Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography

Gregor Seiler[*]

IBM Research – Zurich and ETH Zurich
grs@zurich.ibm.com

Abstract. Constant-time polynomial multiplication is one of the most time-consuming operations in many lattice-based cryptographic constructions. For schemes based on the hardness of Ring-LWE in power-of-two cyclotomic fields with completely splitting primes, the AVX2 optimized implementation of the Number-Theoretic Transform (NTT) from the NewHope key-exchange scheme is the state of the art for fast multiplication. It uses floating point vector instructions. We show that by using a modification of the Montgomery reduction algorithm that enables a fast approach with integer instructions, we can improve on the polynomial multiplication speeds of NewHope and Kyber by a factor of 4.2 and 6.3 on Skylake, respectively.

Keywords: lattice cryptography · NTT · implementation · AVX

1 Introduction

Lattice-based cryptography has emerged as a promising candidate for public-key cryptography that is still secure after the likely advent of quantum computers.

From a computational point of view, many lattice-based cryptographic schemes are based on operations in polynomial rings of the form $(\mathbb{Z}/q\mathbb{Z})[X]/(f(x))$ where f is an irreducible polynomial over $\mathbb{Z}$ and q is a prime number. For schemes whose security is based on the (presumed) hardness of the Ring-LWE problem, f is usually chosen to be a power-of-two cyclotomic $X^n + 1$ whose roots have order $2n$ and q is a prime that splits completely (in the number field). The latter condition is equivalent to $q \equiv 1 \pmod{2n}$. The reason for power-of-two cyclotomic rings and fully splitting primes is that the splitting behaviour of such primes in these rings allows for fast multiplication using the Fast Fourier Transform (FFT), which is also called the Number-Theoretic Transform (NTT) if it is performed over the base field $\mathbb{Z}_q$. A full NTT-based multiplication of two polynomials needs two forward NTTs to transform the input polynomials, a cheap pointwise vector multiplication and one inverse NTT. An advantage of NTT-based multiplication over other multiplication methods is that one can often save NTTs by sampling polynomials directly in the NTT domain, by storing the NTT domain representation of polynomials for later use, or by making use of the linearity of the NTT when computing sums of products of polynomials. This leads to important speed-ups as multiplication is frequently the most time consuming single operation in lattice-based schemes.

Producing fast constant-time implementations of the NTT for power-of-two cyclotomic fields has received considerable attention in cryptography during the last couple of years [GOPS13, ADPS16, GS16b, LN16]. For optimal speed on current Intel processors one has to exploit the fact that the NTT is easily vectorizable and needs to make use of

[*]Gregor Seiler was supported by the SNSF ERC Transfer Starting Grant CRETP2-166734-FELICITY and the H2020 Project SAFEcrypto

SIMD instructions. Unfortunately, "optimizing" compilers are not good at automatically producing vectorized code which means one has to manually implement the NTT algorithm using these instructions if one cares about speed.

Central to such an implementation are vectorized constant-time modular operations in the base field $\mathbb{Z}_q$. For high speed one wants to work with densely packed vectors but the intermediate result of a multiplication before modular reduction occupies twice the width. This together with the fact that in floating point arithmetic modular reductions are very easy – multiplying with a precomputed inverse of the modulus q, rounding down to the next integer, multiplying with q and subtracting yields the standard representative – has led some schemes to employ implementations of the NTT that use floating point arithmetic. For instance NewHope [ADPS16] and Ring-TESLA [GS16a].

Floating point arithmetic on current Intel processors is extremely fast. Floating point multiplications are even slightly faster than integer multiplications but additions are slower [Fog17]. In fact, the floating point NTT of NewHope is, to our knowledge, the fastest NTT implementation used in cryptography prior to this work. The authors state that they "experimented with multiple different approaches to speed up the NTT in AVX" and also tried a "vectorized Montgomery arithmetic" integer approach but found that the floating point approach was faster. Longa and Naehrig [LN16] have also tried an integer approach with a novel modular reduction scheme but their AVX2 optimized NTT is still slightly slower than the floating point one from [ADPS16].

Next to the overhead needed for converting between integer and floating point representations, the most important downside of the floating point approach is that by representing the base field elements in 64 bit double values, only four elements fit in an AVX2 register with a width of 256 bits. So especially for moduli that fit into 16 bits, as for instance in the case of NewHope and Kyber [ADPS16, DLL⁺17], this is quite far from densely packed.

In this work we report on the first, to our knowledge, implementation of the NTT for 16 bit primes that uses integer arithmetic on densely packed 16 dimensional vectors and runs in constant time. For multiplications in the base field we use a modification of the original Montgomery multiplication algorithm [Mon85], which allows the integer multiplications to be split into separate short low and high products. See Tables 1 and 2 for the speed of our implementation. On Haswell it is faster by factors of 3.8 and 5.4 compared to the original AVX2 optimized implementations from NewHope and Kyber, respectively. On Skylake it is faster by factors of 4.2 and 6.3.

This work has started with working on an improved optimized integer-arithmetic NTT for the Dilithium signature scheme [DLL⁺17]. We could significantly speed-up the NTT for Dilithium but the major technique presented in this paper is not applicable. The reason is that Dilithium uses a 32 bit prime and the AVX2 instruction set does not offer separate high and low short multiplication instructions for 32 bit integers. Our implementations are now used in Kyber and Dilithium in the versions submitted to the NIST post-quantum cryptography standardization process available from `https://pq-crystals.org`.

1.1 Notation

Let $q \in \mathbb{Z}$ be a prime number. We write $\mathbb{Z}_q$ for the residue class field $\mathbb{Z}/q\mathbb{Z}$ and $\mathbb{Z}_q^\times$ for its unit group. Let $n = 2^e$ be a power of two and q be a prime number such that $q \equiv 1 \pmod{2n}$. Then we denote by $R = \mathbb{Z}[X]/(X^n + 1)$ the ring of integers in the $2n$-th cyclotomic field and by R_q is quotient $R/qR = \mathbb{Z}_q[X]/(X^n + 1)$. In our notation we carefully distinguish between two integers a and b being congruent to each other modulo m, and a being the standard representative of the residue class of b modulo m, i.e. $a - b$ is divisible by m and $0 \leq a < m$. For the former we write $a \equiv b \pmod{m}$ and for the latter $a = b \bmod m$. Sometimes we need the centralized representative $-\frac{m}{2} \leq a < \frac{m}{2}$ of b modulo m and write $a = b \bmod^{\pm} m$.

Table 1: Haswell cycle counts of our AVX2 optimized NTT multiplication implementation and of the original AVX2 optimized implementations from NewHope and Kyber. The counts are the medians of 10000 executions each. NewHope uses the ring $\mathbb{Z}_q[X]/(X^{1024}+1)$ with $q = 2^{13} + 2^{12} + 1$ and Kyber the ring $\mathbb{Z}_q[X]/(X^{256} + 1)$ with $q = 2^{13} - 2^9 + 1$. A full multiplication consists of two forward NTTs, one inverse NTT and the pointwise vector multiplication.

	NewHope		Kyber	
	Original	This work	Original	This work
Forward NTT	9,820	2,784	2,536	460
Inverse NTT	9,780	2,272	2,557	440
Full multiplication	30,408	8,084	7,800	1,432

Table 2: Skylake cycle counts of our AVX2 optimized NTT multiplication implementation and of the original AVX2 optimized implementations from NewHope and Kyber. The counts are the medians of 10000 executions each. NewHope uses the ring $\mathbb{Z}_q[X]/(X^{1024}+1)$ with $q = 2^{13} + 2^{12} + 1$ and Kyber the ring $\mathbb{Z}_q[X]/(X^{256} + 1)$ with $q = 2^{13} - 2^9 + 1$. A full multiplication consists of two forward NTTs, one inverse NTT and the pointwise vector multiplication.

	NewHope		Kyber	
	Original	This work	Original	This work
Forward NTT	9,113	2,433	2,427	419
Inverse NTT	9,037	2,020	2,367	394
Full multiplication	29,583	7,047	8,019	1,278

1.2 NTT-based multiplication

The fact that q is a prime number such that $q \equiv 1 \pmod{2n}$ means that $2n$ divides the order $q-1$ of the cyclic group $\mathbb{Z}_q^\times$. So $\mathbb{Z}_q$ contains $n = \varphi(2n)$ primitive $2n$-th roots of unity ζ^i where $i = 1, 3, \dots, 2n-1$. It follows that $X^n + 1$ factors into linear polynomials $X - \zeta^i$ over $\mathbb{Z}_q$. Now recall that the Chinese remainder theorem says the natural ring homomorphism

$$f \mapsto \left(f(\zeta), f(\zeta^3), \dots, f(\zeta^{2n-1})\right) : \mathbb{Z}_q[X]/(X^n+1) \to \prod_i \mathbb{Z}_q[X]/(X - \zeta^i)$$

is in fact an isomorphism. The NTT computes this isomorphism and we write $\mathrm{NTT} : R_q \to \mathbb{Z}_q^n$ for it. Then the product fg of two polynomials $f, g \in R_q$ can be computed as $fg = \mathrm{NTT}^{-1}(\mathrm{NTT}(f)\,\mathrm{NTT}(g))$ which involves two forward NTTs one inverse NTT and the pointwise multiplication in $\mathbb{Z}_q^n$. Sometimes one can save NTTs. For example,

$$\sum_{i=1}^t f_i g_i = \sum_{i=1}^t \mathrm{NTT}^{-1}(\mathrm{NTT}(f_i)\,\mathrm{NTT}(g_i)) = \mathrm{NTT}^{-1}\left(\sum_{i=1}^t \mathrm{NTT}(f_i)\,\mathrm{NTT}(g_i)\right).$$

Such sums of products of polynomials need to be computed in the matrix vector multiplication of schemes relying on the Module-LWE problem. As an example consider the Dilithium signature scheme with the parameters of the highest security level where the matrix A has dimensions 6×5. Since A is sampled uniformly random, one can save 30 NTTs by sampling directly in the NTT representation. Then the vector needs only transformed once, saving another 25 NTTs, and because of the linearity of the NTT we only need 6 inverse NTTs, which saves 24 inverse NTTs. So instead of the naive 90 (inverse) NTTs one only needs 11; 5 NTTs to transform the vector and 6 inverse NTTs.

1.3 Outline

In Section 2 we recall the standard theory of how the NTT can be computed in a fast way. This is independent of the base field $\mathbb{Z}_q$ and the same algorithm applies to the complex case over $\mathbb{C}$. In Section 3 we turn to the modular arithmetic in $\mathbb{Z}_q$ and present different reduction algorithms used in our NTT. In particular, in this section we present the modified Montgomery reduction algorithm that allows to split multiplications in separate high and low short products. Section 4 contains a description of our implementation strategy and in Section 5 we compare our implementation to the original ones of NewHope and Kyber.

2 The Fast Fourier Transform

We recall the FFT-trick. See the excellent survey [Ber01]. The Fast Fourier Transform is the observation that isomorphisms such as the one above can be computed quickly in a divide and conquer fashion. Concretely, as $X^n + 1 = X^n - \zeta^n$, one can first compute the Chinese remainder map

$$f \mapsto \left(f \bmod X^{n/2} - \zeta^{n/2}, f \bmod X^{n/2} + \zeta^{n/2}\right)$$
$$\mathbb{Z}_q[X]/(X^n - \zeta^n) \to \mathbb{Z}_q[X]/(X^{n/2} - \zeta^{n/2}) \times \mathbb{Z}_q[X]/(X^{n/2} + \zeta^{n/2})$$

and then continue separately in the two rings of polynomials of degree less than $n/2$, noting that $X^{n/2}+\zeta^{n/2} = X^{n/2}-\zeta^{n+n/2}$. This is very effective because computing the remainders of a polynomial f of degree less then n modulo the two polynomials $X^{n/2} \pm \zeta^{n/2}$ only takes $n/2$ multiplications, $n/2$ subtractions and $n/2$ additions in $\mathbb{Z}_q$. More precisely, if c_i and $c_{n/2+i}$ are the i-th and $(n/2 + i)$-th coefficients of f, then the i-th coefficients c'_i and c''_i of the two reduced polynomials are given by $c'_i = c_i + \zeta^{n/2}c_{n/2+i}$ and $c''_i = c_i - \zeta^{n/2}c_{n/2+i}$. Such an operation is called a butterfly and is sometimes given the name Cooley-Tukey butterfly. The FFT can be easily performed in place. After the k-th level, $0 \leq k < \log(n)$, it produces the vector of polynomials with pairs of coefficients

$$f \bmod \left(X^{n/2^{k+1}} - \zeta^{\mathrm{brv}(2^k+i)}\right), f \bmod \left(X^{n/2^{k+1}} + \zeta^{\mathrm{brv}(2^k+i)}\right),$$

in the order $i = 0, \ldots, 2^k - 1$ where brv maps an $\log(n)$-bit number to its bitreversal, $\mathrm{brv}(b_{\log(n)-1}2^{\log(n)-1} + \cdots + b_1 2 + b_0) = b_0 2^{\log(n)-1} + \cdots + b_{\log(n)-2}2 + b_{\log(n)-1}$. So, after all $\log(n)$ levels we get the vector in $\mathbb{Z}_q^n$ with pairs of coefficients

$$f\left(\zeta^{\mathrm{brv}(n/2+i)}\right), f\left(-\zeta^{\mathrm{brv}(n/2+i)}\right)$$

for $i = 0, \ldots, n/2 - 1$. This representation of f is called the CRT representation or the NTT domain representation.

The inverse transform is computed in much the same way by iteratively inverting the CRT maps which in each level needs precisely the same number of multiplications, additions and subtractions as the forward transform. For instance, coming back to the example from above, $c'_i + c''_i = (c_i + \zeta^{n/2}c_{n/2+i}) + (c_i - \zeta^{n/2}c_{n/2+i}) = 2c_i$ and $\zeta^{-n/2}(c'_i - c''_i) = 2c_{n/2+i}$. Here the butterflies differ to those of the forward transform in that addition and subtraction come before multiplying with a root of unity. These butterflies are called Gentleman-Sande butterflies in some works. Dividing out the factors of 2 can be postponed to after the last level by exploiting the linearity of the CRT maps. Then one needs to multiply all coefficients with n^{-1}. Half of these multiplications are best done together with the multiplications with the last root of unity, yielding $n/2$ additional multiplications.

2.1 Twisting

A slight modification of the standard FFT from above is the following. We have the twisting isomorphism

$$X \mapsto \zeta X \colon \mathbb{Z}_q[X]/(X^n + 1) \to \mathbb{Z}_q[X]/(X^n - 1).$$

Then the CRT map followed by twisting the second factor yields the map

$$\mathbb{Z}_q[X]/(X^n-1) \to \mathbb{Z}_q[X]/(X^{\frac{n}{2}}-1)\times\mathbb{Z}_q[X]/(X^{\frac{n}{2}}+1) \to \mathbb{Z}_q[X]/(X^{\frac{n}{2}}-1)\times\mathbb{Z}_q[X]/(X^{\frac{n}{2}}-1)$$

which can be iterated to 1-dimensional factors. This map can be computed by butterflies of the form of the inverse transform, i.e. Gentleman-Sande butterflies. If, in addition, one reorders the coefficients with the bitreversal permutation before and after the transform, then both the forward and the inverse transform can be computed with the same code by only multiplying with different roots. For the small transforms needed in lattice cryptography this reordering has a significant cost. We therefore advise not to do this and instead implement two different algorithms for the forward and inverse transform.

Also, since the constant coefficients do not change under twisting, the number of multiplications needed is the same. But in a vectorized implementation it is difficult to omit multiplying the constant coefficients thus yielding algorithms that perform $n/2$ additional multiplications. Therefore we don't do any twisting in our implementations. Theoretically we think it is the best approach to do twisting during the inverse transform which leads to Cooley-Tukey butterflies in the forward and inverse transform that are a bit more favourable to the modular reduction scheme. Here the additional multiplications are not a concern since they can be hidden with the additional multiplications by n^{-1}. In the NTT from [ADPS16] twisting and reordering is used in the forward transform whereas the NTT from [LN16] uses the standard approach.

We give iterative pseudocode for the NTT in Algorithms 2.1 and 2.1.

Algorithm 1 Forward NTT of a polynomial $f = c_0 + c_1 X \dots c_{n-1}X^{n-1} \in \mathbb{Z}_q[X]/(X^n+1)$ with precomputed roots of unity $\zeta_k = \zeta^{\mathrm{brv}(k)}$, $0 \le k < n$.

```
$k \leftarrow 1$
for $l \leftarrow n/2$; $l > 0$; $l \leftarrow l/2$ do
    for $s \leftarrow 0$; $s < n$; $s \leftarrow j + l$ do
        for $j \leftarrow s$; $j < s + l$; $j \leftarrow j + 1$ do
            $t \leftarrow \zeta_k \cdot c_{j+l}$
            $c_{j+l} \leftarrow c_j - t$
            $c_j \leftarrow c_j + t$
        end for
        $k \leftarrow k + 1$
    end for
end for
```

2.2 Recursive Implementations

The divide and conquer structure means that the FFT can be implemented most easily in a recursive way. Moreover, for large transforms, a recursive implementation quickly reaches polynomials that fit into cache. Therefore, the FFT literature often advises to a recursive approach. For the comparatively small transforms needed in lattice cryptography and today's cache sizes where the full input polynomial usually fits into cache, the favourable data access pattern of a recursive implementation is not an advantage and instead the overhead of the recursion is very significant. Hence for fast FFTs in lattice cryptography

Algorithm 2 Inverse NTT of a polynomial $f = c_0 + c_1X \dots c_{n-1}X^{n-1} \in \mathbb{Z}_q[X]/(X^n+1)$ with precomputed roots of unity $\zeta_k = \zeta^{-(\mathrm{brv}(k)+1)}$, $0 \le k < n$.

```
$k \leftarrow 0$
for $l \leftarrow 1$; $l < n$; $l \leftarrow 2l$ do
    for $s \leftarrow 0$; $s < n$; $s \leftarrow j + l$ do
        for $j \leftarrow s$; $j < s + l$; $j \leftarrow j + 1$ do
            $t \leftarrow c_j$
            $c_j \leftarrow t + c_{j+l}$
            $c_{j+l} \leftarrow t - c_{j+l}$
            $c_{j+l} \leftarrow \zeta_k \cdot c_{j+l}$
        end for
        $k \leftarrow k + 1$
    end for
end for
for $j \leftarrow 0$; $j < n$; $j \leftarrow j + 1$ do
    $c_j \leftarrow c_j/n$
end for
```

written in assembly language we advise against recursive implementations. But note that for a reference implementations in C the better readability of a recursive implementation might be worth considering. Even more so as in our experiments the gcc compiler completely unrolls such implementations yielding even slightly faster transforms than a corresponding iterative implementation in C.

3 Constant-time modular reduction

As we have already hinted at in the introduction, the modular reduction strategy is central for obtaining a fast vectorized NTT implementation. Since we want to construct a vectorized NTT algorithm based on integer arithmetic we will discuss reduction algorithms in integer arithmetic in this section. In particular a modified version of the Montgomery reduction algorithm and a Barrett-style algorithm.

Divisions need to be avoided for fast modular reductions. In principle, Division-less reductions are achieved by multiplying with a precomputed approximate inverse of the modulus to obtain a candidate quotient from which a candidate remainder can be computed with a second multiplication and a subtraction [BZ10]. Contrary to the floating point case, the precomputed inverse is more complicated in integer arithmetic. In the simplest variant it is a fixed point representation of the inverse of the modulus from which the candidate quotient can be obtained by a multiplication and a shift. The lower precision of such a fixed-point reciprocal means that the candidate remainder will in general not be the standard representative. The true remainder then follows after a couple of correction steps. Since the modulus is known at compile time, optimizing compilers generally use such a strategy. The problem for cryptographic applications lies in that if the correction steps are implemented with branching instructions, then the algorithms are not constant time. One can not rely on a compiler to do this in a constant time manner without carefully checking the assembler output of that particular compiler. So this means one must not use the "%" operator of the C programming language for reducing finite field elements that depend on secret data. Instead one has to manually implement a constant time reduction algorithm. One can construct such an algorithm by leaving out the correction steps and only mapping to the standard representative when the result of a field operation would otherwise overflow. For example, to map $0 \le a < 2q$ to its standard representative in constant time one can use the standard trick of first subtracting q and then shifting arithmetically to the right by $l-1$ where l is the width of a. The result is -1, i.e. a bit string with all 1's, if $0 \leq a < q$ and 0 if $q \leq a < 2q$. So by adding the logical AND of this and q we get the standard representative.

3.1 Montgomery reduction

Let $\beta = 2^l$ be the word size needed so that q fits in one word. For instance $\beta = 2^{16}$ if $q < 2^{16}$ and $\beta = 2^{32}$ if $q < 2^{32}$. Let $a \in \mathbb{Z}$ be a double word with $0 \leq a < q\beta$. For example a can be the product of a word x, $0 \leq x < \beta$, and a y with $0 \leq y < q$. We want to reduce a modulo q. Next to the standard or LSB remainder $0 \leq r < q$ defined by $a = mq + r$ for some $m \in \mathbb{Z}$, one can define the so-called Hensel or MSB remainder r' such that $a = mq + r'\beta$ for $0 \leq m < \beta$. The advantage of Hensel remainders is that there is a very fast algorithm [Mon85], known as the Montgomery reduction algorithm, that computes Hensel remainders. A disadvantage is that instead of computing a representative of the residue class of a modulo q, r' is congruent to $a\beta^{-1} \bmod q$. For reductions of products inside the NTT this is not a problem because one has to multiply by the roots of unity which are compile-time constants. So one can just precompute them with an additional factor of $\beta \bmod q$ so that the results after Montgomery reduction are in fact congruent to the desired value a. This was already done in [ADPS16] and we use the same strategy. Contrary to [ADPS16] we also use Montgomery reductions for the pointwise multiplications of the transformed polynomials. As one can not get the Montgomery factor $\beta \bmod q$ in at this point these pointwise products are in fact Hensel remainders. We then make use of the linearity of the NTT and multiply with an additional Montgomery factor together with n^{-1} after the inverse transform.

The Montgomery reduction algorithm that is usually stated in the literature and in Montgomery's original publication [Mon85] is the following. First note that $a \equiv mq \pmod{\beta}$ which implies $m = aq^{-1} \bmod \beta$. So by multiplying a with the inverse of $-q$ modulo β one gets $-m \bmod \beta = \beta - m$, i.e. the negative of the Hensel quotient modulo β. Then $(a + (\beta - m)q)/\beta$ yields $q + r'$. Therefore, Montgomery reduction can be performed by one multiplication modulo β to get $\beta - m$, one full multiplication to get $(\beta - m)q$ and finally an addition and an exact division by β where the latter can be implemented by a right shift. The reason for multiplying with the inverse of $-q$ instead of the inverse of q is that this leads to the non-negative result $q + r'$. Indeed, we have

$$r' = \frac{a - mq}{\beta} > -\frac{\beta q}{\beta} = -q.$$

On the other hand, $r' < q$ and thus $0 < q + r' < 2q$. In order to get the standard representative of $a\beta^{-1}$ modulo q there is at most one correction step necessary.

In the computation of the sum $a + (\beta - m)q$ of two double words the low words cancel since $a + (\beta - m)q = r'\beta$. So it is tempting to compute only the high word of the product $(\beta - m)q$ which is often cheaper than the full product. Unfortunately, we can not rule out the possibility that the low word of a and hence also that of $(\beta - m)q$ is zero so that there is no carry into the high word. Because of this we will use a sightly modified Montgomery reduction algorithm where we multiply by $q^{-1} \bmod \beta$ and subtract mq from a in signed arithmetic. The algorithm can reduce the product of a signed word x, $-\frac{\beta}{2} \leq x < \frac{\beta}{2}$, and a signed word y such that $-q < y < q$. It outputs the Hensel remainder r' defined by $a = mq + r'\beta$ where $-\frac{\beta}{2} \leq m < \frac{\beta}{2}$, which differs from the one above in that m is now the centralized remainder of aq^{-1} modulo β. See Algorithm 3 and Lemma 2. This signed Montgomery reduction algorithm is cheaper than the original one. Instead of a full multiplication, a double word addition and a double word shift we only need a short high product and a one word subtraction. Also notice that the full a is never needed. Instead, for Algorithm 3 it is enough to separately have the high part a_1 and low part a_0 of a. This is very important in our vectorized NTT algorithm.

Definition 1. Let $a \in \mathbb{Z}$ and q be an odd positive integer. The *Hensel remainder* r' of a modulo q with respect to the word size $\beta = 2^l$ such that $q < \frac{\beta}{2}$ is the unique integer r' such that $a = mq + r'\beta$ with $-\frac{\beta}{2} \leq m < \frac{\beta}{2}$.

Lemma 1. *The Hensel remainder r' of $-\frac{\beta}{2}q \leq a < \frac{\beta}{2}q$ modulo q fulfills $-q < r' < q$.*

Proof. From the defining equation,

$$r' = \frac{a - mq}{\beta} < \frac{\beta q + \beta q}{2\beta} = q$$

and similarly $r' > -q$. □

Algorithm 3 Signed Montgomery reduction

```
Require: $0 < q < \frac{\beta}{2}$ odd, $-\frac{\beta}{2}q \leq a = a_1\beta + a_0 < \frac{\beta}{2}q$ where $0 \leq a_0 < \beta$
Ensure: $r' \equiv \beta^{-1}a \pmod{q}$, $-q < r' < q$
1: $m \leftarrow a_0 q^{-1} \bmod {}^{\pm}\beta$          ▷ signed low product, $q^{-1}$ precomputed
2: $t_1 \leftarrow \left\lfloor \frac{mq}{\beta} \right\rfloor$          ▷ signed high product
3: $r' \leftarrow a_1 - t_1$
```

Lemma 2. *Let q be an odd positive integer that fits in one signed word, $0 < q < \frac{\beta}{2}$, and let $-\frac{\beta}{2}q \leq a < \frac{\beta}{2}q$. Algorithm 3 correctly computes the Hensel remainder of a modulo q with respect to the word size β.*

Proof. From the defining equation $a = mq + r'\beta$ it follows that $a \equiv mq \pmod{\beta}$ and $aq^{-1} \equiv m \pmod{\beta}$. By splitting a in its high and low words, a_1 and a_0, respectively, i.e. $a = a_1\beta + a_0$ where $0 \leq a_0 < \beta$, we get $a \equiv a_0 \pmod{\beta}$ and thus $aq^{-1} \equiv a_0q^{-1} \equiv m \pmod{\beta}$. Since, by definition, $-\frac{m}{2} \leq m < \frac{m}{2}$, the signed low product of a_0 and q^{-1} as computed in Line 1 of the algorithm yields m; that is, $a_0q^{-1} \bmod {}^{\pm}\beta = m$. Define $t = t_1\beta + t_0 = mq$ so that t_1 as computed in line 2 is the high word of $t = mq$. Now observe that $a - t = (a_1 - t_1)\beta + (a_0 - t_0) = r'\beta$ which implies $a_0 \equiv t_0 \pmod{\beta}$ and, since $0 \leq a_0, t_0 < \beta$, $a_0 = t_0$. Hence, $a - t = (a_1 - t_1)\beta = r'\beta$ and $a_1 - t_1 = r'$. Last, as both $a, t \geq -\frac{\beta}{2}q = -\frac{q+1}{2}\beta + \frac{\beta}{2}$ and $a, t < \frac{\beta}{2}q = \frac{q-1}{2}\beta + \frac{\beta}{2}$ we have $-\frac{\beta}{4} \leq -\frac{q+1}{2} \leq a_1, t_1 \leq \frac{q-1}{2} < \frac{\beta}{4}$. So, the subtraction $a_1 - t_1$ in line 3 does not overflow when performed as a one word signed subtraction. □

If one wants to compute a standard representative $0 \leq r' < q$ in constant time, then this can easily be done by using a trick along the lines of the one before this section. Concretely, by shifting the output r' of Algorithm 3 arithmetically to the right by $l - 1$, computing the logical AND of the shifted value and q and adding the result to r'. Note that we don't have to first subtract q. This is another advantage of the signed Montgomery reduction. We map to standard representatives in our NTT implementation when q is close to the word size boundary. See Section 3.4 for the details.

3.2 Specialized reduction

We use Montgomery reduction for reducing two word numbers. Especially when performing additions and subtractions we want to stay within one word. For reducing one word integers better methods than Montgomery reduction exist.

When setting parameters for Ring-LWE based schemes, only the size of the prime q and its splitting behaviour in the cyclotomic ring play a role. Therefore, in the case of fully splitting primes, all prime numbers q in a certain interval and such that $q \equiv 1 \pmod{2n}$

can be chosen. It is therefore advisable to choose primes of a form that allows for fast specialized reduction algorithms and sometimes also fast multiplication. For instance, the NewHope key exchange scheme uses the prime $2^{13} + 2^{12} + 1$, Kyber uses $2^{13} - 2^9 + 1$ and the Dilithium signature scheme $2^{23} - 2^{13} + 1$. We exemplify with the Kyber prime $q = 2^{13} - 2^9 + 1$ how one can do specialized reduction with a prime of such a form.

Algorithm 4 Specialized reduction for the Kyber prime $q = 2^{13} - 2^9 + 1$

```
Require: $-2^{15} \leq a < 2^{15}$
Ensure: $r \equiv a \pmod q$, $-2^{15} + 4q \leq r < 2^{15} - 3q$
1: $t \leftarrow \lfloor \frac{a}{2^{13}} \rfloor$        ▷ arithmetic right shift
2: $u \leftarrow a \bmod 2^{13}$                              ▷ logical and
3: $u \leftarrow u - t$
4: $t \leftarrow t \cdot 2^9$                                 ▷ left shift
5: $r \leftarrow u + t$
```

Lemma 3. *If $-2^{15} \leq a < 2^{15}$, then Algorithm 4 computes an integer r congruent to a modulo $q = 2^{13} - 2^9 + 1$ such that $-2^{15} + 4q \leq r < 2^{15} - 3q$.*

Proof. Let $t, u \in \mathbb{Z}$ be the quotient and remainder of a divided by 2^{13}, i.e. $a = t2^{13} + u$ where $0 \leq u < 2^{13}$. Note that t and u are correctly computed in lines 1 and 2, respectively. The fact that $-2^{15} \leq a < 2^{15}$ implies $-4 \leq t < 4$. Now, from $2^{13} \equiv 2^9 - 1 \pmod q$, it follows that $a = t2^{13} + u \equiv t(2^9 - 1) + u = r$. So r as computed in line 5 is indeed a representative of a modulo q. For the bound we find $r = t(2^9 - 1) + u \leq 3 \cdot (2^9 - 1) + 2^{13} - 1 = 2^{15} - 1 - 3q$ and $r \geq -4(2^9 - 1) = -2^{11} + 4 = -2^{15} + 4q$. □

3.3 Barrett reduction

We turn to the one word reduction algorithm that we use for general primes q. In the reference C implementation of NewHope, a short Barrett reduction algorithm is used that employs the precomputed one word approximate reciprocal

$$v = \left\lfloor \frac{\beta}{q} \right\rfloor.$$

With this reciprocal, when reducing unsigned one-word integers, one achieves representatives that are less then $2q$. When the prime is close to $\frac{\beta}{2}$, as is the case for the NewHope prime, we use a higher precision one word reciprocal. Namely,

$$v = \left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1}\beta}{q} \right\rceil.$$

Note that $\frac{2^{\lfloor \log(q) \rfloor - 1}}{q} < \frac{1}{2}$ implies $v < \frac{\beta}{2}$, i.e. v fits into one signed word. The corresponding reduction algorithm is stated in Algorithm 5.

It is easy so see that the t in line 2 can be computed by a signed high product and an arithmetic right shift by $\lfloor \log(q) \rfloor - 1$. For completeness we give the following lemma.

Lemma 4. *Let q be an odd positive integer that fits in one signed word, $0 < q < \frac{\beta}{2}$, and let $-\frac{\beta}{2} \leq a < \frac{\beta}{2}$. Algorithm 5 correctly computes a representative r of a modulo q such that $0 \leq r \leq q$.*

Proof. We separately consider the two cases where a is non-negative and negative, respec-

Algorithm 5 General reduction for signed one word integers

```
Require: $0 \leq q < \frac{\beta}{2}$, $-\frac{\beta}{2} \leq a < \frac{\beta}{2}$
Ensure: $r \equiv a \pmod{q}$ with $0 \leq r \leq q$
1: $v \leftarrow \left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1}\beta}{q} \right\rceil$    ▷ precomputed
2: $t \leftarrow \left\lfloor \frac{av}{2^{\lfloor \log(q) \rfloor - 1}\beta} \right\rfloor$    ▷ signed high product and arithmetic right shift
3: $t \leftarrow tq \bmod \beta$    ▷ signed low product
4: $r \leftarrow a - t$
```

tively. So first assume $a \geq 0$. Then, with $k = \lfloor \log(q) \rfloor - 1$ and modulo β,

$$\begin{aligned} r &\equiv a - \left\lfloor \frac{av}{2^k \beta} \right\rfloor q < a - \frac{av}{2^k\beta} q + q \\ &\leq a - \frac{a 2^k \beta}{2^k \beta q} q + \frac{aq}{2^{\lfloor \log(q) \rfloor}\beta} + q = \frac{aq}{2^{\lfloor \log(q) \rfloor}\beta} + q \\ &< q + 1. \end{aligned}$$

Here we have used that $\frac{q}{2^{\lfloor \log(q) \rfloor}} < 2$ and $a < \frac{\beta}{2}$. On the other hand,

$$a - \left\lfloor \frac{av}{2^k\beta} \right\rfloor q \geq a - \frac{av}{2^k\beta} q \geq -\frac{aq}{2^{\lfloor \log(q) \rfloor}\beta} > -1.$$

We turn to $a < 0$. In this case we have

$$r \equiv a - \left\lfloor \frac{av}{2^k\beta} \right\rfloor q = a + \left\lceil \frac{(-a)v}{2^k\beta} \right\rceil q.$$

Now it follows similar as for the case $a \geq 0$ that the right hand side is bigger than -1 and smaller than $q + 1$. The claim follows by observing that r is integral and $-\frac{\beta}{2} \leq r < \frac{\beta}{2}$. □

A nice feature of this reduction algorithm is that by using the negative $-v$ of the reciprocal and adding t in line 4 one gets a representative between $-q$ and 0. By carefully alternating between these two modes one can arrange for the sum of two reduced elements to stay in $[-q, q]$.

3.4 Lazy reduction

Depending on the size of the modulus and the headroom to the word size boundary, it is not always necessary to reduce the results modulo q when computing additions and subtractions in $\mathbb{Z}_q$.

In the forward NTT, the size of the coefficients of the polynomials grow by less then q from one level to the next. Recall the Cooley-Tukey butterflies $c'_i = c_i + \zeta \cdot c_{i+l}$, $c''_i = c_i - \zeta \cdot c_{i+l}$. So the coefficients only grow by q because the coefficients c'_i, c''_i of the reduced polynomials are the sum or difference of a previous coefficient c_i and a Montgomery reduced product $\zeta \cdot c_{i+l}$ of a coefficient and a root of unity where $-q < \zeta \cdot c_{i+l} < q$. Since the coefficients can grow both towards plus and minus infinity, the width of the interval they are known to lie in grows by at most $2q$. In the special case of the Kyber prime $q = 2^{13} - 2^9 + 1$, where all the values in the interval $[-4q, 4q]$ fit into one signed word of 16 bit, this means that we have to reduce the lower half c_i, $0 \leq i < l$, of the coefficients of each polynomial in every third level. The dimension of the Kyber ring is $n = 256$. In the 8 levels of the forward NTT, next to the Montgomery reductions, we have to do additional reductions in the third and sixth level. We use Algorithm 4 for this task. Then the output polynomials are known to lie in the interval $]-2^{15} + 2q, 2^{15} - q[$. This is a bit too much for the Montgomery reductions in the pointwise multiplications. Therefore, we also reduce in the last level 7. In total we need $3 \cdot 128 = 768$ additional reductions.

This can be improved by mapping the Montgomery reduced values to standard representatives as described at the end of Section 3.1. Then the intervals only grow by q in a known direction. So by strategically adding a suitable multiple of q to the coefficients they can be held in $[-4q, 4q]$ for 7 levels before they need to be reduced. So it is possible to get rid of the additional reductions in the third and sixth level and only reduce in the last level. It turns out that the more complex Montgomery reductions outweigh the savings and we do not do this in the case of the Kyber prime.

For the NewHope prime $q = 2^{13} + 2^{12} + 1$ the situation is different. Here q fits only twice into one signed word. Without mapping the Montgomery reduced products to non-negative values, the coefficients of the reduced polynomials can already be up to $2q$ after the first level. So we have to reduce them in the second level before we can compute the coefficients of the next smaller polynomials. This pattern repeats and we see that we have to reduce in every level. Here it is worth to make the products ζc_{i+l} be standard representatives. Then we can arrange for the coefficients after the zeroth level be in $]-q, q[$, after the first in $]-q, 2q[$, and after the second in $]-2q, 2q[$. Consequently, we only need to reduce in the third and then again in the sixth and ninth level (the NewHope dimension is 1024).

The inverse NTT with its Gentleman-Sande butterflies is a bit less favourable to reductions. In each level coefficients get added or subtracted that can be Montgomery reduced or not reduced in the level before. This makes it more difficult to design a fast reduction scheme for the Gentleman-Sande butterflies. For the Kyber prime we carefully keep track on the size of individual coefficients and only reduce them when they get to large. For the NewHope prime we reduce half of the coefficients in every level, namely those that get not multiplied by roots and Montgomery reduced.

4 AVX2 optimized implementation

We now give details about our AVX2 optimized NTT for 16 bit primes. The AVX2 instruction set offers separate short low and high multiplication instructions for 16 bit packed integers. They are vpmulhuw and vpmulhw for unsigned and signed high multiplications, respectively, and vpmullw for low multiplications. They all have a latency of 5 cycles on both Haswell and Skylake CPUs [Fog17]. So one can compute a full product of 16 integers with 16 bits each in 10 cycles. In contrast the vpmulld instruction computes only 8 such products in also 10 cycles. Moreover, if one only needs the low or high product as is the case in our Montgomery reduction algorithm, then these instructions are 4 times faster then vpmulld.

We use the 16 vector registers in the following way. We reserve 2 registers for the constants q and $q^{-1} \bmod 2^{16}$. Both contain 16 packed copies of these two constants. At all times we have 128 coefficients packed in 8 vector registers. Then we use five registers for temporary results during the computations. Let us exemplify this for a transform of a polynomial $c_0 + c_1 X + \cdots + c_{255} X^{255}$ in dimension $n = 256$. Before the zeroth level we load the coefficients $c_0, \dots, c_{63}$ into four of the coefficient registers and coefficients $c_{128}, \dots, c_{191}$ into the other four. Then by loading 16 copies of the first root into one of the temporary registers we can compute half of the first level. The other half is computed by loading coefficients $c_{64}, \dots, c_{127}$ and $c_{192}, \dots, c_{255}$. After the first level we are left with two polynomials of degrees 127. Each of this fits completely into our 8 coefficient registers. So we load them completely once and transform them to linear factors inside the registers. In each level we have to multiply half of the coefficients; that is, we have to do four parallel vector multiplications and Montgomery reductions. By interleaving these we can considerably hide the latencies of the multiplication instructions. We also interleave the 8 vector additions and subtractions that are needed.

Shuffling. Starting from level 4, the polynomials have degree less then 16. So they only occupy at most one register. We only need to multiply half of the coefficients of each polynomial with a root. Therefore we shuffle the vectors to group together coefficients that need to be multiplied. Concretely, in level 4, we swap the upper 8 coefficients of a register representing one polynomial with the lower 8 coefficients of a register containing another polynomial using the vperm2i128 instructions. Then the second register contains all the high coefficients that need to be multiplied. After the butterflies and at the beginning of level 5 every half register with 8 coefficients represents one polynomial. We swap the upper 4 coefficients in each 128 bit lane of one register with the lower 4 coefficients of another register using the vshufpd instruction. We continue in this way in levels 6 and 7 with vpshufd/vpblendd and vpshufb/vpblendw, respectively.

Precomputed roots. Instead of loading single precomputed roots and populating the vector registers with broadcast and various shuffling instructions, which is slow, we precompute vectors of roots that can be loaded into a vector register with one aligned load instruction vmovdqa only.

5 Comparison

We performed experiments with our optimized NTT multiplication code and compared it to the original AVX2 optimized floating point NTT multiplication code from NewHope and Kyber. Note that Kyber uses the new NTT from this work now. We have taken the NewHope code from `https://cryptojedi.org/crypto/data/newhope-20160815.tar.bz2` and the original Kyber code from `https://github.com/pq-crystals/kyber`. The experiments were conducted on two machines with an Intel Haswell and and Intel Skylake processor, respectively. The Haswell computer is equipped with an Intel Core i7-4770K CPU running at the constant clock frequency of 3500 Mhz. Hyperthreading and Turbo Boost were switched off. The system runs Debian stable with Linux kernel version 3.16.0 and the code was compiled with gcc 6.3.0. The Skylake computer is a laptop with an Intel Core i7-6600U CPU at a clock frequency of 2600 Mhz with Hyperthreading and Turbo Boost off. The system runs Ubuntu with Linux kernel 4.8.0 and the code was compiled with gcc 6.2.0.

Table 1 gives the numbers for Haswell and Table 2 the ones for the Skylake computer. The cycle counts are the medians of 10000 executions each. A full multiplication incorporates two forward NTTs, one inverse NTT and the pointwise multiplication. In the case of the floating point implementations from NewHope and Kyber, the forward NTTs contain the bitreversal permutation. This is different to the numbers in [ADPS16] and [BDK$^+$17]. There the authors give the cycle counts for the forward NTTs without permuting because in applications one can sometimes omit this step when the input polynomial is a random polynomial where all the coefficients are independently identically distributed. So these works show numbers for the forward NTT which are smaller than those for the inverse transform. As we are interested in the general multiplication speeds we decided to measure the times of forward transforms as they are needed in a proper multiplication.

One reason why we measure a higher speedup on Skylake is that floating point additions have a latency of 4 cycles on Skylake compared to only 3 cycles on Haswell. For floating point multiplications Skylake is faster with a latency of 4 cycles compared to 5 cycles. But additions are the bottleneck in the floating point NTTs; see the discussion in [ADPS16].

Interestingly, the more elaborate reduction scheme in our implementation of the forward NTT for the NewHope prime is actually slower. The reason is that the more complex Montgomery reductions make the dependency chains longer for working on the coefficients that need to be multiplied. In contrast, when reducing the other half of the coefficients, the CPU is able to schedule these reductions concurrently with the multiplications and

Montgomery reductions using out of order execution and register renaming. So these additional reductions are very cheap. It is hence easy to improve on our numbers in the case of the NewHope prime.

Other multiplication algorithms Some designs choose rings and moduli for various reasons that do not allow for fast NTT-based multiplication, for example the NTRU KEM from [HRSS17] and NTRU Prime [BCLvV16]. They typically use Toom-Cook and Karatsuba multiplication. In [HRSS17] the modulus 8192 is a power of two that enables extremely simple modular reduction. The authors report that their AVX2 optimized code uses 11722 Haswell cycles to multiply in the 701-dimensional ring $\mathbb{Z}_{8192}[X]/(X^{701}-1)$. Although this ring is much lower dimensional then the 1024-dimensional ring in NewHope, our NTT-based multiplication for NewHope only needs 8084 Haswell cycles. The AVX2 optimized implementation of NTRU Prime need 26682 cycles to multiply in the 761-dimensional field $\mathbb{Z}_{4591}[X]/(X^{761}-X-1)$.

6 Outlook

The upcoming AVX512 instruction set offers twice as many vector registers of twice the width compared to AVX2. So in this instruction set the NTT in a cyclotomic ring of dimension up to 512 and with a prime of 16 bits can be computed completely inside the registers by using our dense approach. It will be interesting to see the multiplication performance of these processors.

7 Bibliography

References

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 327–343, 2016.

[BCLvV16] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. Ntru prime: reducing attack surface at low cost. Cryptology ePrint Archive, Report 2016/461, 2016. https://eprint.iacr.org/2016/461.

[BDK$^+$17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. Crystals – kyber: a cca-secure module-lattice-based kem. Cryptology ePrint Archive, Report 2017/634, 2017. https://eprint.iacr.org/2017/634.

[Ber01] Daniel J. Bernstein. Multidigit multiplication for mathematicians, 2001.

[BZ10] Richard P. Brent and Paul Zimmermann. Modern computer arithmetic (version 0.5.1). *CoRR*, abs/1004.4710, 2010.

[DLL$^+$17] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. Crystals – dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Report 2017/633, 2017. https://eprint.iacr.org/2017/633.

[Fog17] Agner Fog. Instruction tables, 2017.

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In *Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings*, pages 67–82, 2013.

[GS16a] Shay Gueron and Fabian Schlieker. Optimized ring-tesla, 2016.

[GS16b] Shay Gueron and Fabian Schlieker. Speeding up R-LWE post-quantum key exchange. In *Secure IT Systems - 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016, Proceedings*, pages 187–198, 2016.

[HRSS17] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. High-speed key encapsulation from NTRU. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 232–252, 2017.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, pages 124–139, 2016.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.