Implementing number theoretic transforms*†

JORIS VAN DER HOEVEN[a], GRÉGOIRE LECERF[b]

CNRS (UMR 7161, LIX)
Laboratoire d'informatique de l'École polytechnique
Bâtiment Alan Turing, CS35003
1, rue Honoré d'Estienne d'Orves
91120 Palaiseau, France

a. *Email:* `vdhoeven@lix.polytechnique.fr`
b. *Email:* `lecerf@lix.polytechnique.fr`

Preliminary version of December 16, 2024

We describe a new, highly optimized implementation of number theoretic transforms on processors with SIMD support (AVX, AVX-512, and Neon). For any prime modulus p and any order of the form $r = 2^i \cdot 3^j | p - 1$, our implementation can automatically generate a dedicated codelet to compute the number theoretic transform of order r over $\mathbb{F}_p$. New speed-ups were achieved by relying heavily on non-normalized modular arithmetic and allowing for orders r that are not necessarily powers of two.

KEYWORDS: number theoretic transform, finite fields, codelets, algorithm, complexity bound, FFT, integer multiplication

A.C.M. SUBJECT CLASSIFICATION: G.1.0 Computer-arithmetic, F.2.1 Number-theoretic computations

A.M.S. SUBJECT CLASSIFICATION: 68W30, 68Q17, 68W40

1. INTRODUCTION

Number theoretic transforms (NTTs) were introduced by Pollard [21]. He used them as a tool to design practical algorithms for the efficient multiplication of very large integers. Technically speaking, a number theoretic transform is simply a discrete Fourier transform (DFT) over a finite field. Traditional DFTs work over the complex numbers. A fast algorithm to compute such DFTs was published a few years earlier by Cooley and Tukey [4], although the idea goes back to Gauss [14].

As of today, fast practical algorithms for multiplying large integers are based on number theoretic transforms, complex DFTs [23, first algorithm], or Schönhage-Strassen's algorithm [23, second algorithm]. All three strategies are very competitive and the winner depends on hardware and the application. Since the race is very close, it is interesting to heavily optimize each of the three approaches.

*. Grégoire Lecerf has been supported by the French ANR-22-CE48-0016 NODE project. Joris van der Hoeven has been supported by an ERC-2023-ADG grant for the ODELIX project (number 101142171).

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

†. This article has been written using GNU TeXmacs [16].

In this paper, we will describe a high performance implementation for number theoretic transforms. We focus on modern general purpose CPUs with support for SIMD (Single Instruction Multiple Data) instructions. Our current implementation was written in MATHEMAGIX [17], partly as a test case for the development of our new compiler (a β-version of it has been released this year). It fully exploits SIMD accelerations and we started experimentations with multi-threading.

A first way to optimize NTTs is to design faster modular arithmetic. Let p be a prime number. Traditionally, the result of any arithmetic operation modulo p is always normalized, e.g. to make it lie in $\{0,\ldots,p-1\}$. It was first observed by Harvey [13] that some of these normalizations can be delayed when computing NTTs and that this can lead to significant speed-ups. In section 3, we show how to take this idea one step further, which allows us to delay even more normalizations.

While we were working on our implementation[1], a few other implementations appeared that use similar ideas. The delayed reduction strategy has been generalized to 4-wide and 8-wide butterflies in [2, 24]. Dan Schultz recently added an optimized implementation of this kind to the FLINT library [12]. We note that the idea to delay normalizations is also very popular in linear algebra; see for instance [5, 6]. Another inspiration and noteworthy analogue is the "transient ball arithmetic" that was developed in [18].

The optimization of traditional DFTs is the subject of an extensive literature. The most efficient implementations are based on so-called "codelets" [8, 9, 22]. For a given target order r, the idea is to automatically generate a highly optimized program (called a *codelet*) for DFTs of order r. Such codelets can optimize the use of registers and memory, which makes them in particular cache-efficient. For small orders, we can also unroll the loops, which minimizes the overhead of control structures. In section 4, we apply the codelet approach to number theoretic transforms and discuss some specific issues that arise for this application. Similar experiments have recently been carried out in the SPIRAL system [10, 25].

The last section 5 is dedicated to timings. We implemented our algorithms on five different systems that support SIMD arithmetic of three different widths: 128-bit (ARM Neon), 256-bit (AVX-2), and 512-bit (AVX-512). For orders $r \leqslant 2^{16}$, we achieve speed-ups that are proportional to the SIMD width. For higher orders, the x86-based systems start to suffer from memory access costs, whereas the performance of ARM-based systems remains stable. We also experimented with multi-threaded versions of our algorithms, but without pushing our optimization efforts as far as for the mono-threaded case.

The application of the fast NTTs to integer multiplication requires a wrapper for Chinese remaindering and Kronecker segmentation. This turns out to be highly non-trivial to implement with the same level of optimization as our new NTTs. We plan to address this issue in a separate future work.

2. PRELIMINARIES

2.1. Discrete Fourier transforms

Let $\mathbb{K}$ be a field with a primitive root of unity ω of order r and assume that r is invertible in $\mathbb{K}$. We define the *discrete Fourier transform* (or DFT) of an r-tuple $a = (a_0, \ldots, a_{r-1}) \in \mathbb{K}^r$

1. We started our implementation in 2019, but it remained private due to the experimental nature of the new MATHEMAGIX compiler.

with respect to ω to be $\mathrm{DFT}_\omega(a) = \hat{a} = (\hat{a}_0, \ldots, \hat{a}_{r-1}) \in \mathbb{K}^r$ where

$$\hat{a}_i \;:=\; a_0 + a_1 \omega^i + \cdots + a_{r-1} \omega^{(r-1)i}.$$

That is, $\hat{a}_i$ is the evaluation of the polynomial $A(X) := a_0 + a_1 X + \cdots + a_{r-1} X^{r-1}$ at ω^i. If ω is a primitive r-th root of unity, then so is its inverse $\omega^{-1} = \omega^{r-1}$, and we have

$$\mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(a)) \;=\; r a. \tag{1}$$

Indeed, writing $b := \mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(a))$, we have

$$b_i \;=\; \sum_{j=0}^{r-1} \hat{a}_j \omega^{-ji} \;=\; \sum_{j=0}^{r-1} \sum_{k=0}^{r-1} a_k \omega^{j(k-i)} \;=\; \sum_{k=0}^{r-1} a_k \sum_{j=0}^{r-1} \omega^{j(k-i)} \;=\; \sum_{k=0}^{r-1} a_k (r \delta_{i,k}) \;=\; r a_i,$$

where $\delta_{i,k} = 1$ if $i = k$ and $\delta_{i,k} = 0$ otherwise.

The DFT induces a ring homomorphism from $\mathbb{K}[x] / (x^r - 1)$ into $\mathbb{K}^r$, by sending $A = a_0 + \cdots + a_{r-1} x^{r-1}$ to $\mathrm{DFT}_\omega(a)$. Since the DFT is invertible, this map (that we will still denote by DFT_ω) is actually an isomorphism. If we have a fast way to compute DFTs, this yields an efficient way to multiply "cyclic" polynomials $P, Q \in \mathbb{K}[x] / (x^r - 1)$:

$$P Q \;=\; \mathrm{DFT}_\omega^{-1}(\mathrm{DFT}_\omega(P)\, \mathrm{DFT}_\omega(Q)). \tag{2}$$

Here the multiplication $\mathrm{DFT}_\omega(P)\, \mathrm{DFT}_\omega(Q)$ in $\mathbb{K}^r$ is done entry-wise. Given two polynomials $A, B \in \mathbb{K}[x]$ with $\deg A B < r$, we have

$$A B \bmod (x^r - 1) \;=\; (A \bmod (x^r - 1))\, (B \bmod (x^r - 1))$$

and the product $A B$ can be read off from $A B \bmod (x^r - 1)$. Hence we also obtained an algorithm for the multiplication of A and B. This is called *FFT-multiplication*.

2.2. Divide-and-conquer DFTs

Let ω be a primitive r-th root of unity, with $r = r_1 r_2$ for $1 < r_1, r_2 < r$. Then $\omega_2 := \omega^{r_1}$ is a primitive r_2-th root of unity and $\omega_1 := \omega^{r_2}$ is a primitive r_1-th root of unity. Moreover, for any $i_1 \in \{0, \ldots, r_1 - 1\}$ and $i_2 \in \{0, \ldots, r_2 - 1\}$, we have

$$\begin{aligned}
\hat{a}_{i_1 r_2 + i_2} \;&=\; \sum_{k_1=0}^{r_1-1} \sum_{k_2=0}^{r_2-1} a_{k_2 r_1 + k_1} \omega^{(k_2 r_1 + k_1)(i_1 r_2 + i_2)} \\
&=\; \sum_{k_1=0}^{r_1-1} \omega^{k_1 i_2} \left(\sum_{k_2=0}^{r_2-1} a_{k_2 r_1 + k_1} (\omega^{r_1})^{k_2 i_2} \right) (\omega^{r_2})^{k_1 i_1}.
\end{aligned} \tag{3}$$

This formula can be decomposed in four steps:

$$x_{i_2 r_1 + k_1} \;:=\; \sum_{k_2=0}^{r_2-1} a_{k_2 r_1 + k_1} \omega_2^{k_2 i_2} \tag{4}$$

$$y_{i_2 r_1 + k_1} \;:=\; \omega^{k_1 i_2} x_{i_2 r_1 + k_1} \tag{5}$$

$$z_{i_2 r_1 + i_1} \;:=\; \sum_{k_1=0}^{r_1-1} y_{i_2 r_1 + k_1} \omega_1^{k_1 i_1} \tag{6}$$

$$\hat{a}_{i_1 r_2 + i_2} \;:=\; z_{i_2 r_1 + i_1}. \tag{7}$$

Given a linear map $\phi\colon \mathbb{K}^s \to \mathbb{K}^s$ and $t \in \mathbb{N}$, it will be convenient to use the Kronecker products $\phi \otimes \mathrm{Id}_t\colon \mathbb{K}^{st} \to \mathbb{K}^{st}$ and $\mathrm{Id}_t \otimes \phi\colon \mathbb{K}^{st} \to \mathbb{K}^{st}$, that are defined by

$$\begin{aligned} y &= (\phi \otimes \mathrm{Id}_t)(x) \iff \forall i \in \{0,\ldots,t-1\},\ (y_i,\ldots,y_{(s-1)t+i}) = \varphi(x_i,\ldots,x_{(s-1)t+i}) \\ y &= (\mathrm{Id}_t \otimes \phi)(x) \iff \forall i \in \{0,\ldots,t-1\},\ (y_{is},\ldots,y_{is+s-1}) = \varphi(x_{is},\ldots,x_{is+s-1}). \end{aligned}$$

Using these notations, we may rewrite (4) and (6) as

$$\begin{aligned} x &= (\mathrm{DFT}_{\omega_2} \otimes \mathrm{Id}_{r_1})(a) \\ z &= (\mathrm{Id}_{r_2} \otimes \mathrm{DFT}_{\omega_1})(y). \end{aligned}$$

Step (5) consists of r scalar multiplications by so-called *twiddle factors* $\omega^{k_1 i_2}$. We will denote this *twiddling map* by $\Omega_{r_1,r_2,\omega}$:

$$y = \Omega_{r_1,r_2,\omega}(x).$$

The last step corresponds to the transposition of an $r_2 \times r_1$ matrix. Denoting this transposition map by Π_{r_2,r_1}, the relation (3) becomes

$$\Pi_{r_1,r_2} \circ \mathrm{DFT}_\omega = (\mathrm{Id}_{r_2} \otimes \mathrm{DFT}_{\omega_1}) \circ \Omega_{r_1,r_2,\omega} \circ (\mathrm{DFT}_{\omega_2} \otimes \mathrm{Id}_{r_1}). \tag{8}$$

2.3. DFTs using the Chinese remainder theorem

If $\gcd(r_1, r_2) = 1$ in the previous subsection, then it is actually possible to avoid the twiddling step (5), as first noted by Good in [11]. Consider the Bezout relation

$$u_1 r_1 + u_2 r_2 = 1,$$

where $|u_1| < r_2$ and $|u_2| < r_1$. Then we have the isomorphism of abelian groups

$$\begin{aligned} \mathbb{Z}/r\mathbb{Z} &\longrightarrow \mathbb{Z}/r_1\mathbb{Z} \times \mathbb{Z}/r_2\mathbb{Z} \\ i \bmod r &\longmapsto (u_2 i \bmod r_1, u_1 i \bmod r_2) \\ (r_2 i_1 + r_1 i_2) \bmod r &\longleftarrow (i_1 \bmod r_1, i_2 \bmod r_2) \end{aligned}$$

This Chinese remainder isomorphism induces an isomorphism of $\mathbb{K}$-algebras

$$\begin{aligned} \mathbb{K}[x]/(x^r - 1) &\overset{\xi}{\longrightarrow} \mathbb{K}[x_1]/(x_1^{r_1} - 1) \otimes \mathbb{K}[x_2]/(x_2^{r_2} - 1) \\ x^i &\longmapsto x_1^{u_2 i} x_2^{u_1 i} \\ x^{r_2 i_1 + r_1 i_2} &\longleftarrow x_1^{i_1} x_2^{i_2}. \end{aligned}$$

Representing cyclic polynomials $A = \sum_i a_i x^i \in \mathbb{K}[x]/(x^r - 1)$ and

$$B = \sum_{0 \leqslant i_1 < r_1, 0 \leqslant i_2 < r_2} b_{i_1 r_2 + i_2} x_1^{i_1} x_2^{i_2} \in \mathbb{K}[x_1]/(x_1^{r_1} - 1) \otimes \mathbb{K}[x_2]/(x_2^{r_2} - 1)$$

by the vectors $a, b \in \mathbb{K}^r$ of their coefficients, the map ξ acts as a permutation $\Xi\colon \mathbb{K}^r \to \mathbb{K}^r$: if $B = \xi(A)$ then $b_{i_1 r_2 + i_2} = a_{(r_2 i_1 + r_1 i_2) \operatorname{rem} r}$, for $0 \leqslant i_1 < r_1$ and $0 \leqslant i_2 < r_2$. The bivariate DFT

$$\mathrm{DFT}_{\omega_1,\omega_2} := \mathrm{DFT}_{\omega_1} \otimes \mathrm{DFT}_{\omega_2} := (\mathrm{Id}_{r_1} \otimes \mathrm{DFT}_{\omega_2}) \circ (\mathrm{DFT}_{\omega_1} \otimes \mathrm{Id}_{r_2})$$

sends $B = \sum_{i_1,i_2} b_{i_1 r_2 + i_2} x_1^{i_1} x_2^{i_2}$ to the vector $\hat{b}$ with $\hat{b}_{i_1 r_2 + i_2} = B(\omega_1^{i_1}, \omega_2^{i_2})$. If $B = \xi(A)$ and $\hat{a} = \mathrm{DFT}_\omega(A)$, then

$$B(\omega_1^{i_1}, \omega_2^{i_2}) = B(\omega^{i_1 r_2}, \omega^{i_2 r_1}) = A(\omega^{i_1 r_2 u_2 + i_2 r_1 u_1}),$$

so $\hat{b}_{i_1 r_2 + i_2} = \hat{a}_{(i_1 r_2 u_2 + i_2 r_1 u_1) \operatorname{rem} r}$ and $\hat{b} = \Pi(\hat{a})$ for some permutation $\Pi: \mathbb{K}^r \to \mathbb{K}^r$. Altogether, this yields

$$\Pi \circ \mathrm{DFT}_\omega \;=\; \mathrm{DFT}_{\omega_1, \omega_2} \circ \Xi.$$

2.4. DFTs up to permutations

The above two subsections show that efficient algorithms for DFTs often only compute them up to permutations. For some applications, it is actually not required to perform all these permutations. Assume for instance that we have an efficient algorithm for the computation of the "twisted" $\widetilde{\mathrm{DFT}}_\omega = \Pi \circ \mathrm{DFT}_\omega$ for some permutation Π. Then the variant

$$P Q \;=\; \widetilde{\mathrm{DFT}}_\omega^{-1}(\widetilde{\mathrm{DFT}}_\omega(P)\, \widetilde{\mathrm{DFT}}_\omega(Q))$$

of (2) still yields an efficient method for FFT-multiplication. This technique requires the inverse transform $\widetilde{\mathrm{DFT}}_\omega^{-1}$ to be implemented with more care, since one cannot directly use the formula (1). Instead, one typically uses a similar algorithm as for DFT_ω, but with all steps reversed.

The twisting technique can be combined recursively with the algorithms from the previous two subsections. Assume for instance that $\widetilde{\mathrm{DFT}}_{\omega_1} := \Pi_1 \circ \mathrm{DFT}_{\omega_1}$ and $\widetilde{\mathrm{DFT}}_{\omega_2} := \Pi_2 \circ \mathrm{DFT}_{\omega_2}$ for some permutations $\Pi_1: \mathbb{K}^{r_1} \to \mathbb{K}^{r_1}$ and $\Pi_2: \mathbb{K}^{r_2} \to \mathbb{K}^{r_2}$, and let

$$\widetilde{\mathrm{DFT}}_\omega \;:=\; (\Pi_1 \otimes \Pi_2) \circ \Pi_{r_1, r_2} \circ \mathrm{DFT}_\omega,$$

with the notation of (8). Then, equation (8) implies that

$$\begin{aligned}
\widetilde{\mathrm{DFT}}_\omega &= (\mathrm{Id}_{r_2} \otimes \Pi_1) \circ (\Pi_2 \otimes \mathrm{Id}_{r_1}) \circ (\mathrm{Id}_{r_2} \otimes \mathrm{DFT}_{\omega_1}) \circ \Omega_{r_1, r_2, \omega} \circ (\mathrm{DFT}_{\omega_2} \otimes \mathrm{Id}_{r_1}) \\
&= (\mathrm{Id}_{r_2} \otimes \Pi_1) \circ (\mathrm{Id}_{r_2} \otimes \mathrm{DFT}_{\omega_1}) \circ (\Pi_2 \otimes \mathrm{Id}_{r_1}) \circ \Omega_{r_1, r_2, \omega} \circ (\mathrm{DFT}_{\omega_2} \otimes \mathrm{Id}_{r_1}) \\
&= (\mathrm{Id}_{r_2} \otimes \Pi_1) \circ (\mathrm{Id}_{r_2} \otimes \mathrm{DFT}_{\omega_1}) \circ \tilde{\Omega}_{r_1, r_2, \omega} \circ (\Pi_2 \otimes \mathrm{Id}_{r_1}) \circ (\mathrm{DFT}_{\omega_2} \otimes \mathrm{Id}_{r_1}) \\
&= (\mathrm{Id}_{r_2} \otimes \widetilde{\mathrm{DFT}}_{\omega_1}) \circ \tilde{\Omega}_{r_1, r_2, \omega} \circ (\widetilde{\mathrm{DFT}}_{\omega_2} \otimes \mathrm{Id}_{r_1})
\end{aligned}$$

where $\tilde{\Omega}_{r_1, r_2, \omega} := (\Pi_2 \otimes \mathrm{Id}_{r_1}) \circ \Omega_{r_1, r_2, \omega} \circ (\Pi_2^{-1} \otimes \mathrm{Id}_{r_1})$. For some table $(\omega^{e_0}, \ldots, \omega^{e_{r-1}})$ of precomputable twiddling factors, we have $y_i = \omega^{e_i} x_i$ for any $x, y \in \mathbb{K}^r$ with $y = \tilde{\Omega}_{r_1, r_2, \omega}(x)$.

For DFTs of small orders, we also note that permutations of the input and output coefficients can be achieved without cost. As will be explained in sections 4.1 and 4.2, we will regard such DFTs as completely unrolled *straight-line programs* (SLPs); see definition in [3] for instance. Instead of explicitly permuting the input and output coefficients, it then suffices to appropriately rename all local variables in the SLP. This is particularly useful for the algorithm from section 2.3, which requires permutations of both the input and output coefficients.

2.5. Number theoretic transforms

In the special case when $\mathbb{K}$ is a finite field $\mathbb{F}_p$ for some prime number p, a discrete Fourier transform is also called a *number theoretic transform* (NTT). The most favorable case is when p is of the form $p = s 2^\ell + 1$ or $p = s 2^\ell 3^{\ell'} + 1$, where s is small. Indeed, since the multiplicative group of $\mathbb{F}_p$ is cyclic of order $p - 1$, this ensures the existence of primitive roots of unity ω of large orders of the form 2^ℓ or $2^\ell 3^{\ell'}$. Nice primes p like this are sometimes called *FFT primes*. In what follows, for any order $r \mid (p - 1)$, we will denote by NTT_r a number theoretic transform of the form $\Pi \circ \mathrm{DFT}_\omega$ for some suitable permutation Π and primitive r-th root of unity ω.

Pollard first noted [21] that NTTs can be used to devise efficient practical algorithms for multiplying large integers: using Kronecker segmentation, the two multiplicands $a, b \in \mathbb{N}$ can be rewritten as special values

$$\begin{aligned} a &= a_0 + a_1 2^k + \cdots + a_m 2^{km} \\ b &= b_0 + b_1 2^k + \cdots + b_n 2^{kn} \end{aligned}$$

of integer polynomials $A, B \in \mathbb{N}[x]$ with small coefficients $a_i, b_j \in \{0, \ldots, 2^c - 1\}$. Then $ab = (AB)(2^k)$ and the product AB has coefficients in $\{0, \ldots, (\max(m, n) + 1) 2^{2c} - 1\}$. If

$$(\max(m, n) + 1) 2^{2c} < 2^k, \tag{9}$$

then AB can be read off from its reduction modulo p. If we also have $m + n < r$, then the product AB modulo p can be computed fast using FFT-multiplication.

For the application to integer multiplication, Pollard's technique is most efficient when p nicely fits into a machine integer or floating point number. As soon as $\max(m, n)$ gets large (say $\max(m, n) > 2^{20}$), this constraints c to become fairly small (e.g. $c < 15$ when using double precision floating point arithmetic with $k = 50$). In order to increase c, and reduce the negative impact of $\max(m, n)$ in the constraint (9), Pollard suggested to reduce modulo three FFT-primes p_1, p_2, and p_3 instead of a single one and use Chinese remaindering to work modulo $p_1 p_2 p_3$ instead of p. For instance, when using double precision floating point arithmetic, this allows us to take $k = 150$ and $c < 65$ instead of $k = 50$ and $c < 15$. As of today, this is one of the fastest practical methods for integer multiplication [13].

3. MODULAR FLOATING POINT ARITHMETIC

We refer to [7, 19] for previous work on the implementation of SIMD versions of modular arithmetic. Before starting such an implementation, one first has to decide whether one wishes to rely on floating point or integer arithmetic. In principle, integer arithmetic should be most suitable, since no bits are wasted on the storage of exponents. However, for large moduli of more than 32 bits, floating point arithmetic tends to be supported better by current hardware.

First of all, at least in the case of INTEL processors, throughputs and latencies for basic floating point instructions are better than for their integer counterparts. Secondly, the AVX2 and AVX-512 instruction sets only provide SIMD support for 32×32 bit integer products; although there is an SIMD instruction for 64×64 bit integer multiplication, the high part of the 128 bit result is lost. In comparison, with floating point arithmetic, the full product of a 53×53 bit integer multiplication can be obtained using two fused multiply add instructions.

Based on these hardware considerations, we have decided to implement our NTT with floating point arithmetic. Our implementation focusses on odd moduli $m \geqslant 3$ that fit into double precision numbers and works for SIMD widths up to 8, as allowed by processors with AVX-512 support.

Our approach further extends known ideas. On the one hand, for the normalized representation of integers modulo m, we use signed residues in $\{-\lfloor m/2 \rfloor, \ldots, \lfloor m/2 \rfloor\}$; this provides us with a cheap way to gain one bit of precision and it also simplifies the mathematical analysis. On the other hand, we will allow the results of certain intermediate computations to be non-normalized. The latter idea was already used with success in [13], but we push it further by taking the modulus a bit smaller (thereby sacrificing a few bits of precision), which allows us to redundantly represent modular integers $a \bmod m$ by integers a that can be several times larger than m (this allows for several speed-ups, as we shall see). Several recent implementations [2, 24] also use this kind of super-redundant representations in order to perform basic NTTs of order 4 and 8 without unnecessary reductions. In this paper we will present a generalization to arbitrary orders.

3.1. Pseudo-code

In what follows, we will write μ for the machine precision and assume that $\mu \geqslant 8$. Then any real number in the interval $[1, 2)$ is a floating point number if and only if it is a multiple of $2^{1-\mu}$. Floating point number can be scaled by powers 2^e with exponents $e \in [-E+2, E-1]$. Here $E \in \mathbb{N}$ is assumed to satisfy $E-2 \geqslant 2\mu$. Finally, zero is a special floating point number.

When rounding to nearest, the rounding error of a floating point operation with an exact result in $[1,2)$ is always bounded by $2^{-\mu}$. Given a general operation with an exact result $x \in [1,2)$ and rounded result $\tilde{x}$, this means that $|\tilde{x}-x| \leqslant 2^{-\mu}$. Note also that $\tilde{x} \geqslant 1$ necessarily holds in this case. Consequently, we have

$$|\tilde{x}-x| \leqslant |x| 2^{-\mu} \quad \text{and} \quad |\tilde{x}-x| \leqslant |\tilde{x}| 2^{-\mu}. \tag{10}$$

More generally, these two inequalities hold whenever the exponent of x does not overflow or underflow. Our current implementation is limited to double precision, for which $\mu = 53$ and $E = 1024$, but it could easily be extended to single and other precisions. We assume correct IEEE 754 style rounding to the nearest [1].

We will present our implementation of modular arithmetic using pseudo machine code. The only data type that we will use are SIMD vectors of width w with floating point entries (e.g. $w = 8$ for processors with AVX-512 support, for double precision). Such vectors are typically stored in hardware registers. The first argument of all our algorithms will be the destination register and the subsequent arguments the source registers. For instance, the instruction `add`(a, b, c) adds two SIMD vectors b and c and puts the result in a. In the pseudo-code, the modulus m stands for a SIMD vector of w potentially different moduli. We also denote by u its floating point inverse $1/m$, rounded to nearest.

We assume that basic instructions for SIMD floating point arithmetic are provided by the hardware, namely addition (`add`), subtraction (`sub`), multiplication (`mul`), fused multiply add (`fma`) and its variants (`fms`, `fnma`, `fnms`), as well as an instruction for rounding to the nearest integer (`round`). Precisely, the fused operators are defined as follows:

`fma`(d,a,b,c) means $d := ab + c$		`fnma`(d,a,b,c) means $d := -ab + c$	
`fms`(d,a,b,c) means $d := ab - c$		`fnms`(d,a,b,c) means $d := -ab - c$	

On processors with AVX-512 support, one may implement round using the intrinsic _mm512_roundscale_pd.

The modular analogues for the basic arithmetic operations are suffixed by _mod. For instance, mul_mod(d,a,b) computes w products modulo m. The results of these analogues are not required to be normalized. The modular ring operations, as well as a separate operation normalize for normalization, can be implemented as follows:

Algorithm add_mod(d,a,b)

```
add(d,a,b)
```

Algorithm sub_mod(d,a,b)

```
sub(d,a,b)
```

Algorithm normalize(d,a)

```
mul(x,a,u)
round(q,x)
fnma(d,q,m,a)
```

Algorithm mul_mod(d,a,b)

```
mul(x,a,b)
mul(q,x,u)
round(q,q)
fms(y,a,b,x)
fnma(z,q,m,x)
add(d,z,y)
```

Here x,y,z are temporary variables. Modular analogues of the fused operations fma, fms, fnma, and fnms turn out to be of limited use, since non-normalized addition and subtraction take only one operation.

3.2. Measuring how far we are off from normalization

For the analysis of our algorithms, it suffices to consider the scalar case $w=1$. Consider a modular integer $a \bmod m$ represented by an integer $a \in \mathbb{Z}$ that fits into a floating point number of precision μ. Let

$$\nu_a := \frac{2|a|}{m}$$

and note that $\nu_a < 1$ if and only if $|a| \leqslant \lfloor m/2 \rfloor$, i.e. if and only if a is normalized. If $\nu_a \geqslant 1$, then we regard ν_a as a measure for how non-normalized a really is. For each of our algorithms add_mod, sub_mod, mul_mod, and normalize it is interesting to analyze ν_d as a function of ν_a and ν_b.

Note that $2^\mu + 1$ is the smallest integer that cannot be represented exactly using a floating point number. For this reason, we always assume that the source arguments a and b of our algorithms as well as the modulus m satisfy $|a| \leqslant 2^\mu$, $|b| \leqslant 2^\mu$, and $m \leqslant 2^\mu$. Each of our algorithms is easily seen to be correct, provided that $|d| \leqslant 2^\mu$. In what follows, we will in particular study under which circumstances this can be guaranteed.

In the cases of addition and subtraction, the analysis is simple, since we clearly have $\nu_d \leqslant \nu_a + \nu_b$ and $|d| \leqslant 2^\mu$, provided that $|a| + |b| \leqslant 2^\mu$ (or $\nu_a + \nu_b \leqslant 2^{\mu+1}/m$, equivalently).

PROPOSITION 1. *The algorithm* normalize *satisfies:*

a) If $|a| < 2^{\mu-2}$, then $\nu_d < 1$.

b) If $2^{\mu-2} \leqslant |a| \leqslant 2^\mu$, then $\nu_d \leqslant 1 + {}^5/_m$.

Proof. Let ϵ and δ be the errors for the computations of u and x:

$$\begin{aligned} m^{-1} &= u+\epsilon \\ x &= a\,u+\delta \end{aligned}$$

and recall that $|\epsilon|\leqslant u\,2^{-\mu}$, $|\delta|\leqslant|x|\,2^{-\mu}$, $|\epsilon|\leqslant m^{-1}\,2^{-\mu}$, and $|\delta|\leqslant|a|\,u\,2^{-\mu}$, using (10). Hence,

$$|x-a\,m^{-1}| \;\leqslant\; |a\,u-a\,m^{-1}|+|\delta| \;\leqslant\; |a|\,|\epsilon|+|\delta| \;\leqslant\; |a|\,u\,2^{1-\mu}.$$

Let η be the distance between $a\,m^{-1}$ and the set $\mathbb{Z}+{}^1\!/_2$. Since m is odd, we must have $\eta\geqslant(2\,m)^{-1}$. If $|x-a\,m^{-1}|<\eta$, then we necessarily have $q=\lfloor x\rceil=\lfloor a\,m^{-1}\rceil$, and the end-result d will indeed be normalized. Here $\lfloor x\rceil$ stands for the nearest integer to x. The condition $|x-a\,m^{-1}|<\eta$ is satisfied as soon as $|a|\,u\,2^{1-\mu}<(2\,m)^{-1}$, i.e. if $|a|\,(1+2^{-\mu})<2^{\mu-2}$. Since a is an integer and there are no integers between $2^{\mu-2}/(1+2^{-\mu})$ and $2^{\mu-2}$, this condition further simplifies into $|a|<2^{\mu-2}$. This proves (a).

If $2^{\mu-2}\leqslant|a|\leqslant2^{\mu}$, then the above calculations still yield $|x-a\,m^{-1}|\leqslant2\,u$, whence

$$|q-a\,m^{-1}| \;\leqslant\; |q-x|+|x-a\,m^{-1}| \;\leqslant\; {}^1\!/_2+2\,u$$

and

$$|d| \;=\; |m\,q-a| \;\leqslant\; {}^m\!/_2+2\,u\,m \;\leqslant\; {}^m\!/_2+2\,(1+m\,|\epsilon|) \;\leqslant\; {}^m\!/_2+2(1+2^{-\mu}).$$

This in particular implies (b). □

Remark 2. If $2^{\mu-2}\leqslant|a|\leqslant2^{\mu}$ and $m<2^{\mu-1}-5$, then $|d|=\nu_d\,m/2\leqslant(m+5)/2<2^{\mu-2}$, so a second application of `normalize` will achieve to normalize a.

PROPOSITION 3. *Let*

$$\begin{aligned} C &:= (1+2^{2-\mu})\,m\,2^{-\mu} && \approx\; m\,2^{-\mu} \\ B &:= 1-(m+9)\,2^{-1-\mu} && \approx\; 1-\tfrac{1}{2}m\,2^{-\mu} \\ \gamma &:= \frac{1+\sqrt{1-m\,(1+2^{2-\mu})\,2^{2-\mu}}}{4\,(1+2^{2-\mu})} && \approx\; \tfrac{1}{2}-\tfrac{1}{2}m\,2^{-\mu}-\tfrac{1}{2}\,(m\,2^{-\mu})^2-\cdots. \end{aligned}$$

Then the algorithm `mul_mod` *satisfies:*

a) We have $\nu_d\leqslant1+C\,\nu_a\,\nu_b$.

b) If $|a\,b|\leqslant\frac{1}{2}B\,2^{2\mu}$, *then* $|d|\leqslant2^{\mu}$.

c) If $m\leqslant2^{\mu-2}$, $|a|\leqslant\gamma\,2^{\mu}$, *and* $|b|\leqslant\gamma\,2^{\mu}$, *then* $|d|\leqslant\gamma\,2^{\mu}$.

Proof. We first note that the product $a\,b$ is represented exactly by $a\,b=x+y$, where $|y|\leqslant|x|\,2^{-\mu}$ and $|y|\leqslant|a\,b|\,2^{-\mu}$. By a similar reasoning as in the beginning of the proof of Proposition 1, we have

$$|q-x\,m^{-1}| \;\leqslant\; {}^1\!/_2+|x|\,u\,2^{1-\mu},$$

whence

$$|z| \;\leqslant\; {}^m\!/_2+|x|\,(1+2^{-\mu})\,2^{1-\mu}.$$

Note that $\mu \geqslant 8$ implies $|z| < 2^{\mu}$ so z is the exact value. We deduce that

$$\begin{aligned} |d| &\leqslant |z| + |y| \\ &\leqslant {}^{m}\!/_{2} + |x|\,(1+2^{1-\mu})\,2^{1-\mu} + |a\,b|\,2^{-\mu} \\ &\leqslant {}^{m}\!/_{2} + |a\,b|\,(1+2^{-\mu})\,(1+2^{1-\mu})\,2^{1-\mu} + |a\,b|\,2^{-\mu} \\ &\leqslant {}^{m}\!/_{2} + |a\,b|\,(1+2^{2-\mu})\,2^{1-\mu}. \end{aligned}$$

This relation in particular yields (a). If $|a\,b| \leqslant \frac{1}{2} B\,2^{2\mu}$, then we also obtain

$$\begin{aligned} {}^{m}\!/_{2} + |a\,b|\,(1+2^{2-\mu})\,2^{1-\mu} &\leqslant {}^{m}\!/_{2} + (1+2^{2-\mu})\,B\,2^{\mu} \\ &\leqslant {}^{m}\!/_{2} + (1+2^{2-\mu})\,(1-(m+9)\,2^{-1-\mu})\,2^{\mu} \\ &\leqslant {}^{m}\!/_{2} + (1-m\,2^{-1-\mu})\,2^{\mu} \\ &= 2^{\mu}, \end{aligned}$$

which proves (b). Assume finally that $|a| \leqslant \gamma\,2^{\mu}$ and $|b| \leqslant \gamma\,2^{\mu}$. Then

$$|d| \leqslant {}^{m}\!/_{2} + 2\,\gamma^{2}\,(1+2^{2-\mu})\,2^{\mu}.$$

Since $m \leqslant 2^{\mu-2}$ is odd, we have $m \leqslant 2^{\mu-2} - 1 < 2^{\mu-2}/(1+2^{2-\mu})$, whence the equation

$$2\,t^{2}\,(1+2^{2-\mu})\,2^{\mu} - t\,2^{\mu} + {}^{m}\!/_{2} = 0$$

has two positive roots

$$t_{1,2} = \frac{1 \pm \sqrt{1 - m\,(1+2^{2-\mu})\,2^{2-\mu}}}{4\,(1+2^{2-\mu})}.$$

We took γ to be the largest of these two roots. This implies (c). □

Remark 4. If $|a| \leqslant 2^{\mu-2}$, $|b| \leqslant 2^{\mu-2}$, and $m < 2^{\mu-2}$, then (c) implies $|d| \leqslant 2^{\mu-2}$. Indeed, $m \leqslant 2^{\mu-2} - 1$ in that case, whence $\gamma \geqslant {}^{1}\!/_{4}$.

Remark 5. Multiplication has the benefit of producing partially normalized results, especially if one of the arguments is already normalized. For instance, if $\nu_a \leqslant 1$ and $|b| \leqslant 2^{\mu}$, then (a) yields

$$\nu_d \leqslant 1 + C\,\nu_b \leqslant 1 + C\,2^{\mu+1}/m \leqslant 3 + 2^{3-\mu}.$$

If $m < 2^{\mu-4}$, $\nu_a \leqslant 1$, and $\nu_b \leqslant 4$, then (a) yields

$$\nu_d \leqslant 1 + C\,\nu_a\,\nu_b \leqslant 1 + 4\,(1+2^{2-\mu})\,(2^{\mu-4}-1)\,2^{-\mu} \leqslant 1 + {}^{1}\!/_{4},$$

since $\mu \geqslant 8$.

4. CODELETS FOR NUMBER THEORETIC TRANSFORMS

The easiest way to implement DFTs is to write a single routine that takes the order r and the coefficients as input and then applies, say, Cooley and Tukey's algorithm using a double loop. However, this strategy is register-inefficient for small orders and cache-inefficient for large orders. Suboptimal loop unrolling may be another problem.

For our implementation, we use the alternative strategy based on "codelets". For a given finite field $\mathbb{F}_p$ and a given target order r, the idea is to automatically generate a highly optimized program NTT_r for number theoretic transforms of order r over $\mathbb{F}_p$ (up to permutations as explained in section 2.4). Such highly optimized programs that are dedicated to one specific task are called *codelets*. We refer to [8, 9, 22] for early work on codelets in the more traditional context of DFTs over $\mathbb{C}$.

For small orders, codelets are typically designed by hand. For large orders, they are generated automatically in terms of codelets of smaller lengths. We developed a small "codelet library" inside MATHEMAGIX for the automatic generation of codelets for NTTs of different orders as well as other linear maps. One difference with previous work is that all code is written in the MATHEMAGIX language and that codelets can be both generated and executed at runtime.

There are often multiple solutions to a given task. For instance, for an NTT of composite order r, there may be multiple ways to factor $r = r_1 r_2$ and then apply one of the algorithms from section 2.2 or 2.3. One advantage of codelets is that we may generate codelets for each solution, do some benchmarking at runtime, and then select the most efficient solution. For large orders, when benchmarking becomes more expensive, the most efficient tactics can be cached on disk. Our codelet library exploits this kind of ideas.

4.1. Straight-line programs over modular integers

A straight-line program (SLP) over $\mathbb{Z}/m\mathbb{Z}$ is a map $f: (\mathbb{Z}/m\mathbb{Z})^k \rightarrow (\mathbb{Z}/m\mathbb{Z})^\ell$ that can be implemented using a sequence of ring operations in $\mathbb{Z}/m\mathbb{Z}$. For instance, if $\mathrm{i} \in \mathbb{Z}/m\mathbb{Z}$ is a primitive root of unity of order 4, then the following SLP with input $s = (s_0, s_1, s_2, s_3)$ and output $d = (d_0, d_1, d_2, d_3)$ computes an NTT of order 4:

$x_0 := s_0 + s_2$
$x_1 := s_1 + s_3$
$x_2 := s_0 - s_2$
$x_3 := s_1 - s_3$
$x_3 := \mathrm{i} \times x_3$
$d_0 := x_0 + x_1$
$d_1 := x_0 - x_1$
$d_2 := x_2 + x_3$
$d_3 := x_2 - x_3$

Here x_0, x_1, x_2, x_3 are auxiliary variables. Note also that the output has been re-indexed in bit-reversed order: if $\hat{s} = \mathrm{DFT}_{\mathrm{i}}(s)$, then $d_0 = \hat{s}_0$, $d_1 = \hat{s}_2$, $d_2 = \hat{s}_1$, and $d_3 = \hat{s}_3$.

For an actual machine implementation of such an SLP, we may replace the ring operations by the algorithms `add_mod`, `sum_mod`, and `mul_mod` from the previous section. However, we have to be careful that all non-normalized intermediate results fit into μ bit integers. An easy linear-time greedy algorithm to ensure this is to insert `normalize` instructions whenever some intermediate result might not fit.

More precisely, for every intermediate result z of a binary operation on a and b, we use the results from section 3.2 to compute a bound for ν_z. Whenever this bound exceeds $2^{\mu+1}/m$, we insert an instruction to normalize a or b (we pick a if $\nu_a \geqslant \nu_b$ and b otherwise).

For instance, if $2^{\mu+1}/m$ is slightly larger than $3\,{}^1\!/_2$, then applying this strategy to the above SLP yields

`add_mod`(x_0,s_0,s_2)	$\nu_{x_0} \leqslant 2$	
`add_mod`(x_1,s_1,s_3)	$\nu_{x_1} \leqslant 2$	
`sub_mod`(x_2,s_0,s_2)	$\nu_{x_2} \leqslant 2$	
`sub_mod`(x_3,s_1,s_3)	$\nu_{x_3} \leqslant 2$	
`mul_mod`(x_3,i,x_3)	$\nu_{x_3} \leqslant 2\,{}^1\!/_7$	(using part (a) of Proposition 3)
`normalize`(x_0,x_0)	$\nu_{x_0} \leqslant 1^+$	(using Proposition 1)
`add_mod`(d_0,x_0,x_1)	$\nu_{x_0} \leqslant 3^+$	
`sub_mod`(d_1,x_0,x_1)	$\nu_{x_2} \leqslant 3^+$	
`normalize`(x_3,x_3)	$\nu_{x_3} \leqslant 1^+$	(using Proposition 1)
`add_mod`(d_2,x_2,x_3)	$\nu_{x_2} \leqslant 3^+$	
`sub_mod`(d_3,x_2,x_3)	$\nu_{x_3} \leqslant 3^+$	

Here we implicitly assumed that the input arguments s_0, s_1, s_2, s_3 are normalized. We also wrote 1^+ for a number that is slightly larger than 1 (e.g. $1+2^{5-\mu}$ will do). We indeed had to normalize x_0 or x_1 before the instruction `add_mod`(d_0,x_0,x_1), since otherwise $\nu_{x_0}+\nu_{x_1}=4$ would exceed $3\,{}^1\!/_2$. Similarly, we normalized x_3, since $2+2\,{}^1\!/_7=4\,{}^1\!/_7>3\,{}^1\!/_2$. On the other hand, after these normalizations, we did not need to normalize x_1 and x_2, since $2+1^+ =: 3^+ < 3\,{}^1\!/_2$.

A few remarks about this approach.

- The greedy strategy is not optimal in the sense that we might have prevented overflows by inserting an even smaller number of normalization instructions. One obvious optimization is to do the normalization of x_0 right after its computation, which may improve subsequent bounds and avoid other normalizations. One may iterate the greedy algorithm a few times after applying this optimization. It is an interesting question how to design even better algorithms.

- We may choose to normalize the output d_0, d_1, d_2, d_3 or not. If we allow for non-normalized output, then it may be interesting to also allow for non-normalized input. The greedy algorithm generalizes to this case as long as we know bounds for $\nu_{s_0}, \nu_{s_1}, \nu_{s_2}, \nu_{s_3}$.

- The bound for ν_{x_3} after multiplication with i is suboptimal: since i is a constant, ν_{i} is also a known constant in $(0,1)$. We may use this constant instead of the rough bound $\nu_{\mathrm{i}} \leqslant 1$. For instance, if $\nu_{\mathrm{i}} \leqslant {}^1\!/_4$, then $\nu_{x_3} \leqslant 1\,{}^2\!/_7$ after the multiplication of x_3 with i. Consequently, $\nu_{x_2}+\nu_{x_3}=3\,{}^2\!/_7<3\,{}^1\!/_2$, which makes it no longer necessary to normalize x_3.

4.2. Number theoretic transforms of small orders

For NTTs of order up to 64, it is efficient to use unrolled codelets that can be regarded as SLPs. Throughout our implementation, we assumed that $2^{\mu}/m \geqslant 31^+$. This allows for moduli $m > 2^{48}$ with a “capacity” of 48 bits. We wrote our transforms by hand and manually determined the best spots for normalizations. Although this means that we did apply the general greedy algorithm from the previous section, we did use the same strategy to determine bounds for the ν_z, where z ranges over the intermediate results, and verify that we always have $\nu_z \leqslant 31$.

For small power of two orders $r \in \{2,4,8\}$, we use the classical Cooley–Tukey algorithm without any normalization. Writing s and d for the input and output vectors in $(\mathbb{Z}/m\mathbb{Z})^r$, and setting $\nu_s := \max(\nu_{s_0}, \ldots, \nu_{s_{r-1}})$, Proposition 3 implies $\nu_d \leqslant r\,\nu_s$. In view of Remark 5, the worst case $\nu_d = r\,\nu_s$ may only occur for evaluations at 1 and -1 (which only involve additions and subtractions).

NTTs of medium-size power of two orders $r \in \{16,32,64\}$ are decomposed into smaller NTTs with one intermediate twiddling stage: writing NTT_r for a number theoretic transform of order r (while assuming the usual bit-reverse indexing), we apply the formulas

$$\begin{aligned}\mathrm{NTT}_{16} &= (\mathrm{Id}_4 \otimes \mathrm{NTT}_4) \circ \mathrm{Twiddle}_{4,4} \circ (\mathrm{NTT}_4 \otimes \mathrm{Id}_4)\\ \mathrm{NTT}_{32} &= (\mathrm{Id}_4 \otimes \mathrm{NTT}_8) \circ \mathrm{Twiddle}_{4,8} \circ (\mathrm{NTT}_4 \otimes \mathrm{Id}_8)\\ \mathrm{NTT}_{64} &= (\mathrm{Id}_8 \otimes \mathrm{NTT}_8) \circ \mathrm{Twiddle}_{8,8} \circ (\mathrm{NTT}_8 \otimes \mathrm{Id}_8),\end{aligned}$$

for suitable diagonal linear twiddle maps $\mathrm{Twiddle}_{r_1,r_2}$. Our implementation automatically generates completely unrolled codelets for NTT_{16}, NTT_{32}, and NTT_{64}. The twiddling step has the advantage of partially normalizing the intermediate results (see Remark 5). For instance, if $x = (\mathrm{NTT}_8 \otimes \mathrm{Id}_8)(s)$, $y = \mathrm{Twiddle}_{8,8}(x)$, $d = (\mathrm{Id}_8 \otimes \mathrm{NTT}_8)(y)$, and $\nu_s \leqslant 3$, then $\nu_x \leqslant 24$, $\nu_y \leqslant 1\,{}^{24}\!/_{31} \leqslant 1\,{}^{7}\!/_{8}$, and $\nu_d \leqslant 15$, using part (a) of Proposition 3. Furthermore, the fact that several twiddle factors are equal to one allows for an optimization: whenever we need to multiply c with one, we simply normalize c instead. In the case of NTT_{16}, the error bounds are a bit better, and we only need to perform two normalizations.

We also implemented hand-coded NTTs for some mixed-radix orders of the form $r = 2^i 3^j$. If $r = 3$, and assuming that ω is a primitive third root of unity, we use the following algorithm, without normalization:

$d_1 := s_1 - s_2$
$d_0 := \omega \times d_1$
$d_2 := s_0 - s_1$
$d_2 := d_2 - d_0$
$d_1 := s_0 - s_2$
$d_1 := d_1 + d_0$
$d_0 := s_0 + s_1$
$d_0 := d_0 + s_2$

One verifies that $\nu_{d_0} \leqslant 3\,\nu_s$ and $\nu_{d_1} \leqslant 3\,(1 + {}^{2}\!/_{93})\,\nu_s$. For $r \in \{6, 12\}$, we use Good's algorithm to decompose NTT_r in terms of NTT_3 and $\mathrm{NTT}_{r/3}$. This yields $\nu_d \leqslant r\,(1 + {}^{2}\!/_{93})\,\nu_s$. For $r = 9$, we use the formula

$$\mathrm{NTT}_9 = (\mathrm{Id}_3 \otimes \mathrm{NTT}_3) \circ \mathrm{Twiddle}_{3,3} \circ (\mathrm{NTT}_3 \otimes \mathrm{Id}_3)$$

without normalization, which yields $\nu_d \leqslant 9\,(1 + {}^{2}\!/_{93})\,\nu_s$. For $r \in \{18, 24\}$, we use Good's algorithm with an intermediate normalization step. For $r = 48$, we use the formula

$$\mathrm{NTT}_{48} = (\mathrm{Id}_6 \otimes \mathrm{NTT}_8) \circ \mathrm{Twiddle}_{6,8} \circ (\mathrm{NTT}_6 \otimes \mathrm{Id}_8),$$

while taking benefit of the partial normalization of the twiddling step.

4.3. Large orders

Now assume that we wish to compute an NTT of large order $r \in 2^{\mathbb{N}} 3^{\mathbb{N}}$ with $r > 64$. For input and output vectors $s, d \in (\mathbb{Z}/r\mathbb{Z})^r$, we require that $\nu_d \leqslant 24$ whenever $\nu_s \leqslant 2$. This is indeed the case for all codelets from the previous subsection.

Let $r = r_1 r_2$ be a decomposition of r with $1 < r_1, r_2 < r$. We recursively assume that we know how to compute codelets for NTT_{r_1} and NTT_{r_2}. Now we compute NTT_r using the formula

$$\mathrm{NTT}_r \;=\; (\mathrm{Id}_{r_1} \otimes \mathrm{NTT}_{r_2}) \circ \mathrm{Twiddle}_{r_1,r_2} \circ (\mathrm{NTT}_{r_1} \otimes \mathrm{Id}_{r_2}). \tag{11}$$

Following (4), we compute $x = (\mathrm{NTT}_{r_1} \otimes \mathrm{Id}_{r_2})(s)$ using

$$(x_j, x_{r_2+j}, \ldots, x_{r-r_2+j}) \;:=\; \mathrm{NTT}_{r_1}(s_j, s_{r_2+j}, \ldots, s_{r-r_2+j}) \tag{12}$$

for $j = 0, \ldots, r_2 - 1$. Note that this yields $\nu_x \leqslant 24$ whenever $\nu_s \leqslant 2$, by our assumption on NTT_{r_1}. Setting $y := \mathrm{Twiddle}_{r_1,r_2}(x)$, let $\omega_{i,j}$ be the twiddle factors with

$$y_{ir_2+j} \;=\; \omega_{i,j} x_{ir_2+j},$$

for $i = 0, \ldots, r_1 - 1$ and $j = 0, \ldots, r_2 - 1$. For $i = 0, \ldots, r_1 - 1$, we next compute

$$(d_{ir_2}, d_{ir_2+1}, \ldots, d_{ir_2+r_2-1}) \;:=\; \mathrm{NTT}_{r_2}(\omega_{i,0} x_{ir_2}, \omega_{i,1} x_{ir_2+1}, \ldots, \omega_{i,r_2-1} x_{ir_2+r_2-1}), \tag{13}$$

which corresponds to combining (5) and (6). If $\nu_x \leqslant 24$, then $\nu_y \leqslant 1\,{}^{30}\!/_{31} \leqslant 2$ and $\nu_d \leqslant 24$, by our assumption on NTT_{r_2}. This proves our inductive requirement that $\nu_d \leqslant 24$ whenever $\nu_s \leqslant 2$.

It is important to keep an eye on the cache efficiency for the actual implementation. In the formula (12), we precompute the twiddle factors and store the vectors $(d_{ir_2}, \ldots, d_{ir_2+r_2-1})$, $(x_{ir_2}, \ldots, x_{ir_2+r_2-1})$, and $(\omega_{i,0}, \ldots, \omega_{i,r_2-1})$ in contiguous segments of memory. Instead of applying a global twiddling step $\mathrm{Twiddle}_{r_1,r_2}$, we use a loop over i and mix the twiddling step with the NTTs of length r_2; this avoids traversing r elements in memory twice and thereby improves the cache efficiency. If r_2 is really large, then we might rewrite NTT_{r_2} in terms of smaller NTTs and move the twiddling step even further inside.

As to (12), we first need to move the input slices $(s_j, s_{r_2+j}, \ldots, s_{r-r_2+j})$ with "stride" r_2 into contiguous memory. After applying the NTT, we also need to put the result back in the slice $(x_j, x_{r_2+j}, \ldots, x_{r-r_2+j})$ with stride r_2. These memory rearrangements correspond to two $r_1 \times r_2$ and $r_2 \times r_1$ matrix transpositions. For cache efficiency, we need to minimize the number of full traversals of vectors of size r, so it is again better to use a loop over j. However, entries of the form $s_{2^\ell k}, \ldots, s_{2^\ell k + 2^\ell - 1}$ are typically stored in the same cache line for some $\ell \in \mathbb{N}$ that depends on the hardware. It is better to treat these contiguous entries together, which can be achieved by cutting the loop over j into chunks of size 2^ℓ (in fact, slightly different chunk sizes may perform even better). This corresponds to rewriting the $r_1 \times r_2$ matrix transposition into $r_2/2^\ell$ transpositions of $r_1 \times 2^\ell$ matrices (and similar for the inverse transposition).

4.4. SIMD acceleration

Assume now that we are on hardware that supports SIMD vectors of width w and let us investigate how to vectorize the algorithm (11) from the previous subsection.

Let us first consider the case when $w^2 | r$ and $r_2 := w$. The first step (12) is easy to vectorize: we may reinterpret the map $\mathrm{NTT}_{r_1} \otimes \mathrm{Id}_{r_2}$ on vectors of size r with entries in $\mathbb{Z}/m\mathbb{Z}$ as the map $\mathrm{NTT}_{r_1} \otimes \mathrm{Id}_{r_2/w}$ on vectors of size r/w with SIMD entries in $(\mathbb{Z}/m\mathbb{Z})^w$. As to the second step (13), we have $\mathrm{Id}_{r_1} \otimes \mathrm{NTT}_w = \mathrm{Id}_{r_1/w} \otimes \mathrm{Id}_w \otimes \mathrm{NTT}_w$ and the computation of the map $\mathrm{Id}_w \otimes \mathrm{NTT}_w$ reduces to the computation of $\mathrm{NTT}_w \otimes \mathrm{Id}_w$, using a $w \times w$ matrix transposition. In order to compute the map $(\mathrm{Id}_{r_1} \otimes \mathrm{NTT}_w) \circ \mathrm{Twiddle}_{r_1,w}$, this leads us to use a loop of length r_1/w that treats blocks of w^2 coefficients at the time. On each block, we first multiply with the twiddle factors, then apply a $w \times w$ matrix transposition, and finally an inline codelet for NTT_w for SIMD coefficients of width w.

The idea behind SIMD implementations of $w \times w$ matrix transposition is to recursively reduce this operation to two $(w/2) \times (w/2)$ transpositions in different lanes, followed by a linear number of operations to combine the results. This leads to an algorithm that does the full transposition in $O(w \log_2 w)$ steps. Concrete implementations on existing hardware are a bit tricky: every next generation or brand of SIMD processors (SSE, AVX2, AVX-512, Neon, …) tends to provide another set of instructions that are useful for this application (`permute`, `blend`, `shuffle`, etc.). Instead of covering all cases, we illustrate the above recursive meta-algorithm by showing how to perform a 4×4 matrix transposition using 8 SIMD instructions on machines with AVX2 support:

```
x[0] = _mm256_permute2f128_pd (s[0], s[2], 32)
x[2] = _mm256_permute2f128_pd (s[0], s[2], 49)
x[1] = _mm256_permute2f128_pd (s[1], s[3], 32)
x[3] = _mm256_permute2f128_pd (s[1], s[3], 49)
d[0] = _mm256_shuffle_pd (x[0], x[1], 0)
d[1] = _mm256_shuffle_pd (x[0], x[1], 15)
d[2] = _mm256_shuffle_pd (x[2], x[3], 0)
d[3] = _mm256_shuffle_pd (x[2], x[3], 15)
```

Let us now turn to the case when $w^3 | r$. For any decomposition $r = r_1 r_2$ with $w | r_1$ and $w^2 | r_2$, we may apply the above vectorization technique for the computation of DFT_{r_2}. Using the loop (13), this also allows us to vectorize $(\mathrm{Id}_{r_1} \otimes \mathrm{NTT}_{r_2}) \circ \mathrm{Twiddle}_{r_1,r_2}$. Combined with the fact that the map $\mathrm{NTT}_{r_1} \otimes \mathrm{Id}_{r_2}$ trivially vectorizes, this provides us with an alternative vectorization of the formula (11). This technique has the advantage of providing more flexibility for the choices of r_1 and r_2. In particular, for large orders r, taking $r_1 \approx r_2 \approx \sqrt{r}$ tends to ensure a better cache efficiency.

For our actual implementation, we emulate an SIMD width $w' := 2w$ of twice the hardware SIMD width w for the given floating point type. This means that every instruction is "duplexed" for the "low" and "high" parts. For instance, the codelet for NTT_3 from section 4.2 gives rise to the following duplex version for coefficients in $(\mathbb{Z}/m\mathbb{Z})^2$:

$$d_2 := s_2 - s_4$$
$$d_3 := s_3 - s_5$$
$$d_0 := \omega \times d_2$$
$$d_1 := \omega \times d_3$$

name	CPU	SIMD	cores	frequency	memory
i7	Intel Core i7	4×64	4 physical, 8 logical	2.8 GHz	16 Gb 1.6 GHz DDR3
xeon	Intel Xeon W	8×64	8 physical, 16 logical	3.2 GHz	32 Gb 2.67 GHz DDR4
zen4	AMD 7950X3D	8×64	16 physical, 32 logical	5.759 GHz	128 Gb, 5.2GHz DDR5
m1	Apple M1	2×64	4 performance, 4 efficiency	3.2 GHz	16 Gb
m3	Apple M3 Pro	2×64	6 performance, 6 efficiency	4.05 GHz	36 Gb

Table 1. Overview of the five machines on which we benchmarked our implementation.

$$
\begin{aligned}
&d_4 := s_0 - s_2\\
&d_5 := s_1 - s_3\\
&d_4 := d_4 - d_0\\
&d_5 := d_5 - d_1\\
&\vdots
\end{aligned}
$$

The main advantage of this technique is that it spreads data dependencies. This makes the code less vulnerable to delays caused by instructions with high latencies. One disadvantage of duplexing is that it doubles the pressure on hardware registers. Nevertheless, this is less of a problem on modern processors with many SIMD registers.

5. TIMINGS

We implemented the algorithms described in this paper in the MATHEMAGIX language. In order to test the new algorithms, one has to add `--enable-mmcomp` to the configuration options, before compiling MATHEMAGIX. Here `mmcomp` stands for the new experimental MATHEMAGIX compiler. The code for the number theoretic transforms can be found in the `mmx/hpc` subdirectory of the `mmlib` package (revision 11190 available at https://sourcesup.renater.fr/projects/mmx/). The binary that we used to obtain the timings in Table 2 was compiled using the following command:

```
mmcomp --optimize hpc/bench/dft_best_mixed_bench.mmx
```

We systematically use the fixed prime number $p = 1439\cdot 2^{28}\cdot 3^6+1 = 281597114843137$ as our modulus for all NTTs. Consequently, $\mathbb{F}_p$ has power of two roots of unity of orders up till $r = 2^{28}$, which is also an upper bound for our main benchmarks. For orders $r \leqslant 2^{25}$, our implementation does an extensive search for the best strategy. For larger orders, we make a suboptimal guess based on the best observed strategies for orders $\leqslant 2^{25}$.

We tested our implementation on five different systems whose characteristics are specified in Table 1. These systems cover three SIMD widths:

- 128 bits, using ARM Neon on the Apple M1 and M3 based machines;
- 256 bits, on an old laptop whose Intel Core i7 4980HQ CPU supports AVX-2;
- 512 bits, on two machines with AVX-512 support: Intel Xeon W-2140B and AMD RYZEN9 7950X3D.

Note that the AVX-512 support on the AMD RYZEN9 7950X3D processor is “double pumped” in the sense that certain instructions (like FMA) are done in two parts of 256 bits each.

r	i7	xeon	zen4	m1	m3	r	i7	xeon	zen4	m1	m3
64	0.10µs	51.4ns	16.5ns	0.11µs	73.4ns	884736	5.52ms	3.10ms	1.53ms	4.51ms	3.25ms
128	0.25µs	0.11µs	56.9ns	0.25µs	0.17µs	995328	6.17ms	3.55ms	1.79ms	5.01ms	3.59ms
192	0.43µs	0.17µs	94.3ns	0.40µs	0.26µs	1048576	7.15ms	4.13ms	1.92ms	5.77ms	4.15ms
256	0.56µs	0.26µs	0.15µs	0.57µs	0.38µs	1179648	7.84ms	4.30ms	2.09ms	6.26ms	4.46ms
384	0.89µs	0.40µs	0.27µs	0.85µs	0.57µs	1327104	8.68ms	4.74ms	2.44ms	6.83ms	4.86ms
512	1.24µs	0.56µs	0.34µs	1.23µs	0.80µs	1492992	9.96ms	5.57ms	2.73ms	7.96ms	5.71ms
576	1.30µs	0.60µs	0.40µs	1.33µs	0.87µs	1572864	10.8ms	6.02ms	3.04ms	8.85ms	6.30ms
768	1.93µs	0.85µs	0.54µs	1.87µs	1.25µs	1769472	11.7ms	6.74ms	3.36ms	9.59ms	6.81ms
1024	2.57µs	1.20µs	0.77µs	2.70µs	1.82µs	1990656	13.2ms	7.50ms	3.84ms	10.6ms	7.54ms
1152	2.91µs	1.29µs	0.85µs	2.99µs	1.99µs	2097152	15.4ms	9.01ms	4.15ms	13.0ms	9.07ms
1536	4.17µs	1.83µs	1.19µs	4.14µs	2.76µs	2359296	16.3ms	9.58ms	4.55ms	13.5ms	9.60ms
1728	4.90µs	2.19µs	1.46µs	4.53µs	3.02µs	2654208	18.4ms	10.5ms	5.10ms	14.7ms	10.4ms
2048	6.21µs	2.65µs	1.67µs	5.81µs	3.98µs	2985984	21.1ms	11.9ms	5.91ms	16.5ms	11.9ms
2304	6.92µs	2.89µs	1.89µs	6.26µs	4.23µs	3145728	22.6ms	13.1ms	6.18ms	18.9ms	13.6ms
3072	9.96µs	4.03µs	2.73µs	8.77µs	5.90µs	3538944	25.1ms	14.3ms	6.97ms	20.4ms	14.9ms
3456	10.7µs	4.82µs	3.51µs	9.54µs	6.61µs	3981312	28.4ms	16.2ms	7.74ms	22.6ms	16.5ms
4096	13.7µs	6.10µs	4.22µs	12.5µs	8.65µs	4194304	33.3ms	19.5ms	9.19ms	27.6ms	21.0ms
4608	14.6µs	6.71µs	5.10µs	13.6µs	9.42µs	4718592	33.2ms	20.6ms	9.75ms	28.2ms	20.5ms
5184	17.5µs	8.10µs	5.66µs	16.2µs	11.2µs	5308416	38.8ms	22.9ms	11.9ms	30.9ms	22.6ms
6144	20.2µs	9.50µs	6.45µs	19.5µs	13.4µs	5971968	44.7ms	25.9ms	13.8ms	34.1ms	24.9ms
6912	21.8µs	10.1µs	7.07µs	20.9µs	14.6µs	6291456	47.7ms	28.6ms	15.5ms	42.0ms	32.6ms
8192	29.6µs	14.2µs	8.76µs	26.9µs	18.8µs	7077888	54.1ms	32.0ms	16.8ms	43.6ms	31.4ms
9216	30.6µs	13.9µs	9.76µs	29.7µs	20.7µs	7962624	62.3ms	36.7ms	19.4ms	47.4ms	34.6ms
10368	36.1µs	16.8µs	11.6µs	34.2µs	24.0µs	8388608	70.0ms	40.1ms	21.7ms	59.2ms	44.3ms
12288	43.0µs	19.7µs	13.4µs	41.6µs	29.0µs	9437184	78.7ms	45.2ms	24.7ms	61.5ms	45.3ms
13824	47.4µs	20.8µs	14.7µs	47.1µs	33.5µs	10616832	86.1ms	51.9ms	27.0ms	65.6ms	47.2ms
16384	63.8µs	27.6µs	18.3µs	61.4µs	42.6µs	11943936	98.5ms	58.4ms	31.1ms	72.8ms	51.9ms
18432	66.9µs	28.9µs	20.7µs	63.9µs	45.7µs	12582912	0.11s	62.6ms	33.4ms	87.2ms	65.5ms
20736	81.3µs	34.4µs	23.9µs	71.0µs	51.6µs	14155776	0.12s	70.4ms	36.5ms	93.6ms	64.9ms
24576	99.7µs	41.4µs	28.4µs	88.3µs	65.0µs	15925248	0.14s	79.3ms	41.5ms	0.10s	71.4ms
27648	0.12ms	44.6µs	32.1µs	97.0µs	70.9µs	16777216	0.16s	87.2ms	46.7ms	0.12s	93.1ms
32768	0.14ms	59.5µs	43.6µs	0.13ms	91.6µs	18874368	0.17s	94.9ms	50.4ms	0.13s	99.0ms
36864	0.16ms	68.9µs	43.8µs	0.13ms	98.3µs	21233664	0.20s	0.11s	57.2ms	0.14s	99.5ms
41472	0.16ms	83.6µs	52.0µs	0.15ms	0.11ms	23887872	0.22s	0.12s	64.2ms	0.16s	0.11s
49152	0.22ms	97.3µs	66.2µs	0.19ms	0.14ms	25165824	0.25s	0.14s	67.6ms	0.18s	0.15s
55296	0.23ms	0.12ms	77.5µs	0.20ms	0.15ms	28311552	0.28s	0.15s	76.7ms	0.21s	0.16s
65536	0.31ms	0.15ms	99.7µs	0.26ms	0.19ms	31850496	0.31s	0.17s	86.9ms	0.23s	0.16s
73728	0.33ms	0.17ms	0.11ms	0.28ms	0.21ms	33554432	0.36s	0.19s	93.7ms	0.26s	0.20s
82944	0.35ms	0.20ms	0.12ms	0.31ms	0.23ms	37748736	0.38s	0.26s	0.12s	0.29s	0.23s
98304	0.46ms	0.25ms	0.15ms	0.44ms	0.32ms	42467328	0.43s	0.28s	0.13s	0.32s	0.25s
110592	0.48ms	0.28ms	0.16ms	0.43ms	0.32ms	47775744	0.49s	0.32s	0.16s	0.36s	0.27s
124416	0.59ms	0.34ms	0.19ms	0.54ms	0.39ms	50331648	0.54s	0.33s	0.16s	0.39s	0.30s
131072	0.68ms	0.38ms	0.19ms	0.58ms	0.43ms	56623104	0.57s	0.38s	0.18s	0.44s	0.34s
147456	0.75ms	0.40ms	0.21ms	0.66ms	0.49ms	63700992	0.64s	0.41s	0.20s	0.49s	0.38s
165888	0.81ms	0.45ms	0.25ms	0.74ms	0.53ms	67108864	0.80s	0.49s	0.24s	0.54s	0.44s
196608	1.02ms	0.58ms	0.31ms	0.92ms	0.66ms	75497472	0.83s	0.51s	0.26s	0.61s	0.47s
221184	1.13ms	0.62ms	0.35ms	1.01ms	0.73ms	84934656	0.92s	0.61s	0.29s	0.67s	0.54s
248832	1.36ms	0.73ms	0.37ms	1.12ms	0.83ms	95551488	0.98s	0.66s	0.33s	0.75s	0.59s
262144	1.45ms	0.88ms	0.42ms	1.27ms	0.93ms	100663296	1.12s	0.78s	0.35s	0.83s	0.66s
294912	1.57ms	0.87ms	0.46ms	1.39ms	1.00ms	113246208	1.19s	0.78s	0.39s	0.90s	0.71s
331776	1.86ms	0.97ms	0.52ms	1.54ms	1.12ms	127401984	1.34s	0.94s	0.44s	1.03s	0.81s
393216	2.33ms	1.26ms	0.65ms	1.92ms	1.41ms	134217728	1.61s	1.00s	0.47s	1.15s	0.93s
442368	2.49ms	1.35ms	0.71ms	2.13ms	1.56ms	150994944	1.77s	1.21s	0.53s	1.26s	1.04s
497664	2.88ms	1.68ms	0.84ms	2.40ms	1.73ms	169869312	1.88s	1.29s	0.59s	1.38s	1.11s
524288	3.19ms	1.88ms	0.91ms	2.71ms	1.99ms	191102976	2.16s	1.46s	0.66s	1.59s	1.28s
589824	3.41ms	1.94ms	1.01ms	2.95ms	2.10ms	201326592	2.34s	1.54s	0.70s	1.69s	1.37s
663552	3.91ms	2.19ms	1.11ms	3.25ms	2.39ms	226492416	2.71s	1.81s	0.79s	1.90s	1.55s
786432	5.03ms	2.80ms	1.34ms	4.11ms	2.97ms	254803968	2.90s	1.93s	0.88s	2.08s	1.66s

Table 2. Extensive timings for number theoretic transforms of lengths $2^6 = 64$ up till $2^{28} = 254803968$ over the field $\mathbb{F}_p$ with $p = 1439 \cdot 2^{28} \cdot 3^6 + 1$.

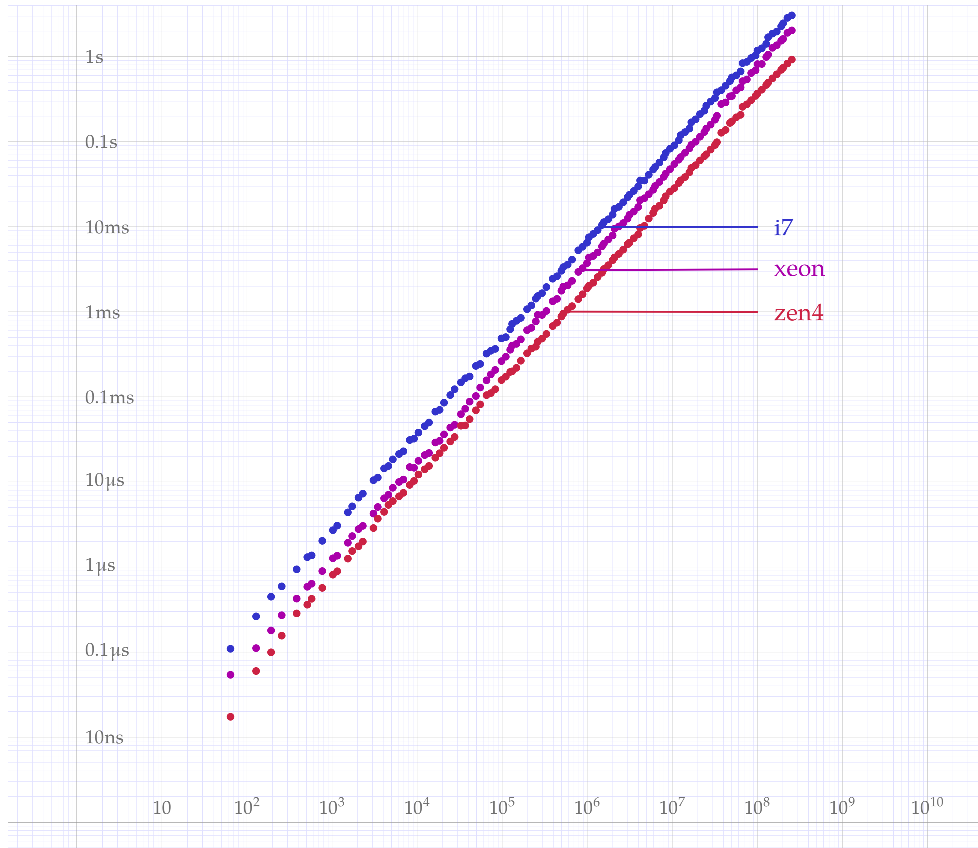

Figure 1. Graphical representation of the timings from Table 2, for the first three columns only.

5.1. Mono-threaded implementation

The main focus of this paper is on a mono-threaded implementation of number theoretic transforms, while exploiting SIMD instruction-level parallelism. We optimized our code for orders r with $2^6 \leqslant r < 2^{28}$ and $2^6 | r | 2^{28} \cdot 3^6$. For reference and completeness, a full table with all observed timings is provided in Table 2. For each entry, we computed three timings and reported the median one. In Figure 1, we also plotted the corresponding graph. For readability, this graph is limited to the x86-based architectures. The graph shows a familiar nearly affine behavior for the computation time in the double logarithmic scale. The sudden increase of the timings of the xeon curve near $r \approx 2^{16}$ is due to an inefficient memory hierarchy on this computer.

Now one NTT of a power of two length $r = 2^\ell$ consists of $^1/_2 r \ell$ butterflies. For a better understanding of the efficiency of our implementation it is therefore useful to divide our timings by $^1/_2 r \log_2 r$, even when r is not a power of two. Figure 2 shows the resulting normalized timings. In addition, Figure 3 shows these normalized timings when measuring the cost of a butterfly in cycles instead of nanoseconds.

From Figure 3, we can read off that a doubled SIMD width indeed makes the implementation approximately twice as fast on x86-based architectures. Although Apple's M1 and M3 chips only support 128-bit wide SIMD, they actually behave more like CPUs with 256-bit wide SIMD. This is probably due to the fact that Apple chips provide four instead of two FMA units, which potentially doubles the instruction-level parallelism.

For large r, the M1 and M3 perform even better, thanks to a more efficient memory hierarchy.

When multiplying the cost in cycles with the SIMD width $w \in \{2, 4, 8\}$ (while taking $w = 4$ for the M1 and M3 chips in view of what precedes), then we note that the cost of a single butterfly stays between 5 and 8 cycles for $r \leqslant 10^5$; this is similar to the performance of traditional scalar NTTs. For larger orders r, the cost on x86-based platforms slowly increases to 12 cycles per butterfly (and even to 15 cycles per butterfly for the Xeon W system and $r > 2^{25}$, but we recall that our implementation was not optimized as heavily in this range).

We also note the somewhat irregular behavior of the graphs in Figures 2 and 3, with discrepancies as large as 25% for close orders r. This is *not* due to inaccurate or irreproducible timings: it turns out that the efficiency of the generated codelets is highly dependent on the combinatorics of the divisors of r. In particular, if r is a power of two, then the "pool of available strategies" is somewhat smaller than if r is divisible by a few powers of 3. Consequently, the average costs in cycles tend to be slightly higher for power of two lengths. When using NTTs as the workhorse of evaluation-interpolation strategies (e.g. for integer multiplication), then it may be interesting to use truncated Fourier transforms (TFTs) from [15, 20]: this should allow us to smooth out the graphs from Figures 2 and 3, and benefit from the best possible NTTs for close orders $r' \gtrapprox r$.

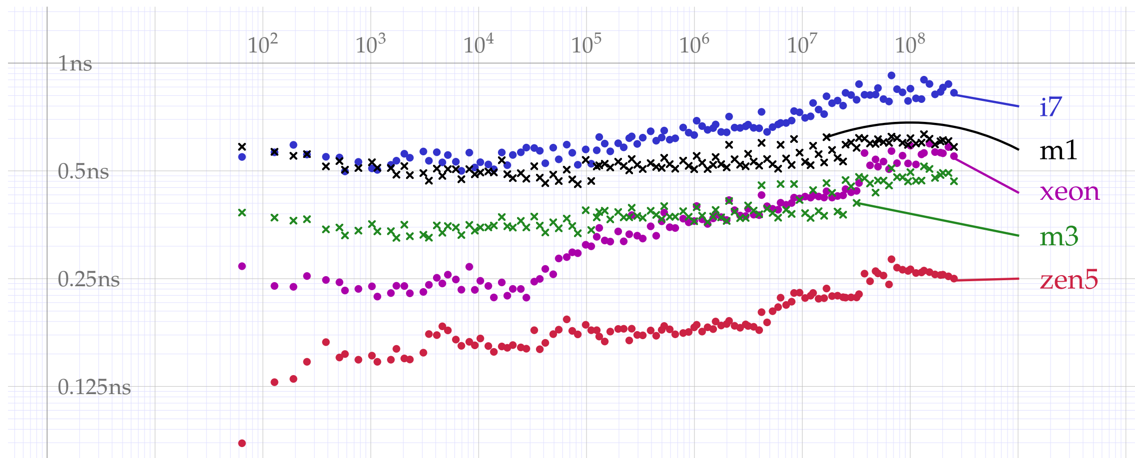

Figure 2. Timings divided by $\frac{1}{2} r \log_2 r$, which corresponds to the number of butterflies in the case when the order r of the NTT is a power of two.

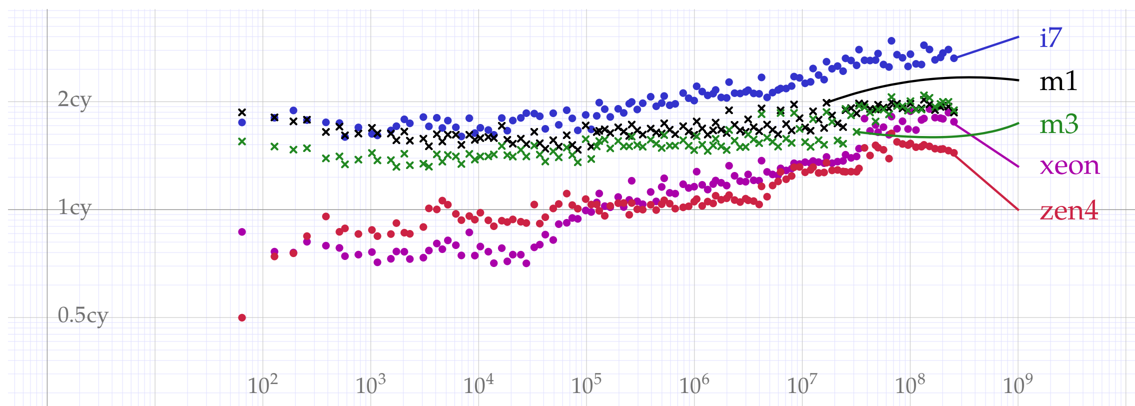

Figure 3. Variant of Figure 2 when using the cycle time as a unit instead of nanoseconds.

	i7			xeon			zen4			m1			m3		
	f48	f61	i61	f48	f61	i61	f48	f61	i61	f48	f61	i61	f48	f61	i61
64	1.51	1.92	4.71	0.86	1.09	4.94	0.49	0.63	4.16	1.87	2.37	2.77	1.55	1.97	2.62
256	1.54	1.95	4.26	0.80	1.02	4.40	0.83	1.06	3.77	1.78	2.26	2.47	1.48	1.89	2.30
1024	1.41	1.79	4.19	0.75	0.96	4.45	0.87	1.10	3.81	1.69	2.15	2.55	1.44	1.83	2.36
4096	1.56	1.98	4.22	0.79	1.01	4.47	0.99	1.26	3.82	1.62	2.06	2.51	1.43	1.81	2.34
16384	1.56	1.98	4.89	0.77	0.98	4.58	0.92	1.17	4.05	1.71	2.18	2.69	1.51	1.91	2.59
65536	1.64	2.08	5.47	0.91	1.15	4.64	1.09	1.39	4.02	1.61	2.05	2.74	1.49	1.89	2.70
262144	1.73	2.19	5.49	1.19	1.51	5.22	1.03	1.31	4.29	1.73	2.19	2.78	1.60	2.03	2.70
1048576	1.91	2.43	5.29	1.26	1.60	5.96	1.06	1.34	4.25	1.76	2.24	2.91	1.60	2.04	2.69
4194304	2.02	2.57	5.41	1.35	1.72	7.48	1.15	1.46	5.07	1.92	2.43	3.18	1.85	2.35	2.81
16777216	2.23	2.84	6.48	1.39	1.76	7.58	1.33	1.70	6.37	1.99	2.52	3.19	1.87	2.38	2.87
67108864	2.56	3.25	8.37	1.80	2.28	7.59	1.61	2.05	8.81	1.99	2.53	3.25	2.06	2.62	2.92

Table 3. Comparison of the number of clock cycles per butterfly between our new implementation and the implementation from [13]. We recall that the implementation from [13] uses a prime p with $2^{61} \leqslant p < 2^{62}$ and unsigned integer arithmetic. The "f48" columns correspond to our new timings for our prime p with $2^{48} \leqslant p < 2^{49}$. The "f61" columns contain the scaled variants through multiplication by $^{61}/_{48}$. The "i61" columns show the timings for the algorithm from [13].

We also compared our new algorithm to the one from [13]. For this purpose, we slightly adapted the companion C code from [13] to work also on M1 and M3 platforms and to support orders $r \leqslant 2^{28}$. The timings for both implementations are shown in Table 3. We recall that the algorithm from [13] uses a 62 bit prime and unsigned integer arithmetic, instead of a 49 bit prime and double precision floating point arithmetic. We have also indicated the appropriately rescaled timings.

5.2. Pure SIMD mode

In the previous subsection, we explored the cost of computing a single NTT of order r over $\mathbb{F}_p$, while exploiting SIMD support in the processor. However, in order to fully benefit from hardware SIMD support of width w, it is better to compute w NTTs in parallel, as a single NTT of order r over $\mathbb{F}_p^w$. Here $\mathbb{F}_p^w$ stands for the set of SIMD vectors of length w over $\mathbb{F}_p$. Such a vector fits exactly into a single hardware SIMD register. We will say that we are in *pure SIMD mode* when we can compute w NTTs in parallel in this way. This typically happens when we need to compute many NTTs (e.g. when doing an integer matrix product as in [13]), in chunks of w NTTs at a time. By contrast, we are in *thwarted SIMD mode* if we want to compute a single or less than w NTTs, as in the previous subsection. Although this mode may still benefit from SIMD arithmetic, this typically requires non-trivial data rearrangements as described in section 4.4.

In order to compare the pure and thwarted modes, it is useful to compare the following three costs:

1. the cost of a single NTT of length r over $\mathbb{F}_p$,
2. w^{-1} times the cost of w NTTs of length r over $\mathbb{F}_p$, and
3. w^{-1} times the cost of a single NTT of length r over $\mathbb{F}_p^w$.

Although the first two costs ought to be similar most of the time, this is not necessarily the case when r is close to one of the (L1, L2, L3, ...) cache sizes. In Table 4, we show

r	i7	×4	simd_4	xeon	×8	simd_8	zen4	×8	simd_8	m3	×2	simd_2
64	0.10µs	0.11µs	83.1ns	51.4ns	65.5ns	37.8ns	16.5ns	33.1ns	24.5ns	73.4ns	73.3ns	52.0ns
128	0.25µs	0.27µs	0.19µs	0.11µs	0.12µs	85.0ns	56.9ns	69.2ns	56.5ns	0.17µs	0.16µs	0.13µs
256	0.56µs	0.57µs	0.40µs	0.26µs	0.28µs	0.18µs	0.15µs	0.17µs	0.13µs	0.38µs	0.37µs	0.29µs
512	1.24µs	1.27µs	1.05µs	0.56µs	0.63µs	0.49µs	0.34µs	0.39µs	0.35µs	0.80µs	0.80µs	0.67µs
1024	2.57µs	2.65µs	2.32µs	1.20µs	1.29µs	1.21µs	0.77µs	0.81µs	0.77µs	1.82µs	1.80µs	1.67µs
2048	6.21µs	6.26µs	4.88µs	2.65µs	2.76µs	2.39µs	1.67µs	1.73µs	1.62µs	3.98µs	3.82µs	3.34µs
4096	13.7µs	13.2µs	11.5µs	6.10µs	6.35µs	5.21µs	4.22µs	4.22µs	4.21µs	8.65µs	8.41µs	7.67µs
8192	29.6µs	31.3µs	27.7µs	14.2µs	14.6µs	13.6µs	8.76µs	8.92µs	8.60µs	18.8µs	18.3µs	16.9µs
16384	63.8µs	64.9µs	57.6µs	27.6µs	31.9µs	33.5µs	18.3µs	18.6µs	17.9µs	42.6µs	40.1µs	36.6µs
32768	0.14ms	0.15ms	0.13ms	59.5µs	69.3µs	75.1µs	43.6µs	42.7µs	41.2µs	91.6µs	86.7µs	77.7µs
65536	0.31ms	0.30ms	0.31ms	0.15ms	0.17ms	0.16ms	99.7µs	0.10ms	95.1µs	0.19ms	0.18ms	0.17ms
131072	0.68ms	0.72ms	0.66ms	0.38ms	0.44ms	0.40ms	0.19ms	0.21ms	0.20ms	0.43ms	0.40ms	0.52ms
262144	1.45ms	1.49ms	1.53ms	0.88ms	1.02ms	0.94ms	0.42ms	0.43ms	0.43ms	0.93ms	0.87ms	0.87ms
524288	3.19ms	3.31ms	3.17ms	1.88ms	2.09ms	1.95ms	0.91ms	0.94ms	0.92ms	1.99ms	1.89ms	1.88ms
1048576	7.15ms	7.03ms	7.04ms	4.13ms	4.62ms	4.31ms	1.92ms	1.97ms	2.13ms	4.15ms	4.27ms	4.15ms
2097152	15.4ms	15.7ms	15.3ms	9.01ms	9.32ms	9.63ms	4.15ms	5.38ms	5.02ms	9.07ms	9.73ms	9.42ms
4194304	33.3ms	36.1ms	35.7ms	19.5ms	19.6ms	20.5ms	9.19ms	11.0ms	11.3ms	21.0ms	20.8ms	20.2ms
8388608	70.0ms	77.9ms	76.5ms	40.1ms	40.9ms	49.4ms	21.7ms	23.4ms	24.4ms	44.3ms	43.3ms	41.7ms
16777216	0.16s	0.16s	0.16s	87.2ms	90.5ms	0.11s	46.7ms	51.4ms	53.2ms	93.1ms	90.6ms	86.0ms

Table 4. Comparison between the timings of NTTs in the thwarted and pure SIMD modes. The greyed entries correspond to timings that were obtained using slightly less optimized DFTs.

a detailed comparison of these costs for power of two lengths. The overhead of thwarted with respect to pure SIMD is most important for small orders r.

5.3. Multi-threading

The main focus of this paper has been on an almost optimal implementation of the NTT using SIMD double precision arithmetic, but in the mono-threaded case. Naturally, we were curious about the performance of our work when using multiple threads. In this last subsection, we report on some preliminary experiments, while emphasizing that this is work in progress which has not yet benefited from a similar level of optimization. We recompiled MATHEMAGIX with the `--enable-threads` configure option.

Before measuring the thread efficiency of our implementation, it is useful to analyze the maximal theoretical speed-up that multi-threading may provide on our testing platforms. For this, we created a simple loop that calls many times an empty function, and measured the speed-up when executing this loop in a multi-threaded fashion. The results have been reported in Table 5. For the x86-based platforms, we note that the number of logical threads is twice as high as the number of physical threads, due to hyperthreading.

	1	2	3	4	5	6	7	8	9	10	11	12	14	16	24	32
i7	1.00	2.04	2.79	3.35	3.72	4.09	4.50	4.64								
xeon	1.00	2.00	2.99	4.00	5.00	5.98	6.61	6.96	6.84	7.38	7.57	7.47	7.65	7.78		
zen4	1.00	1.99	2.93	3.83	4.69	5.57	6.41	7.23	8.07	8.99	9.79	10.6	12.2	13.1	17.5	20.7
m1	1.00	1.93	2.80	3.73	4.21	4.71	5.19	5.61								
m3	1.00	1.89	2.82	3.75	4.67	5.32	5.96	6.62	7.26	7.87	8.46	8.56				

Table 5. Speed-up when executing a dummy loop using multiple threads. The top row indicates the number of threads that we use to execute our code. The main table indicates the observed speed-ups. The greyed entries correspond to the degraded regime in which hyperthreading or efficiency cores enter the stage.

	i7			xeon			m1			m3		
	4	6	8	8	12	16	4	6	8	6	8	12
4096	1.16						2.12			1.94		
8192	1.59	1.25					2.54	1.58		2.62	1.53	
16384	2.05	1.75	1.34	1.12			3.04	2.24	2.07	3.59	2.44	1.91
32768	2.31	2.33	1.75	1.61	1.24		3.17	2.57	2.45	4.06	3.15	2.79
65536	2.66	2.62	2.17	2.42	1.94		3.30	2.83	2.82	4.42	3.48	3.36
131072	2.74	2.92	2.22	3.94	3.09	1.87	3.50	3.25	3.09	4.72	4.34	4.16
262144	2.65	2.98	2.44	4.75	3.27	2.22	3.69	3.65	3.77	4.87	4.34	5.07
524288	2.98	3.01	2.76	4.22	3.94	3.03	3.63	3.93	3.80	4.67	4.62	5.02
1048576	3.00	3.34	2.97	4.35	4.08	3.48	3.45	3.71	3.71	4.61	4.53	4.71
2097152	3.20	3.29	3.29	3.94	3.77	3.18	3.42	3.76	4.02	4.16	4.50	4.70
4194304	3.16	3.11	3.02	4.04	3.59	3.08	3.28	3.64	3.84	4.75	5.04	5.61
8388608	3.00	3.09	3.02	3.81	3.33	3.11	3.21	3.70	4.09	4.70	5.12	5.63
16777216	3.06	3.12	3.27	4.26	3.66	3.26	3.18	3.69	4.05	4.79	5.15	5.87
33554432	3.06	3.18	3.27	3.78	3.79	3.42	3.30	3.75	4.15	4.72	5.63	6.35

Table 6. Parallel speed-ups for our number theoretic transforms as a function of the transform length r, the platform, and the number of threads τ.

It turns out that our simple loop does benefit from hyperthreading, but that this is less likely to happen for highly optimized HPC code like our NTTs. For the M1 and M3 processors, efficiency cores tend to be much slower than performance cores, with similar pitfalls as in the case of hyperthreading. Note that the observed speed-ups always decreased when using a number of threads that exceeds the maximal number of logical threads for the platform.

For our main experiments, the number of threads τ is fixed for each individual timing. The main codelets that benefit from multi-threaded execution are of the form $\mathrm{Id}_n \otimes T$ or $T \otimes \mathrm{Id}_n$ for some other codelet T and some large integer n. For a parameter ϕ (that we call the *flooding factor*), let us write $n = n_1 + \cdots + n_{\tau\phi}$ with $\lfloor n / (\tau \phi) \rfloor = n_1 \leqslant \cdots \leqslant n_{\tau\phi} = \lceil n / (\tau \phi) \rceil$. Now we subdivide the execution of, say, $T \otimes \mathrm{Id}_n$ into the execution of chunks $T \otimes \mathrm{Id}_{n_i}$ for $i = 1, \ldots, \tau \phi$. These chunks are executed by our τ threads using the work stealing paradigm. In our codelet factory, the most appropriate flooding factor ϕ for given n and T is determined dynamically.

The first scenario we tested is that of a single NTT of length r, while using τ threads. For each platform, we investigate three values for τ: the number of physical or performance cores, the number of logical or performance plus efficiency cores, and an intermediate value. The resulting timings are shown in Table 6. We observed a large variance, especially on x86 platforms. We consider the speed-ups on the m1 and m3 platforms to be reasonably satisfactory for a first implementation. The timings for the other platform are far from optimal. We omitted the results for the zen4 platform, since they were so absurd that they require further investigation.

The second scenario is when we wish to compute a large number N of NTTs of the same order r. This is similar to the pure SIMD mode that we investigated in subsection 5.2. We use a similar work stealing approach as above. Each individual thread repeatedly computes NTTs of order r over $\mathbb{F}_p^w$. In Table 7, we reported detailed timings in the single case when $N = r = 1024$ and only for the m3 platform. The table also indicates the best flooding factor ϕ as a function of the number of threads τ.

	1	2	4	8	16	32	64	128	256	time
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	3.50ms
2	1.93	1.93	1.94	1.93	1.93	1.92	1.91	1.88	1.84	1.81ms
3	2.89	2.90	2.88	2.88	2.88	2.87	2.86	2.84	2.76	1.21ms
4	3.82	3.85	3.83	3.81	3.80	3.78	3.73	3.67	3.52	0.91ms
5	4.65	4.66	4.61	4.63	4.61	4.62	4.41	4.34	4.24	0.75ms
6	3.18	4.01	4.56	4.83	4.67	4.87	4.85	4.72	4.28	0.72ms
7	3.58	4.49	4.75	4.65	5.01	5.21	5.17	4.94	4.32	0.67ms
8	3.85	4.80	4.84	5.01	5.24	5.38	5.33	4.91	4.33	0.65ms
9	4.32	5.00	4.99	5.01	5.46	5.64	5.45	5.14	4.30	0.62ms
10	4.56	5.75	5.56	5.78	5.68	5.87	5.71	5.18	4.29	0.60ms
11	4.01	5.14	5.51	5.74	5.79	5.87	5.81	5.14	4.19	0.60ms
12	4.07	4.78	5.26	5.11	5.38	5.39	5.33	4.93	3.81	0.65ms

Table 7. Speed-ups on an M3 processor for $N := 1024$ NTTs of length $r := 1024$ with SIMD coefficients in $\mathbb{F}_p^2$, where $p := 1439 \cdot 2^{28} \cdot 3^6 + 1$, as a function of the number τ of threads and the flooding factor ϕ. The highlighted entries in each row correspond to the best flooding factor for a given number of threads. The last column reports the timing for the best flooding factor.

BIBLIOGRAPHY

[1] ANSI/IEEE. IEEE standard for binary floating-point arithmetic. Technical Report, ANSI/IEEE, New York, 2008. ANSI-IEEE Standard 754-2008. Revision of IEEE 754-1985, approved on June 12, 2008 by IEEE Standards Board.

[2] J. Bradbury, N. Drucker, and M. Hillenbrand. NTT software optimization using an extended Harvey butterfly. *Cryptology ePrint Archive, Paper 2021/1396*, 2021. `https://eprint.iacr.org/2021/1396`.

[3] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*. Springer-Verlag, Berlin, 1997.

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

[5] J.-G. Dumas, P. Giorgi, and C. Pernet. FFPACK: finite field linear algebra package. In J. Schicho, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 119–126. New York, NY, USA, 2004. ACM.

[6] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.*, 35(3):19–1, 2008.

[7] P. Fortin, A. Fleury, F. Lemaire, and M. Monagan. High-performance SIMD modular arithmetic for polynomial evaluation. *Concurr. Comput. Pract. Exp.*, 33(16):e6270, 2021.

[8] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180. New York, NY, USA, 1999. ACM.

[9] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.

[10] S. Fu, N. Zhang, and F. Franchetti. Accelerating high-precision number theoretic transforms using Intel AVX-512. 2024. `https://spiral.ece.cmu.edu/pub-spiral/pubfile/PACT_2024_AVX_371.pdf`.

[11] I. J. Good. The interaction algorithm and practical Fourier analysis. *J. R. Stat. Soc. Series B*, 20(2):361–372, 1958.

[12] W. Hart and the FLINT Team. FLINT: Fast Library for Number Theory. From 2008. Software available at `http://www.flintlib.org`.

[13] D. Harvey. Faster arithmetic for number-theoretic transforms. *J. Symbolic Comput.*, 60:113–119, 2014.

[14] M. T. Heideman, D. H. Johnson, and C. S. Burrus. Gauss and the history of the fast Fourier transform. *Arch. Hist. Exact Sci.*, 34(3):265–277, 1985.

[15] J. van der Hoeven. The truncated Fourier transform and applications. In J. Gutierrez, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 290–296. New York, NY, USA, 2004. ACM.

[16] J. van der Hoeven. *The Jolly Writer. Your Guide to GNU TeXmacs*. Scypress, 2020.

[17] J. van der Hoeven and G. Lecerf. *Mathemagix User Guide*. HAL, 2013. `https://hal.archives-ouvertes.fr/hal-00785549`.

[18] J. van der Hoeven and G. Lecerf. Evaluating straight-line programs over balls. In P. Montuschi, M. Schulte, J. Hormigo, S. Oberman, and N. Revol, editors, *2016 IEEE 23nd Symposium on Computer Arithmetic*, pages 142–149. IEEE, 2016.

[19] J. van der Hoeven, G. Lecerf, and G. Quintin. Modular SIMD arithmetic in Mathemagix. *ACM Trans. Math. Softw.*, 43(1):5–1, 2016.

[20] R. Larrieu. The truncated Fourier transform for mixed radices. In M. Burr, editor, *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 261–268. New York, NY, USA, 2017. ACM.

[21] J. M. Pollard. The fast Fourier transform in a finite field. *Math. Comput.*, 25(114):365–374, 1971.

[22] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proc. IEEE*, 93(2):232–275, 2005. Special issue on “Program Generation, Optimization, and Adaptation”.

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

[24] D. Takahashi. An implementation of parallel number-theoretic transform using Intel AVX-512 instructions. In F. Boulier, M. England, T. M. Sadykov, and E. V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing. CASC 2022.*, volume 13366 of *Lect. Notes Comput. Sci.*, pages 318–332. Springer, Cham, 2022.

[25] N. Zhang, A. Ebel, N. Neda, P. Brinich, B. Reynwar, A. G. Schmidt, M. Franusich, J. Johnson, B. Reagen, and F. Franchetti. Generating high-performance number theoretic transform implementations for vector architectures. In *2023 IEEE High Performance Extreme Computing Conference* (*HPEC*), pages 1–7. Los Alamitos, CA, USA, 2023. IEEE.