots, t-1$,
2. Compute $2^{i\ell/t} F_i(2^\ell)$ and $2^{i\ell/t} G_i(2^\ell)$ for $i = 0, \ldots, t-1$,

3. Compute $f(M_i) = \sum_{j=0}^{t-1} 2^{2ij\ell n/t^2} F_j(2^\ell) 2^{j\ell/t}$ and $g(M_i) = \sum_{j=0}^{t-1} 2^{2ij\ell n/t^2} G_j(2^\ell) 2^{j\ell/t}$.

The first step is essentially free if the coefficients are positive (as they can be all be represented with less than ℓ bits), since it is just a matter of reordering the coefficients of f and g. It becomes more complicated when the coefficients are signed, since we have to take carries into account. This becomes particularly tedious when attempting to compute this in constant time, as f and g can contain secret information in certain settings (e.g., for Kyber and Saber). It is mentioned by [1, §5.3] that this costs some performance, but no further details are given. We propose a fairly simple solution in the case that all coefficients are greater than $-q$ for some integer q, for example when sampled from $\mathbb{Z}_q$ in an interval of size q centered around 0. Letting $Q = \sum_{j=0}^{n/t-1} qX^j$, we compute

$$\begin{aligned} F_i(2^\ell) &= (F_i + Q)(2^\ell) - Q(2^\ell), \\ G_i(2^\ell) &= (G_i + Q)(2^\ell) - Q(2^\ell), \end{aligned} \tag{4}$$

noting that $F_i + Q$ and $G_i + Q$ are polynomials with positive coefficients. In settings where q is known in advance (which is the case for all relevant cryptographic schemes such as Kyber and Saber), the polynomial $Q(2^\ell)$ can be precomputed. The additional cost is therefore t additions of polynomials of degree $n/t-1$ and t subtractions in $\mathbb{Z}/(2^{\ell n/t}+1)$, both of which are very easily implemented in constant time.

The second step requires shifts by $2^{i\ell/t}$, which are cheap and often even free through the use of (for example) barrel shifters. Moreover, if $i\ell/t$ is a multiple of the word size of the platform, then such shifts can be implemented by simply relabeling words. There can potentially be an additional cost by adding a reduction modulo $2^{n\ell/t}+1$, but this has linear cost and can even be completely avoided by using lazy reduction techniques.

The main cost comes from the third step. However, the main advantage of decomposing Line 1 of Algorithm 1 in this fashion should now be clear: computing the linear combinations has naïve complexity of $O(t^2)$ operations, but can instead be implemented with complexity $O(t \log t)$ by using Cooley-Tukey butterflies [11]. These are particularly easy to implement when t is a power of two. Note that the butterfly algorithm also requires several multiplications by roots of unity $2^{2ij\ell n/t^2}$, but these are constructed to be powers of two. Therefore these multiplication operations can be computed with simple shifts, or by relabeling words if $2ij\ell n/t^2$ is a multiple of the word size. In addition, one can see that many of them vanish modulo $2^{\ell n/t}+1$. Completely analogous statements apply to Line 3 of Algorithm 1, which is in essence an inverse NTT. By way of example, we summarize the required operations for $n = 256$ and $\ell = 32$ (the setting of Kyber and Saber) for $\tau = 1,2,3$ in Figure 1. A similar structure is preserved for larger τ, but becomes more tedious to write down. The operations demonstrate that the number of additions/subtractions is simply $t \log t$, as expected. In particular, it shows that the case $t = 2$ reduces to the same operations as computed by Harvey [15].

$t = 2$

$(F_0, F_1) = (F_0 + F_1, F_0 - F_1)$.

$t = 4$

$(F_0, F_1, F_2, F_3) = (F_0 + F_1, F_0 - F_1, F_2 + F_3, F_2 - F_3)$,
$(F_0, F_2, F_1, F_3) = (F_0 + F_2, F_0 - F_2, F_1 + (F_3 \ll 1024), F_1 - (F_3 \ll 1024))$.

$t = 8$

$(F_0, F_1), (F_2, F_3) = (F_0 + F_1, F_0 - F_1), (F_2 + F_3, F_2 - F_3)$,
$(F_4, F_5), (F_6, F_7) = (F_4 + F_5, F_4 - F_5), (F_6 + F_7, F_6 - F_7)$,
$(F_0, F_2), (F_1, F_3) = (F_0 + F_2, F_0 - F_2), (F_1 + (F_3 \ll 512), F_1 - (F_3 \ll 512))$,
$(F_4, F_6), (F_5, F_7) = (F_4 + F_6, F_4 - F_6), (F_5 + (F_7 \ll 512), F_5 - (F_7 \ll 512))$,
$(F_0, F_4), (F_1, F_5) = (F_0 + F_4, F_0 - F_4), (F_1 + (F_5 \ll 256), F_1 - (F_5 \ll 256))$,
$(F_2, F_6) = (F_2 + (F_6 \ll 512), F_2 - (F_6 \ll 512))$,
$(F_3, F_7) = (F_3 + (F_7 \ll 768), F_3 - (F_7 \ll 768))$.

Figure 1: Example operations required for the NTT with t-th root of unity $2^{16384/t^2}$ for choices $\tau = 1,2,3$, where all operations take place in the ring $\mathbb{Z}/(2^{8192/t}+1)$. Here we write $F_i = 2^{32i/t} \cdot F_i(2^{32})$ and compute the NTT in place.

3.3.3 Multiplication

The multiplications in Line 2 of Algorithm 1 are technically the most straightforward operations: simply multiplying t pairs of integers modulo $2^{\ell n/t}+1$. The reduction can be performed by observing that $2^{\ell n/t} \equiv -1 \bmod 2^{\ell n/t}+1$ and subtracting the top half from the bottom. An interesting remark can be made about the size of the multiplier. As integers modulo $2^{\ell n/t}+1$ can be represented in $\ell n/t+1$ bits, while t is a power of 2, we will need a slightly awkward size. For example, for $n = 256$ and $\ell = t = 32$ we need a 257-bit multiplier. A similar problem arises in [1], who use a multiplier that can handle integers slightly larger than 2048 bits.

A particularly interesting case is where the internal word size w of the co-processor is fairly small. In that case, assuming for simplicity that $wt \mid \ell n$, we need exactly $\lceil (\ell n/t + 1)/w \rceil$ registers with a total of $\ell n/t + w$ bits to represent the $(\ell n/t+1)$-bit integers. As the $w-1$ redundant bits should be more than enough to accumulate the values during the forward butterfly (e.g., when $w = 64$ or $w = 128$), no reductions modulo $2^{\ell n/t}+1$ are necessary before multiplying values with $\lceil (\ell n/t+1)/w \rceil$ limbs each. This application of lazy reduction simplifies the computations of the butterflies.

3.3.4 Recovering polynomials

The final step in Kronecker+ (see Line 4 of Algorithm 1) is to convert the integer(s) result of the inverse NTT back to the polynomial representation in $\mathbb{Z}[X]/(X^n+1)$. This conversion is denoted as "sneeze" in [1, Algorithm 8] and also supports signed coefficients. This operation would be particularly simple if the coefficients h_i of $h = f \cdot g$ are guaranteed to be

positive, but this will not always be the case in cryptographic contexts. We make two simplifications compared to [1]: first, we remove the variable-time if-statement that depends on the (potentially secret) input that checks whether a limb is larger than $2^{\ell-1}$. We replace this by Line 5 in Algorithm 2 that computes the carry bit as a simple "or" operation of two bits. Second, we remove the explicit subtraction of 2^{ℓ} to move the values h_i from the interval $[0, 2^{\ell}-1]$ to $[-2^{\ell/2}, 2^{\ell/2}-1]$ since this can be achieved with a cast to a signed value, i.e., by simply interpreting the bit representation in a different way. The full algorithm is summarized in Algorithm 2.

Algorithm 2 Coefficient recovery in $\mathbb{Z}[X]/(X^n+1)$.

Input: t integers $h^{(i)} = \sum_{j=0}^{n/t} a_{i,j} \cdot 2^{j\ell}$ with $a_{i,j} \in [0, 2^{\ell}-1]$
Output: $h = \sum_{i=0}^{n-1} h_i \cdot X^i$ with $h_i \in [-2^{\ell/2}, 2^{\ell/2}-1]$

1: **for** $i = 0, 1, \ldots, t-1$ **do**
2: carry $\leftarrow 0$
3: **for** $j = 0, 1, \ldots, n/t-1$ **do**
4: limb $\leftarrow a_{i,j} +$ carry
5: carry $\leftarrow$ (limb $\gg (\ell-1)$) $\mid$ $(a_{i,j} \gg (\ell-1))$
6: $h_{j \cdot t+i} \leftarrow$ limb
7: $h_i \leftarrow h_i - (\text{carry} + a_{i,n/t})$

4 Implementation Results

The goal of this section is to support the claims on runtime made in §3.3 and to extrapolate these results to provide runtime estimates on platforms that are most relevant for Kronecker+. For this purpose we focus on the context of cryptographic implementations with the help of contemporary co-processors that support integer multiply or multiply-and-accumulate operations. In particular, we implement the Saber proposal and begin with a brief explanation on this setting in §4.1. Then we provide operation counts for this implementation in terms of multiply-and-accumulate and addition operations (where subtractions are counted as additions) based on the analysis of §3.3. We provide cycle counts for the various parts of our implementation on an ARM Cortex-M4 core: the recommended embedded target platform selected by NIST. For testing and benchmarking we used the pqm4 framework [17]: the de facto standard benchmarking and testing framework for embedded post-quantum cryptography. This framework allows us to relatively accurately predict the runtime of the Kronecker+ procedures based on the cycle counts for multiply-and-accumulate and addition operations. After demonstrating the effectiveness of this approach on the Cortex-M4 core, we then extrapolate this to predict the runtime of Saber routines on co-processors with larger multiplier and adder widths and show the feasibility of Kronecker+ on such platforms.

We emphasize that this approach is somewhat different from existing works that directly implement Kronecker variants on a fixed choice of co-processor. For example, [1] implement Kyber on the SLE78 with a (approximately) 2048-bit multiplier, while Wang, Gu and Yang implement Saber on the ESP32 with a big integer multiplier of 1536 or 2048 bits [39]. Our approach is much more theoretic: we provide an accurate complexity analysis of Kronecker+ based on multiply-and-accumulate and addition operations that does not rely on asymptotics (Harvey did provide asymptotics in his initial paper on multipoint Kronecker substitution [15], but they are not useful for practical performance). This allows to characterize the runtime of Kronecker+ much more accurately compared to [1] and [39]. The main drawback is that by relying on a computational model, one might question the appropriateness of the chosen model. However, as we show with the proof of concept Cortex-M4 implementation in §4.3, the performance of Kronecker+ is indeed dominated by multiply-and-accumulate and addition operations. Further, by abstracting away the platform we are also able to *remove* unwanted implementation specific details that muddy the efficiency analysis of Kronecker+. This is especially useful in scenarios where the co-processor is closed off, as is the case for SLE78, hindering reproducibility of the cycle counts. In our case, the resulting detailed complexity analysis is completely platform-independent and can be used to estimate the runtime on a variety of systems much more easily.

4.1 Kronecker+ for Saber

In this section we discuss the application of Kronecker+ to the Round 3 version of Saber. The arithmetic core in Saber is to multiply a $k \times k$ matrix $\mathbf{A}$ with a $k \times 1$ vector $\mathbf{s}$, where the entries of the matrix and vectors are elements of $\mathbb{Z}_q[X]/(X^{256}+1)$ for $q = 2^{13}$. This operation is typically referred to as `MatrixVectorMul`. A similar operation called `InnerProd` performs an inner product of two $k \times 1$ vectors. More concretely, the main operation is to multiply and accumulate k polynomials as $b_i = a_{i,0} \cdot s_0 + \ldots + a_{i,k-1} \cdot s_{k-1}$ which in turn has to be performed for $0 \leq i < k$ for `MatrixVectorMul`, and for $i = 0$ for `InnerProd`. When using Kronecker substitution, from a performance point of view, it is beneficial to multiply and accumulate these k polynomials in integer representation to avoid converting them back separately (requiring k times fewer inverse transformations of Kronecker substitution).

In order to determine the required precision in Kronecker (i.e. the parameter ℓ), the bounds on the input need to be determined. Recall from §2.3.1 that the bound ℓ is independent of the modulus $X^{256}+1$. The coefficients of the polynomial $a_{i,j}$ are uniform in $\mathbb{Z}_q$ and can be represented in the interval $[-q/2, q/2-1] = [-2^{12}, 2^{12}-1]$ since q is even. The coefficients of s_j are samples in $[-\nu, \nu]$, where $\nu = 5, 4, 3$ depending on the security level (here $\nu = \mu/2$ where $\mu = 10, 8, 6$ in the Saber specification). This means that the product of

2 coefficients lies in the interval $[-2^{12}\nu, (2^{12}-1)\nu]$, requiring at most $\lceil \log(2^{13}\nu - \nu + 1) \rceil$ bits to represent. As each coefficient of the product is an accumulation of $k \cdot 256$ such coefficients, for a signed version of Kronecker it is sufficient to set $\ell = \lceil \log(2^{13}\nu - \nu + 1) + \log(k \cdot 256) \rceil$. This results in $\ell = 25$ for all security levels of Saber. However, as also noted in [1], it is beneficial to use $\ell = 32$ such that it aligns nicely with the byte boundaries and 32-bit datatypes on modern computer architectures. Moreover, many of the required steps in Kronecker simplify significantly.

A completely analogous analysis can be made for Kyber and Dilithium, whose polynomial rings are very similar to that of Saber from the perspective of Kronecker+. However, for Kyber and Dilithium the situation is different. The approach taken in the NIST submission by Kyber optimizes for polynomial multiplication with the proposed NTT approach described in [34]. An example is that the large Kyber matrix $\mathbf{A} \in (\mathbb{Z}_q/(X^{256}+1))^{k \times k}$, for the Kyber modulus $q = 3329$ and the parameter $k \in \{2,3,4\}$ depending on the parameter set, is sampled directly into the NTT domain. These design decisions have an impact on the performance of alternative approaches. When considering the Kronecker approach, the authors of [1] note that this "basically nullify all gains from a different and faster algorithm for polynomial multiplication" and decide *not* to be compatible with the Kyber specification. We re-iterate that this comment is absolutely right, prohibiting efficient implementation of Kronecker+ for those schemes in their current shape.

4.2 Theoretical Model

In this section we give a theoretical estimate for the runtime of Kronecker+. For this purpose we subdivide the algorithm into various subroutines according to the description of §3.3, creating simple and small steps that allow for straightforward counting of multiply-and-accumulate and additions (or subtractions). Recall that Kronecker+ mostly works in the ring $\mathbb{Z}/(2^{\ell n/t}+1)$, meaning that all counted operations are of $\ell n/t + 1$ bits. The operation counts are summarized in Table 2 and are exactly as performed in our implementation.

The first step is Line 1 of Algorithm 1, which we divide up into the three steps listed in §3.3.2. The combination of the first two is referred to as `phi_and_shift`, while the third operation is a butterfly routine called `forward_bfly`. The butterfly can be implemented with $t \log t$ additions and $T(\tau)$ multiplications by roots of unities, where $T(\tau) = 0,0,1,5,17$ for $\tau = 0,1,2,3,4$ (see Figure 1 for examples). The roots of unities $2^{2ij\ell n/t^2}$ (for $0 \le i,j < t$) are powers of two; even better for $\tau \le 4$ (recall that $t = 2^\tau$) one always has $2ij\ell n/t^2 \equiv 0 \pmod{32}$ which means that all shifts come for free by relabeling and one just has to compute a modular reduction which can be implemented as an addition (even as a half addition on average, though we choose to be conservative here). The `phi_and_shift` routine does not require explicit multiplications or additions, but we mention it as it reveals an interesting step of the algorithm requiring memory operations. If a co-processor is employed, this would be the step where the data is loaded onto it. Although it does not contribute to the arithmetic cost, we decide to include it anyway as it should not be forgotten. Additionally, if one considers signed coefficients, one will have to follow up the routine with t subtractions by $Q(2^\ell)$ as described in Equation (4). This operation is referred to as `make_signed`.

Function	# Muls	# Adds
`phi_and_shift`	–	–
`make_signed`	–	t
`forward_bfly`	–	$t \log t + T(\tau)$
`make_positive`	–	t
`multiply`	t	$2t$
`backward_bfly`	–	$t \log t + T(\tau)$
`divide_twos`	–	$2t$
`recover_coeffs`	–	–

Table 2: Number of $(\ell n/t + 1)$-bit multiplications and additions used in our implementation for the various parts of the Kronecker+ algorithm where $T = [0,0,1,5,17,49,\ldots]$ is a lookup table containing the number of required multiplications by roots of unity in the butterfly operations which boil down to additions in practice.

The next operation is the multiplication in Line 2 of Algorithm 1, called `multiply`, which requires t multiplications and $2t$ additions for the reduction modulo $2^{\ell n/t}+1$. To simplify its implementation we assume that the inputs are positive, which is not necessarily true after the butterfly. Therefore we include the function `make_positive` that adds a fixed multiple of $2^{\ell n/t}+1$ large enough to make all values positive. This costs t additions.

Line 3 of Algorithm 1 starts with another butterfly operation `backward_bfly` with the same cost as `forward_bfly`. Afterwards the operation `divide_twos` divides by $t \cdot 2^{i\ell/t}$ for $i = 0,1,\ldots,t-1$. As $i\ell/t$ will be small, this can be implemented as a very small shift followed by two additions. Finally, Line 4 performs the steps laid out in Algorithm 2. Analogously to `phi_and_shift`, this has no arithmetic cost but can interpreted as loading the coefficients from the co-processor back to the host device (with some minor overhead). This operation is named `recover_coeffs`.

4.3 Cortex-M4 Implementation

We present cycle counts for the various routines of Kronecker+ defined in the previous section. The platform of choice is the STM32F4DISCOVERY development board containing a Cortex-M4 core. This enables us to use the pqm4 benchmark framework [17]. All measurements were taken with the `arm-none-eabi-gcc` cross compiler version 10.2.1.

Until now we have discussed the subroutines as a sequence of calls to $(\ell n/t+1)$-bit multiplications and additions, but it remains to discuss how to implement those themselves. For additions this is straightforward: with a w-bit multiplier (where $w = 32$ on the Cortex-M4) we use $\lceil(\ell n/t+1)/w\rceil$ additions (with carry) of word size w. For multiplications, several choices can be made. We make the pragmatic choice for schoolbook multiplication, whose complexity is easy to analyze. That is, it requires $\lceil(\ell n/t+1)/w\rceil^2$ multiply-and-accumulate operations on words of w bits. This does not necessarily lead to the fastest implementation on the Cortex-M4, as larger multiplications (for smaller choices of t) would definitely benefit from applying Karatsuba or Toom-Cook. However, we re-iterate that the goal is not to set a speed record on this particular platform, but rather to investigate the runtime of Kronecker+ in order to make realistic and meaningful predictions on platforms which are equipped with an arithmetic co-processor.

The implementation is derived from the C reference implementation of Saber,[2] where the calls to `poly_mul_acc` are replaced by Kronecker+. The implementation of Kronecker+ is written in C except the calls to multiplications and additions, for which assembly routines were taken from the GMP library version 6.2.1. More precisely, a multiplication of $\ell n/t+1$ bits integers requires approximately 286006, 73476, 19364, 5352, 1608 cycles for $\tau = 0,\ldots,4$ respectively, while an addition requires 1518, 781, 416, 232, 139 cycles. In particular, the cycle counts for multiplications and addition allow us to reconstruct the cycle counts for more involved routines fairly accurately. For example, looking at Table 2, a multiplication for $\tau = 2$ requires 4 multiplications and 8 additions with a total cost of $4\cdot 19364 + 8\cdot 416 = 80784$ cycles. Analogous computations can be done for other operations, where the difference between the actual cycle count and the estimate would reveal a disconnect between theory and practice.

Finally, we also include cycle counts for the main Saber operations `MatrixVectorMul` and `InnerProd`. We select the NIST Level 3 security level variant (itself called Saber) in which `MatrixVectorMul` performs a 3×3 matrix-vector multiplication where the matrix elements lie in $\mathbb{Z}_{2^{13}}$ and the vector elements are in $[-4,4]$. It is important to note that the polynomial products can be accumulated in the Kronecker domain. That is, we need to apply $9+3=12$ forward transformations for the matrix and the vector, but only 3 backwards transformations for the resulting 3×1 vector. For $\tau > 0$ ($\tau = 0$ is special as some operations can be completely ignored) we call $12\times$`phi_and_shift`, $12\times$`forward_bfly`, $3\times$`make_signed`, $12\times$`make_positive`, $9\times$`multiply`, $3\times$`backward_bfly`, $3\times$`divide_twos` and $3\times$`recover_coeffs`. Moreover, the accumulation itself requires a further $9t$ additions. Adding the respective cycle counts together from Table 3, we obtain a total of 1479k, 901k, 646k, 563k cycles for $\tau = 1,2,3,4$. Completely analogous computations can be done for $\tau = 0$ and `InnerProd`. This demonstrates that the cost of `MatrixVectorMul` and `InnerProd` is accurately described as a combination of the subroutines as described above. Note that compared to leading works on the Cortex-M4 [3, 9], we are indeed quite a bit slower. Again, the goal is not to provide speed records on this specific platforms, but rather to use it for estimating performance on more interesting systems.

[2] https://github.com/KULeuven-COSIC/SABER/tree/master/Reference_Implementation_KEM

Function	τ				
	0	1	2	3	4
`phi_and_shift`	1.5	1.8	2.3	2.5	1.7
`make_signed`	1.5	1.6	1.7	1.9	2.2
`forward_bfly`	0	2.0	4.4	7.6	12.5
`make_positive`	2.3	2.4	2.4	2.5	2.9
`multiply`	290	151	82	48	30
`backward_bfly`	0	2.0	4.7	7.8	12.5
`divide_twos`	0	4.0	4.6	4.1	5.1
`recover_coeffs`	2.3	2.8	2.9	3.2	1.6
`MatrixVectorMul`	2672	1486	907	652	568
Bermudo et al. [3]	317				
Chung et al. [9]	125				
`InnerProd`	905	513	326	249	230
Bermudo et al. [3]	99				
Chung et al. [9]	57				

Table 3: Cycle counts of Kronecker+ operations in 1000s of cycles, where all operations are rounded up to 100s of cycles except multiplications, which are rounded up to 1000s.

4.4 Kronecker+ on Co-processors

Having determined how to divide `MatrixVectorMul` up into several subroutines, and how to approximate the costs of those routines by the number of multiplications and additions they require, we can now put those two together. For simplicity we focus on the main arithmetic operation of Saber (with $k = 3$), for which we present the operation counts in Table 4. For example, for $\tau = 1$ a 4097-bit multiplication and addition require 73476 and 781 cycles respectively, which leads to a predicted 142k cycles for `MatrixVectorMul`. This can be compared to the actual value in Table 3, showing it is extremely close. In general the predictions are more accurate for smaller τ, which appears to be due to an underestimation of the cost of the butterfly operations as τ grows (these require quite a few memory operations in the current implementation). However, even for $\tau = 4$ the difference is less than 11% so still allows to give fairly accurate estimations. For completeness we also include the operation counts if one were to do simple Kronecker substitution and applied recursive Karatsuba

		τ				
	op.	0	1	2	3	4
This	#M	9	18	36	72	144
	#A	33	126	327	819	1983
Karatsuba	#M	9	27	81	243	729
	#A	33	129	447	1461	4623

Table 4: Number of multiplications and additions for `MatrixVectorMul` for Saber using Kronecker+ or recursive Karatsuba (applied τ times) with Kronecker substitution. In each column the operations are performed on integers of $\ell n/t+1 = 8192/t+1$ bits, where $t = 2^\tau$.

τ times, showing that Kronecker+ is strongly favored in the context of Saber. Here a single layer of Karatsuba is assumed to split a single multiplication into 3 multiplications and 7 additions/subtractions of half its original size [30, §1.3.2]. Of course, a combination of the two is also possible.

By constructing Table 4, we can now estimate the performance of Kronecker+ on *any* platform by providing the cost of a multiplication and an addition. Let us consider the case of an arithmetic co-processor that contains w-bit multiply-and-accumulate, addition and subtraction (with carry) instructions. As such co-processors are created precisely for the purpose of performing these operations, we can expect them all to require 1 cycle (or a small constant number of cycles). Moreover, as such processors are much simpler devices than a Cortex-M4 (for example) with memory that is directly addressable by the instructions, we do not expect any overhead from loading and storing limbs from and to memory. Finally, let us assume that multiplication is simply implemented via the schoolbook method using $\lceil(\ell n/t+1)/w\rceil^2$ multiply-and-accumulate instructions.

The estimated cycle counts for such a platform are displayed in Table 5. Note that these assumptions do not necessarily hold for each platform: similar tables can easily be generated for a different set of assumptions (e.g., different cycle counts per limb or the ability to use the Karatsuba method). Multiplier lengths are typically powers of two, though might also be somewhat larger to allow techniques such as lazy reduction. For example, the work by Albrecht et al. [1] uses a multiplier slightly larger than 2048 bits. Having an additional bit works especially well with the $(8192/t+1)$-bit operations in Kronecker+, as can be seen in Table 5. The multiplier sizes are inspired by those used for implementing elliptic-curve cryptography or RSA. In general, we see that the cycle counts are significantly lower than 125k reported by Chung et al. [9] and the 317k by Bermudo et al. [3]. We believe this demonstrates the potential of employing a contemporary co-processor for polynomial multiplication. Though this conclusion initially only holds true when using a co-processor in combination with a Cortex-M4 core, the result will only

	τ				
w	0	1	2	3	4
128	40k	24k	16k	13k	14k
129	39k	22k	14k	11k	10k
256	11k	7.3k	5.9k	5.9k	7.2k
257	10k	6.6k	4.9k	4.4k	4.5k
512	3.2k	2.6k	2.5k	3.1k	4.5k
513	2.8k	2.1k	1.9k	1.9k	2.1k
1024	1.0k	1.1k	1.3k	1.9k	–
1025	840	792	798	891	–
2048	390	540	798	–	–
2049	276	324	363	–	–

Table 5: Number of cycles required to execute `MatrixVectorMul` on an arithmetic co-processor with w-bit words and multiply-and-accumulate and addition/subtraction instructions that take 1 cycle each.

be further amplified when pairing a co-processor with a less efficient core.

A few remarks can be made about Table 5. First, these cycle counts *exclude* the `phi_and_shift` and `recover_coeffs` functions. These routines essentially load the polynomials onto the co-processor, and retrieve them at the end of Kronecker+. This cost is fixed across all τ and depends on the communication costs of the co-processor. Especially in the case of large word sizes w, where the number of operations on the co-processor becomes almost negligible, the total cost might be completely determined by communication. This says more about how few operations are necessary on the co-processor rather than how much communication is needed.

Secondly, a dedicated co-processor is separated from the main processor and can therefore perform unrelated operations concurrently. Much of Kronecker+ could therefore be computed while other operations are in progress (e.g., the pseudo-random generation of the matrix), making much of it "free". The lower the latency on the co-processor, the easier it will be to run it in parallel with other operations.

Finally, a polynomial in Kronecker domain consists of at least 1024 bytes, which could pose challenges for smaller platforms. In the simplest case 3 polynomials can be stored simultaneously to allow a straightforward multiply-and-accumulate operation, requiring 3072 bytes of memory. If less memory is available, one can perform time-memory trade-offs to be able to fit the computation onto the device.

5 Conclusions

We introduced a more flexible way of computing polynomial multiplications in the ring $\mathbb{Z}[X]/(X^n+1)$ that can be

combined particularly well with Kronecker substitution and allows for efficient implementation using widely available arithmetic co-processors. This algorithm, which we refer to as Kronecker+, makes use of the available roots of unity by computing a symbolic NTT and can be seen as a variant of the Nussbaumer algorithm, as well as a generalization of Harvey's multipoint Kronecker substitution.

From a theoretical point of view this allows for faster polynomial multiplication in the targeted ring $\mathbb{Z}[X]/(X^n+1)$ on computer architectures with large multipliers. From a practical point of view we outline implementation considerations when contemporary co-processors are put to the task of accelerating post-quantum cryptography. We have demonstrated the potential of Kronecker+ in this setting by implementing the post-quantum finalist scheme Saber.

References

[1] Martin R. Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based schemes using an RSA co-processor. *IACR TCHES*, 2019(1):169–208, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7338.

[2] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016.

[3] 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(2):222–244, Mar. 2020.

[4] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[5] D.J. Bernstein. Multidigit Multiplication For Mathematicians. 1997. https://cr.yp.to/papers/m3.pdf.

[6] Marco Bodrato and Alberto Zanoni. Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices. In Christopher W. Brown, editor, *ISSAC 2007*, pages 17–24. ACM press, July 2007.

[7] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy – SP*, pages 553–570. IEEE Computer Society, 2015.

[8] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *Cryptography and Coding 2013*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2013.

[9] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings. Cryptology ePrint Archive, Report 2020/1397, 2020. https://eprint.iacr.org/2020/1397.

[10] S. A. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966.

[11] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[12] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[13] Espressif Systems. ESP32 Technical Reference Manual. https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[14] Pierrick Gaudry, Alexander Kruppa, and Paul Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC*, 07 2007.

[15] David Harvey. Faster polynomial multiplication via multipoint Kronecker substitution. *Journal of Symbolic Computation*, 44(10):1502–1510, 2009.

[16] Infineon. SLE 78CAFX1M1SPHM. https://www.infineon.com/cms/en/product/security-smart-card-solutions/security-controllers/sle-78/sle-78cafx1m1sphm.

[17] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. https://eprint.iacr.org/2019/844.

[18] Anatoly Karatsuba and Yuri Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics-Doklady 7, 595–596, 1963.

[19] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[20] L. Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal für die reine und angewandte Mathematik*, 92:1–122, 1882.

[21] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[22] Gilad David Maayan. The IoT rundown for 2020: Stats, risks, and solutions. https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx.

[23] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426. Springer, Heidelberg, August 1986.

[24] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[25] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[26] H. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, 1980.

[27] NXP Semiconductors. NXP secure microcontroller SmartMX P71D321. https://www.nxp.com/docs/en/fact-sheet/P71D321.pdf.

[28] John M. Pollard. The fast Fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971.

[29] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[30] Paul Zimmerman Richard Brent. Modern Computer Arithmetic. 2010.

[31] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[32] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[33] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In Jacques Calmet, editor, *Computer Algebra*, pages 3–15. Springer Berlin Heidelberg, 1982.

[34] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[35] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[36] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.1)*, 2020. https://www.sagemath.org.

[37] A.L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[38] G. van der Lubbe. A New Hope for Nussbaumer. 2016. https://www.cs.ru.nl/bachelors-theses/2016/Gerben_van_der_Lubbe___4389026___A_New_Hope_for_Nussbaumer.pdf.

[39] Bin Wang, Xiaozhuo Gu, and Yingshan Yang. Saber on ESP32. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security - ACNS 2020*, volume 12146 of *LNCS*, pages 421–440. Springer, 2020.

[40] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, William Whyte, John M. Schanck, Andreas Hulsing, Joost Rijneveld, Peter Schwabe, Oussama Danba, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRUEncrypt. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.