Faster FFTs in medium precision

Joris van der Hoeven[a], Grégoire Lecerf[b]

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

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

November 10, 2014

In this paper, we present new algorithms for the computation of fast Fourier transforms over complex numbers for "medium" precisions, typically in the range from 100 until 400 bits. On the one hand, such precisions are usually not supported by hardware. On the other hand, asymptotically fast algorithms for multiple precision arithmetic do not pay off yet. The main idea behind our algorithms is to develop efficient vectorial multiple precision fixed point arithmetic, capable of exploiting SIMD instructions in modern processors.

Keywords: floating point arithmetic, quadruple precision, complexity bound, FFT, SIMD

A.C.M. subject classification: G.1.0 Computer-arithmetic

A.M.S. subject classification: 65Y04, 65T50, 68W30

1. Introduction

Multiple precision arithmetic [4] is crucial in areas such as computer algebra and cryptography, and increasingly useful in mathematical physics and numerical analysis [2]. Early multiple precision libraries appeared in the seventies [3], and nowadays GMP [11] and MPFR [8] are typically very efficient for large precisions of more than, say, 1000 bits. However, for precisions which are only a few times larger than the machine precision, these libraries suffer from a large overhead. For instance, the MPFR library for arbitrary precision and IEEE-style standardized floating point arithmetic is typically about a factor 100 slower than double precision machine arithmetic.

This overhead of multiple precision libraries tends to further increase with the advent of wider SIMD (*Single Instruction, Multiple Data*) arithmetic in modern processors, such as the Intel® AVX technology. Indeed, it is hard to take advantage of wide SIMD instructions when implementing basic arithmetic for integer sizes of only a few words. In order to fully exploit SIMD instructions, one should rather operate on vectors of integers of a few words. A second problem with current SIMD arithmetic is that CPU vendors tend to privilege wide floating point arithmetic over wide integer arithmetic, which would be most useful for speeding up multiple precision libraries.

In order to make multiple precision arithmetic more useful in areas such as numerical analysis, it is a major challenge to reduce the overhead of multiple precision arithmetic for small multiples of the machine precision, and to build libraries with direct SIMD arithmetic for multiple precision numbers.

One existing approach is based on the "**TwoSum**" and "**TwoProduct**" algorithms [7, 22], which allow for the exact computation of sums and products of two machine floating point numbers. The results of these operations are represented as sums $x + y$ where x and y have no "overlapping bits" (e.g. $\lfloor \log_2 |x| \rfloor \geqslant \lfloor \log_2 |y| \rfloor + 52$ or $x = y = 0$). The **TwoProduct** algorithm

can be implemented using only two instructions when hardware features the *fused-multiply-add* (FMA) and *fused-multiply-subtract* (FMS) instructions, as is for instance the case for INTEL's AVX2 enabled processors. The **TwoSum** algorithm could be done using only two instructions as well if we had similar fused-add-add and fused-add-subtract instructions. Unfortunately, this is not the case for current hardware.

It is classical that double machine precision arithmetic can be implemented reasonably efficiently in terms of the **TwoSum** and **TwoProduct** algorithms [7, 22, 23]. The approach has been further extended in [24] to higher precisions. Specific algorithms are also described in [21] for triple-double precision, and in [15] for quadruple-double precision. But these approaches tend to become inefficient for large precisions.

For certain applications, efficient fixed point arithmetic is actually sufficient. One good example is the fast discrete Fourier transform (FFT) [6], which is only numerically accurate if all input coefficients are of the same order of magnitude. This means that we may scale all input coefficients to a fixed exponent and use fixed point arithmetic for the actual transform. Moreover, FFTs (and especially multi-dimensional FFTs) naturally benefit from SIMD arithmetic.

In this paper, we will show how to efficiently implement FFTs using fixed point arithmetic for small multiples of the machine precision. In fact, the actual working precision will be of the form $k\,p$, where $k \in \{2, 3, ...\}$, $p = \mu - \delta$ for a small integer δ (typically, $\delta = 4$), and μ denotes the number of fractional bits of the mantissa (also known as the trailing significant field, so that $\mu = 52$ for IEEE double precision numbers).

Allowing for a small number δ of "nail" bits makes it easier to handle carries efficiently. On the downside, we sacrifice a few bits and we need an additional routine for normalizing our numbers. Fortunately, normalizations can often be delayed. For instance, every complex butterfly operation during an FFT requires only four normalizations (the real and imaginary parts of the two outputs).

Redundant representations with nail bits, also known as *carry-save* representations, are very classical in hardware design, but rarely used in software libraries. The term "nail bits" was coined by the GMP library [11], with a different focus on high precisions. However, the GMP code which uses this technology is experimental and disabled by default. Redundant representations have also found some use in modular arithmetic. For instance, they recently allowed to speed up modular FFTs [13].

Let $\mathsf{F}_\pi(n)$ denote the time spent to compute an FFT of length n at a precision of π bits. For an optimal implementation at precision $k\,\mu$, we would expect that $\mathsf{F}_{k\mu}(n) \approx k\,\mathsf{F}_\mu(n)$. However, the naive implementation of a product at precision $k\ \mu$ requires at least $\binom{k+1}{2}\ \mu$ machine multiplications, so $\mathsf{F}_{k\mu}(n) \approx \binom{k+1}{2}\,\mathsf{F}_\mu(n)$ is a more realistic goal for small k. The first contribution of this paper is a series of optimized routines for basic fixed point arithmetic for small k. These routines are vectorial by nature and therefore well suited to current INTEL AVX technology. The second contribution is an implementation inside the C++ librairies of MATHEMAGIX [17] (which is currently limited to a single thread). For small k, our timings seem to indicate that $\mathsf{F}_{kp}(n) \approx 2\binom{k+1}{2}\,\mathsf{F}_\mu(n)$. For $k = 2$, we compared our implementation with FFTW3 [9] (in which case we only obtained $\mathsf{F}_{2\mu}(n) \approx 200\,\mathsf{F}_\mu(n)$ when using the built-in type `__float128` of GCC) and a home-made *double-double* implementation along the same lines as [23] (in which case we got $\mathsf{F}_{2\mu}(n) \approx 10\,\mathsf{F}_\mu(n)$).

Although the main ingredients of this paper (fixed point arithmetic and nail bits) are not new, we think that we introduced a novel way to combine them and demonstrate their efficiency in conjunction with modern wide SIMD technology. There has been recent interest in efficient multiple precision FFTs for the accurate integration of partial differential equations from hydrodynamics [5]. Our new algorithms should be useful in this context, with special emphasis on the case when $k = 2$. We also expect that the ideas in this paper can be adapted for some other applications, such as efficient computations with truncated numeric Taylor series for medium working precisions.

Our paper is structured as follows. Section 2 contains a detailed presentation of fixed point arithmetic for the important precision $2\,p$. In Section 3, we show how to apply this arithmetic to the computation of FFTs, and we provide timings. In Section 4, we show how our algorithms generalize to higher precisions $k\,p$ with $k > 2$, and present further timings for the cases when $k = 3$ and $k = 4$. In our last section, we discuss possible speed-ups if certain SIMD instructions were hardware-supported, as well as some ideas on how to use asymptotically faster algorithms in this context.

Notations

Throughout this paper, we assume IEEE arithmetic with correct rounding and we denote by $\mathbb{F}$ the set of machine floating point numbers. We let $\mu > 2$ be the machine precision minus one (which corresponds to the number of fractional bits of the mantissa) and let $E_{\min}$ and $E_{\max}$ be the minimal and maximal exponents of machine floating point numbers. For IEEE double precision numbers, this means that $\mu = 52$, $E_{\min} = -1022$ and $E_{\max} = 1023$.

The algorithms in this paper are designed to work with all possible rounding modes. Given $x, y \in \mathbb{F}$ and $* \in \{+, -, \cdot\}$, we denote by $\circ(x * y)$ the rounding of $x * y$ according to the chosen rounding mode. If e is the exponent of $x * y$ and $E_{\max} > e \geqslant E_{\min} + \mu$ (i.e. in absence of overflow and underflow), then we notice that $|\circ(x * y) - x * y| < 2^{e-\mu}$.

Modern processors usually support fused-multiply-add (FMA) and fused-multiply-subtract (FMS) instructions, both for scalar and SIMD vector operands. Throughout this paper, we assume that these instructions are indeed present, and we denote by $\circ(x\,y + z)$ and $\circ(x\,y - z)$ the roundings of $x\,y + z$ and $x\,y - z$ according to the chosen rounding mode.

2. Fixed point arithmetic for quasi doubled precision

Let $p \in \{6, \ldots, \mu - 2\}$. In this section, we are interested in developing efficient fixed point arithmetic for a bit precision $2\,p$. Allowing p to be smaller than μ makes it possible to efficiently handle carries during intermediate computations. We denote by $\delta = \mu - p \geqslant 2$ the number of extra "nail" bits.

2.1. Representation of fixed point numbers

We denote by $\mathcal{F}_{p,2}$ the set of fixed point numbers which can be represented as

$$x = x_0 + x_1,$$

where $x_0, x_1 \in \mathbb{F}$ are such that

$$x_0 \in \mathbb{Z}\,2^{-p} \tag{1}$$
$$|x_0| < 2^{\delta} \tag{2}$$
$$|x_1| < 2^{\delta - p}. \tag{3}$$

It will be convenient to write $x = [x_0, x_1]$ for numbers of the above form. Since $\mathcal{F}_{p,2}$ contains all numbers $x = k\,2^{-2p}$ with $k \in \mathbb{Z}$ and $|x| < 1$, this means that $\mathcal{F}_{p,2}$ can be thought of as a fixed point type of precision $2\,p$.

Remark 1. Usually, we will even have $|x_0| < 1$ and $|x_1| < 2^{-p}$, but the extra flexibility provided by (2) and (3) will be useful during intermediate computations. In addition, for efficiency reasons, we do *not* require that $x_1 \in \mathbb{Z}\,2^{-2p}$.

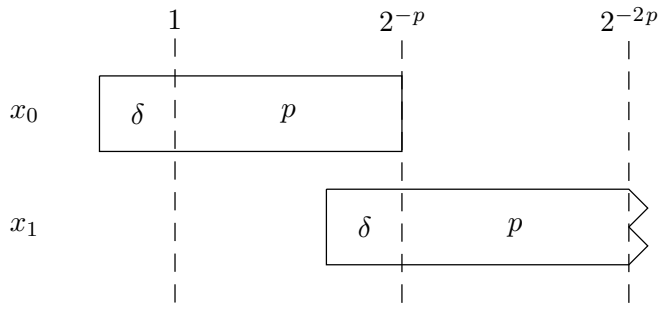

Figure 1. Schematic representation of the decomposition $x = [x_0, x_1] = x_0 + x_1$.

2.2. Splitting numbers at a given exponent

An important subalgorithm for efficient fixed point arithmetic computes the truncation of a floating point number at a given exponent:

Algorithm Split$_e(x)$

$a := \circ(x + 3/2 \cdot 2^{e+\mu})$
return $\circ(a - 3/2 \cdot 2^{e+\mu})$

PROPOSITION 2. *Given $x \in \mathbb{F}$ and $e \in \{E_{\min}, \ldots, E_{\max} - \mu\}$ such that $|x| < 2^{e+\mu-2}$, the algorithm* **Split**$_e$ *computes a number $\tilde{x} \in \mathbb{F}$ with $\tilde{x} \in \mathbb{Z}\, 2^e$ and $|\tilde{x} - x| < 2^e$.*

Proof. Let $A = x + 3/2 \cdot 2^{e+\mu}$. Since $|x| < 2^{e+\mu-2}$, we have

$$5/4 \cdot 2^{e+\mu} < A < 7/4 \cdot 2^{e+\mu}.$$

Since a is obtained by rounding A, it follows that

$$5/4 \cdot 2^{e+\mu} \leqslant a \leqslant 7/4 \cdot 2^{e+\mu},$$

whence the exponent of a is $e + \mu$ and $|A - a| < 2^e$, for any rounding mode. Since $3/2 \cdot 2^{e+\mu}$ also has exponent $e + \mu$, it follows that $\tilde{x} = a - 3/2 \cdot 2^{e+\mu}$ satisfies $\tilde{x} = \circ(\tilde{x})$. Furthermore, a and $3/2 \cdot 2^{e+\mu}$ are both integer multiples of 2^e, whence so is $\tilde{x}$. Finally, $|\tilde{x} - x| = |(a - 3/2 \cdot 2^{e+\mu}) - (A - 3/2 \cdot 2^{e+\mu})| < 2^e$. □

Remark 3. Assuming that the rounding mode is set towards zero, the condition $|x| < 2^{e+\mu-1}$ suffices in Proposition 2. Indeed, this weaker condition implies $2^{e+\mu} < A < 2 \cdot 2^{e+\mu}$, and therefore $2^{e+\mu} \leqslant a < 2 \cdot 2^{e+\mu}$, with the rest of the proof unchanged. The same remark holds for Proposition 7 below and allows several of the subsequent bounds to be slightly improved. As a consequence, this sometimes allows us to decrease δ by one. However, in this paper, we prefer to avoid hypotheses on the rounding mode, for the sake of portability, efficiency, and simplicity. Nevertheless, our observation may be useful on some recent processors (such as INTEL's AVX-512 enabled ones), which make it possible to force rounding modes at the level of individual instructions.

2.3. Normalization

Let $B \in [1, 2^\delta]$. We will write $\mathcal{F}_{p,2;B}$ for the subset of $\mathcal{F}_{p,2}$ of all numbers $x = [x_0, x_1]$ with

$$|x_1| < B\, 2^{-p} \leqslant 2^{\delta - p}.$$

Numbers in $\mathcal{F}_{p,2;1}$ are said to be in *normal form*.

Remark 4. Both $x = [2^{-p}, -2^{-p}/2]$ and $y = [0, 2^{-p}/2]$ are in normal form, and both x and y represent the number $2^{-p}/2$. Consequently, our fixed point representation is still *redundant* under this normalization (this is reminiscent from Avižienis' representations [1]).

Algorithm Normalize(x)

$c := \mathbf{Split}_{-p}(x_1)$
return $[\circ(x_0 + c), \circ(x_1 - c)]$

PROPOSITION 5. *Given $x \in \mathcal{F}_{p,2}$ with $|x_0| < 2^\delta - 2^{-p+\delta}$, the algorithm* **Normalize** *returns $\tilde{x} \in \mathcal{F}_{p,2;1}$ with $|\tilde{x} - x| < 2^{-2p-\delta}$.*

Proof. Since $p \geqslant 2$, using Proposition 2 with $e = -p$, we have $|x_1 - c| < 2^{-p}$ and $c \in \mathbb{Z}\, 2^{-p}$. Using the fact that $|x_1| < 2^{\delta-p}$, this entails $|c| < (2^\delta + 1)\, 2^{-p}$ and therefore $|c| \leqslant 2^{\delta-p}$. Hence $|x_0 + c| < 2^\delta$, so that $\circ(x_0 + c) = x_0 + c$. Finally, $|x_1 - c| < 2^{-p}$ implies $|\circ(x_1 - c) - (x_1 - c)| < 2^{-p-\mu}$, so that $|\tilde{x} - x| \leqslant |\tilde{x}_0 - (x_0 + c)| + |\tilde{x}_1 - (x_1 - c)| < 2^{-p-\mu}$. □

2.4. Addition and subtraction

Non normalized addition and subtraction of fixed point numbers are straightforward:

Algorithm Add(x, y)

return $[\circ(x_0 + y_0), \circ(x_1 + y_1)]$

Algorithm Subtract(x, y)

return $[\circ(x_0 - y_0), \circ(x_1 - y_1)]$

PROPOSITION 6. *Let $x \in \mathcal{F}_{p,2;B}$ and $y \in \mathcal{F}_{p,2;C}$ with $S := B + C + 2^{-p} \leqslant 2^\delta$. If $|x_0 + y_0| < 2^\delta$, then we have $a = \mathbf{Add}(x, y) \in \mathcal{F}_{p,2;S}$ with $|a - (x + y)| < 2^{-2p}$. If $|x_0 - y_0| < 2^\delta$, then we have $b = \mathbf{Subtract}(x, y) \in \mathcal{F}_{p,2;S}$ with $|b - (x - y)| < 2^{-2p}$.*

Proof. Since both x_0 and y_0 are multiple of 2^{-p}, the addition $a_0 = x_0 + y_0$ is exact. Furthermore, the exponents of x_1 and y_1 are both strictly bounded by $\delta - p$. Consequently, $|a_1 - (x_1 + y_1)| < 2^{\delta-p-\mu} = 2^{-2p}$ and $|a_1| < |x_1 + y_1| + 2^{-2p} \leqslant |x_1| + |y_1| + 2^{-2p} < S\, 2^{-p}$. Finally $|a - (x + y)| \leqslant |a_0 - (x_0 + y_0)| + |a_1 - (x_1 + y_1)| = |a_1 - (x_1 + y_1)| < 2^{-2p}$. The statement for the subtraction follows by replacing y by $-y$. □

2.5. Multiplication

Our multiplication algorithm of fixed point numbers is based on a subalgorithm $\mathbf{LongMul}_e$ which computes the exact product of two numbers $x, y \in \mathbb{F}$ in the form of a sum $x\, y = h + l$, with the additional constraint that $h \in \mathbb{Z}\, 2^e$. Without this additional constraint (and in absence of overflow and underflow), h and l can be computed using the classical "Two Product" algorithm: $h := \circ(x\, y)$, $l := \circ(x\, y - h)$. Our $\mathbf{LongMul}_e$ algorithm exploits the FMA and FMS instructions in a similar way.

Algorithm LongMul$_e$(x, y)

$a := \circ(x\, y + {}^3\!/_2 \cdot 2^{e+\mu})$
$h := \circ(a - {}^3\!/_2 \cdot 2^{e+\mu})$
$l := \circ(x\, y - h)$
return (h, l)

Algorithm Multiply(x, y)

$(h, l) := \mathbf{LongMul}_{-p}(x_0, y_0)$
$l := \circ(x_0\, y_1 + l)$
$l := \circ(x_1\, y_0 + l)$
return $[h, l]$

PROPOSITION 7. *Let $x, y \in \mathbb{F}$ and $e \in \{E_{\min} + \mu, \ldots, E_{\max} - \mu\}$ be such that $|x\, y| < 2^{\mu+e-2}$. Then the algorithm* $\mathbf{LongMul}_e(x, y)$ *computes a pair $(h, l) \in \mathbb{F}^2$ with $h \in \mathbb{Z}\, 2^e$, $|h + l - x\, y| < 2^{e-\mu}$, and $|l| \leqslant 2^e$. In addition, if $x\, y \in \mathbb{Z}\, 2^{e-\mu}$, then $h + l = x\, y$ and $|l| < 2^e$.*

Proof. In a similar way as in the proof of Proposition 2, one shows that $h \in \mathbb{Z}\, 2^e$ and $|h - x\, y| < 2^e$. It follows that $|l| \leqslant 2^e$ and $|l - (x\, y - h)| < 2^{e-\mu}$. If $x\, y \in \mathbb{Z}\, 2^{e-\mu}$, then the subtraction $x\, y - h$ is exact, whence $|l| = |h - x\, y| < 2^e$. □

PROPOSITION 8. *Let $x \in \mathcal{F}_{p,2;B}$ and $y \in \mathcal{F}_{p,2;C}$ with $|x_0| < B$, $|y_0| \leqslant C$ and $B\,C \leqslant 2^{\delta-2}$. Then we have $r = \mathbf{Multiply}(x, y) \in \mathcal{F}_{p,2;2BC+2}$ and $|r - x\,y| < (B\,C + 2)\,2^{-2p}$.*

Proof. Let us write l_{I}, l_{II} and l_{III} for the successive values of l taken during the execution of the algorithm. Since $|x_0\,y_0| < B\,C \leqslant 2^{\delta-2}$, Proposition 7 implies that $h \in \mathbb{Z}\,2^{-p}$, $|l_{\mathrm{I}}| < 2^{-p}$, and $h + l_{\mathrm{I}} = x_0\,y_0$. Using that $|x_1| < B\,2^{-p}$ and $|y_1| < C\,2^{-p}$, we next obtain $|l_{\mathrm{II}}| \leqslant (B\,C + 1 + 2^{-p})\,2^{-p}$ and $|l_{\mathrm{III}}| \leqslant (2\ B\ C\ +\ 1\ +\ 2\ \cdot\ 2^{-p})\ 2^{-p}$. We also get that $|l_{\mathrm{II}} - (l_{\mathrm{I}} + x_0\ y_1)| \leqslant 2^{-2p}$ and $|l_{\mathrm{III}} - (l_{\mathrm{II}} + x_1\,y_0)| \leqslant 2^{-2p}$. Finally we obtain:

$$\begin{aligned} |r - x\,y| &\leqslant |r - x_0\,y_0 - x_0\,y_1 - x_1\,y_0 - x_1\,y_1| \\ &\leqslant |h + l_{\mathrm{I}} - x_0\,y_0| + |l_{\mathrm{II}} - l_{\mathrm{I}} - x_0\,y_1| + |l_{\mathrm{III}} - l_{\mathrm{II}} - x_1\,y_0| + |x_1\,y_1| \\ &< 2 \cdot 2^{-2p} + B\,C\,2^{-2p}. \end{aligned}$$

□

2.6. C++ implementation inside MATHEMAGIX

For our C++ implementation inside MATHEMAGIX, we introduced the template type

```
template<typename C, typename V> fixed_quadruple;
```

The parameter `C` corresponds to a built-in numeric type such as `double`. The parameter `V` is a "traits" type (called the "variant" in MATHEMAGIX) and specifies the precision p (see [18] for details). When instantiating for `C=double` and the default variant `V`, the type `fixed_quadruple<double>` corresponds to $\mathcal{F}_{p,2}$ with $p = 48$ (see the file `numerix/fixed_quadruple.hpp`).

Since all algorithms from this section only use basic arithmetic instructions (add, subtract, multiply, FMA, FMS) and no branching, they admit straightforward SIMD analogues. MATHEMAGIX features a very systematic support for SIMD types and operations [18]. This provides us with SIMD versions for multiple precision fixed point arithmetic simply by instantiating the above template types for a suitable numeric SIMD class, such as `avx_double` from `numerix/avx.hpp`.

3. FAST FOURIER TRANSFORMS

In this section we describe how to use the fixed point arithmetic functions to compute FFTs. The number of nail bits δ is adjusted to perform a single normalization stage per butterfly. In the next paragraphs we follow the classical Cooley and Tukey in place algorithm [6].

3.1. Complex arithmetic

We implement complex analogues **ComplexNormalize**, **ComplexAdd**, **ComplexSubtract** and **ComplexMultiply** of **Normalize**, **Add**, **Subtract** and **Multiply** in a naive way. We have fully specified **ComplexMultiply** below, as an example. The three other routines proceed componentwise, by applying the real counterparts on the real and imaginary parts. Here $\Re u$ and $\Im u$ represent the real and imaginary parts of u respectively. The norm $\|u\|_\infty \in \mathbb{R}$ of the complex number u is defined as $\max(|\Re u|, |\Im u|)$.

Algorithm ComplexMultiply(u, v)

$a := \mathbf{Multiply}(\Re u, \Re v)$
$b := \mathbf{Multiply}(\Im u, \Im v)$
$c := \mathbf{Multiply}(\Re u, \Im v)$
$d := \mathbf{Multiply}(\Im u, \Re v)$
return $\mathbf{Subtract}(a, b) + \mathbf{Add}(c, d)\,\mathrm{i}$

3.2. Butterflies

The basic building block for fast discrete Fourier transforms is the complex butterfly operation. Given a pair $(u, v) \in \mathbb{C}^2$ and a precomputed root of unity $\omega \in \mathbb{C}$, the butterfly operation computes a new pair $(u + v\,\omega, u - v\,\omega)$. For inverse transforms, one rather computes the pair $(u + v, (u - v)\,\omega)$ instead. For simplicity, and without loss of generality, we may assume that the approximation of ω in $\mathcal{F}_{p,2;1}[\mathrm{i}]$ satisfies $\|\omega_0\|_\infty \leqslant 1$.

Algorithm DirectButterfly(u, v, ω)

$z := \mathbf{ComplexMultiply}(\omega, v)$
$u' := \mathbf{ComplexAdd}(u, z)$
$v' := \mathbf{ComplexSubtract}(u, z)$
$\tilde{u} := \mathbf{ComplexNormalize}(u')$
$\tilde{v} := \mathbf{ComplexNormalize}(v')$
return $(\tilde{u}, \tilde{v})$

Algorithm InverseButterfly(u, v, ω)

$u' := \mathbf{ComplexAdd}(u, v)$
$z := \mathbf{ComplexSubtract}(u, v)$
$v' := \mathbf{ComplexMultiply}(\omega, z)$
$\tilde{u} := \mathbf{ComplexNormalize}(u')$
$\tilde{v} := \mathbf{ComplexNormalize}(v')$
return $(\tilde{u}, \tilde{v})$

PROPOSITION 9. *Let u, v, $\omega \in \mathcal{F}_{p,2;1}[\mathrm{i}]$ with $\|u_0\|_\infty < 1$, $\|v_0\|_\infty < 1$, $\|\omega_0\|_\infty \leqslant 1$ and assume that $\delta \geqslant 4$. Then $(\tilde{u}, \tilde{v}) = \mathbf{DirectButterfly}(u, v, \omega) \in \mathcal{F}_{p,2;1}[\mathrm{i}]^2$ and $\|\tilde{u} - (u + v\,\omega)\|_\infty$, $\|\tilde{v} - (u - v\,\omega)\|_\infty < 7 \cdot 2^{-2p}$.*

Proof. Let a, b, c, d be as in **ComplexMultiply** with $u = \omega$ and v as arguments. From Proposition 8, it follows that $a \in \mathcal{F}_{p,2;4}$ and $|a - \Re\omega\,\Re v| < 3 \cdot 2^{-2p}$, and we have similar bounds for b, c and d. From Proposition 6, we next get $z \in \mathcal{F}_{p,2;8+2^{-p}}[\mathrm{i}]$ and $\|z - \omega\,v\|_\infty < 5 \cdot 2^{-2p}$. Applying Proposition 6 again, we obtain $u', v' \in \mathcal{F}_{p,2;9+2\cdot 2^{-p}}[\mathrm{i}]$, $\|u' - (u + v\,\omega)\|_\infty < 6 \cdot 2^{-2p}$ and $\|v' - (u - v\,\omega)\|_\infty < 6 \cdot 2^{-2p}$. The conclusion now follows from Proposition 5. □

PROPOSITION 10. *Let u, v, $\omega \in \mathcal{F}_{p,2;1}[\mathrm{i}]$ with $\|u_0\|_\infty < 1$, $\|v_0\|_\infty < 1$, $\|\omega_0\|_\infty \leqslant 1$ and assume that $\delta \geqslant 4$. Then $(\tilde{u}, \tilde{v}) = \mathbf{InverseButterfly}(u, v, \omega) \in \mathcal{F}_{p,2;1}[\mathrm{i}]^2$ and $\|\tilde{u} - (u + v)\|_\infty$, $\|\tilde{v} - (u - v)\,\omega\|_\infty < 9 \cdot 2^{-2p}$.*

Proof. Proposition 6 yields $u', z \in \mathcal{F}_{p,2;2+2^{-p}}[\mathrm{i}]$ and $\|u' - (u + v)\|_\infty < 2^{-2p}$, and $\|z - (u - v)\|_\infty < 2^{-2p}$. Proposition 5 then gives us $\tilde{u} \in \mathcal{F}_{p,2;1}[\mathrm{i}]$, $\|\tilde{u} - (u + v)\|_\infty < 2 \cdot 2^{-2p}$. Let a, b, c, d be as in **ComplexMultiply** with ω and z as arguments. From Proposition 8, it follows that $a \in \mathcal{F}_{p,2;6+2\cdot 2^{-p}}$, $|a - \Re\omega\,\Re(u - v)| < (4 + 2 \cdot 2^{-p}) \cdot 2^{-2p}$, and we have similar bounds for b, c and d. From Proposition 6 again, we get $v' \in \mathcal{F}_{p,2;12+5\cdot 2^{-p}}[\mathrm{i}]$ and $\|v' - (u - v)\,\omega\|_\infty < (8 + 5 \cdot 2^{-p}) \cdot 2^{-2p}$. Finally Proposition 5 gives us $\tilde{v} \in \mathcal{F}_{p,2;1}[\mathrm{i}]$, $\|\tilde{v} - (u - v)\,\omega\|_\infty < 9 \cdot 2^{-2p}$. □

For reason of space, we will not go into more details concerning the total precision loss in the FFT, which is of the order of $\log n$ bits at size n. We refer the reader to the analysis in [14], which can be adapted to the present context thanks to the above propositions.

3.3. Timings

Throughout this article, timings are measured on a platform equipped with an INTEL CORE™ *i7-4770* CPU @ 3.40 GHz and 8 GB of *1600 MHz DDR3*. It runs the JESSIE GNU DEBIAN® operating system with a LINUX® kernel version 3.14 in 64 bit mode. We measured average timings, while taking care to avoid CPU throttling issues. We compile with GCC [10] version 4.9.1.

ν	8	9	10	11	12	13	14	15	16
`double`	0.43	0.94	2.3	5.4	15	34	85	190	400
`long double`	6.1	14	31	70	153	332	720	1600	3400
`__float128`	89	205	463	1000	2300	5100	11000	24000	51000

Table 1. FFTW 3.3.4 in size $n = 2^\nu$, in micro-seconds.

For the sake of comparison, Table 1 displays timings to compute FFTs over complex numbers with the FFTW library version 3.3.4 [9]. Timings are obtained *via* the command `test/bench` bundled with the library. The row `double` corresponds to the standard double precision in C and the configuration options `--enable-avx` and `--enable-fma`. The row `long double` is obtained in the same way with the configuration options `--enable-long-double`; on current platform this corresponds to using the X87 instructions on 80 bits wide floating point numbers. The last row means using the IEEE compliant quadruple precision type `__float128` provided by the compiler, and configured with `--enable-fma` and `--enable-quad-precision`.

ν	8	9	10	11	12	13	14	15	16
`double`	0.54	1.1	2.5	6.8	16	37	85	220	450
`long double`	9.5	21	48	110	230	500	1100	2300	5000
`__float128`	94	220	490	1100	2400	5300	11000	25000	53000
`quadruple`	5.0	11	25	55	120	260	570	1200	2600
`fixed_quadruple`	2.8	6.4	15	33	71	160	380	820	1700
MPFR (113 bits)	270	610	1400	3100	6800	15000	33000	81000	230000

Table 2. MATHEMAGIX in size $n = 2^\nu$, in micro-seconds.

Our MATHEMAGIX implementations of the algorithm in this paper are available from revision 9621. Timings are reported in Table 2. We have implemented the *split-radix* algorithm [19]. The configuration process of MATHEMAGIX automatically detects and activates AVX2 and FMA instruction sets. Roots of unity and necessary twiddle factors are precomputed once with full precision by using MPFR version 3.1.2 based on GMP version 6.0.0.

The three first rows concern the same built-in numerical types as in the previous table. The row `quadruple` makes use of our own non IEEE compliant implementation of the algorithms of [23], sometimes called *double-double* arithmetic (see file `numerix/quadruple.hpp`). The row `fixed_quadruple` corresponds to the new algorithms of this section with nail bits. In the rows `double`, `quadruple`, and `fixed_quadruple`, the computations fully benefit from AVX and FMA instructions. Finally the row MPFR contains timings obtained when using MPFR floating point numbers with bit precision set to 113. At this precision, our implementation involves an overhead due to the fact that the MPFR type `mpfr_t` is wrapped into a C++ class with reference counters (see file `numerix/floating.hpp`).

Let us mention that our type `quadruple` requires 5 arithmetic operations (counting FMA and FMS for 1) for a product, and 11 for an addition or subtraction. A direct butterfly thus amounts to 86 elementary operations. On the other hand, our type `fixed_quadruple` needs only 48 such operations. It is therefore satisfying to observe that these estimates are reflected in practice for small sizes, where the cost of memory caching is negligible.

4. GENERALIZATIONS TO HIGHER PRECISIONS

The representation with nail bits and the algorithms designed so far for doubled precision can be extended to higher precisions, as explained in the next paragraphs.

4.1. Representation and normalization

Given $1 \leqslant B \leqslant 2^\delta$ and an integer $k \geqslant 2$, we denote by $\mathcal{F}_{p,k;B}$ the set of numbers of the form

$$x \;=\; x_0 + \cdots + x_{k-1},$$

where $x_0, \ldots, x_{k-1} \in \mathbb{F}$ are such that

$$\begin{aligned} x_i &\in \mathbb{Z}\, 2^{-(i+1)p} && \text{for } 0 \leqslant i < k-1 \\ |x_i| &< B\, 2^{-ip} && \text{for } 1 \leqslant i < k. \end{aligned}$$

We write $x = [x_0, ..., x_{k-1}]$ for numbers of the above form and abbreviate $\mathcal{F}_{p,k} = \mathcal{F}_{p,k;2^\delta}$. Numbers in $\mathcal{F}_{p,k;1}$ are said to be in *normal form*. Of course, we require that $(k-1)\,p$ is smaller than $-E_{\min} - \mu$. For this, we assume that $k \leqslant 19$. The normalization procedure generalizes as follows (of course, the loop being unrolled in practice):

Algorithm Normalize(x)

$r_{k-1} := x_{k-1}$
for i **from** $k-1$ **down to** 1 **do**
  $c_i := \mathbf{Split}_{-ip}(r_i)$
  $\tilde{x}_i := \circ(r_i - c_i)$
  $r_{i-1} := \circ(x_{i-1} + c_i)$
$\tilde{x}_0 := r_0$
return $[\tilde{x}_0, ..., \tilde{x}_{k-1}]$

Proposition 11. *Given a fixed point number $x \in \mathcal{F}_{p,k;B}$ with $|x_0| < 2^\delta - 2^{\delta-p}$, $B \leqslant 2^\delta - 2^{\delta-p}$, the algorithm* **Normalize** *returns $\tilde{x} \in \mathcal{F}_{p,k;1}$ with $|\tilde{x} - x| < 2^{-kp-\delta}$.*

Proof. During the first step of the loop, when $i = k-1$, by Proposition 2 we have $|r_{k-1} - c_{k-1}| < 2^{-(k-1)p}$ and $c_{k-1} \in \mathbb{Z}\, 2^{-(k-1)p}$. Using the fact that $|r_{k-1}| < 2^{\delta-(k-1)p}$, this entails $|c_{k-1}| < (2^\delta + 1)\, 2^{-(k-1)p}$ and therefore $|c_{k-1}| \leqslant 2^{\delta-(k-1)p}$. Hence $|x_{k-2} + c_{k-1}| \leqslant |x_{k-2}| + |c_{k-1}| < 2^{\delta-(k-2)p}$, so that $r_{k-2} = \circ(x_{k-2} + c_{k-1}) = x_{k-2} + c_{k-1}$. Using similar arguments for $i = k-2, ..., 1$, we obtain $\tilde{x} \in \mathcal{F}_{p,k;1}$ and $|\tilde{x} - x| \leqslant |\tilde{x}_{k-1} - (r_{k-1} - c_{k-1})|$. Finally, since $|r_{k-1} - c_{k-1}| < 2^{-(k-1)p}$ we have $|\tilde{x}_{k-1} - (r_{k-1} - c_{k-1})| < 2^{-(k-1)p-\mu}$, which concludes the proof. □

4.2. Ring operations

The generalizations of **Add** and **Subtract** are straightforward: $\mathbf{Add}(x,\, y) = [\circ(x_0 + y_0),\, ...,\, \circ(x_{k-1} + y_{k-1})]$, and $\mathbf{Subtract}(x, y) = [\circ(x_0 - y_0), ..., \circ(x_{k-1} - y_{k-1})]$.

Proposition 12. *Let $x \in \mathcal{F}_{p,k;B}$ and $y \in \mathcal{F}_{p,k;C}$ with $S := B + C + 2^{-p} \leqslant 2^\delta$. If $|x_0 + y_0| < 2^\delta$, then $a = \mathbf{Add}(x,\, y) \in \mathcal{F}_{p,k;S}$ with $|a - (x + y)| < 2^{-kp}$. If $|x_0 - y_0| < 2^\delta - 2^{\delta-p}$, then $b = \mathbf{Subtract}(x, y) \in \mathcal{F}_{p,k;S}$ with $|a - (x - y)| < 2^{-kp}$.*

Proof. Since both x_i and y_i are integer multiples of 2^{-ip} for $0 \leqslant i \leqslant k-2$, the additions $a_i = x_i + y_i$ are exact. Furthermore, the exponents of x_{k-1} and y_{k-1} are both strictly bounded by $\delta - (k-1)\,p$. Consequently, $|a - (x + y)| \leqslant |a_{k-1} - (x_{k-1} + y_{k-1})| < 2^{\delta-(k-1)p-\mu} = 2^{-kp}$ and $|a_{k-1}| < |x_{k-1} + y_{k-1}| + 2^{-kp} \leqslant |x_{k-1}| + |y_{k-1}| + 2^{-kp} < S\, 2^{-(k-1)p}$. The statement for the subtraction follows by replacing y by $-y$. □

Multiplication can be generalized as follows (all loops again being unrolled in practice):

Algorithm Multiply(x, y)

$(r_0, r_1) := \mathbf{LongMul}_{-p}(x_0, y_0)$
for i **from** 1 **to** $k-2$ **do**
  $(h, r_{i+1}) := \mathbf{LongMul}_{-(i+1)p}(x_0, y_i)$
  $r_i := \circ(r_i + h)$
  for j **from** 1 **to** i **do**
    $(h, l) := \mathbf{LongMul}_{-(i+1)p}(x_j, y_{i-j})$
    $r_i := \circ(r_i + h)$
    $r_{i+1} := \circ(r_{i+1} + l)$
for i **from** 0 **to** $k-1$ **do**
  $r_{k-1} := \circ(x_i\, y_{k-1-i} + r_{k-1})$
return $[r_0, ..., r_{k-1}]$

PROPOSITION 13. *Let $x \in \mathcal{F}_{p,k;B}$ and $y \in \mathcal{F}_{p,k;C}$ with $|x_0| < B$, $|y_0| \leqslant C$, $B\,C \leqslant 2^{\delta-2}$, and $S := k\,(B\,C + 1 + 2^{-p}) - 1 \leqslant 2^{\delta}$. Then $r = \mathbf{Multiply}(x, y) \in \mathcal{F}_{p,k;S}$ with $|r - x\,y| < S\,2^{-kp}$.*

Proof. Let us write $(h_{i,j},\, l_{i,j})$ for the pair returned by $\mathbf{LongMul}_{-(i+j+1)p}(x_i,\, y_j)$, for $0 \leqslant i + j \leqslant k-2$. Since $B\,C \leqslant 2^{\delta-2}$, Proposition 7 implies that $h_{i,j} \in \mathbb{Z}\,2^{-(i+j+1)p}$, $|l_{i,j}| < 2^{-(i+j+1)p}$, and $h_{i,j} + l_{i,j} = x_i\,y_j$. At the end of the algorithm we have $r_0 = h_{0,0}$ and $r_t = \sum_{i+j=t} h_{i,j} + \sum_{i+j=t-1} l_{i,j}$ for all $1 \leqslant t \leqslant k-2$, and therefore $|r_t| < (t+1)\,B\,C\,2^{-tp} + t\,2^{-tp} \leqslant S\,2^{-tp}$ for all $0 \leqslant t \leqslant k-2$. In particular no overflow occurs and these sums are all exact. Let $s_0 = \sum_{i+j=k-2} l_{i,j}$ represent the value of r_{k-1} before entering the second loop. Then let $s_{i+1} = \circ(x_i\,y_{k-1-i} + s_i)$ for $0 \leqslant i \leqslant k-1$, so that $r_{k-1} = s_k$ holds at the end of the algorithm. We have $|s_0| < (k-1)\,2^{-(k-1)p}$, and $|s_i| < (i\ B\ C\ +\ k\ -\ 1\ +\ i\ 2^{-p})\ 2^{-(k-1)p}$ for all $0 \leqslant i \leqslant k$. For the precision loss, from $|s_{i+1} - (x_i\,y_{k-1-i} + s_i)| \leqslant 2^{-kp}$, we deduce that

$$\begin{aligned} |r - x\,y| &\leqslant \sum_{i=0}^{k-1} |s_{i+1} - (x_i\,y_{k-1-i} + s_i)| + \sum_{i+j \geqslant k} |x_i\,y_j| \\ &< k\,2^{-kp} + B\,C \sum_{i+j \geqslant k} 2^{-(i+j)p}. \end{aligned}$$

The proof follows from $\sum_{i+j\geqslant k} 2^{-(i+j)p} = 2^{-kp} \sum_{i=0}^{k-2} (k-1-i)\,2^{-ip} = \frac{k - 1 - k\,2^{-p} + 2^{-kp}}{(1-2^{-p})^2}\,2^{-kp} \leqslant (k-1)(1+2^{-p})\,2^{-kp}$, while using that $2 \leqslant k \leqslant 19$ and $p \geqslant 5$. □

4.3. Fast Fourier Transforms

We can use the same butterfly implementations as in Section 3.2. The following generalization of Proposition 9 shows that we may take $\delta = 4$ as long as $k \leqslant 4$, and $\delta = 5$ as long as $k \leqslant 8$ for the direct butterfly operation.

PROPOSITION 14. *Let $u, v, \omega \in \mathcal{F}_{p,k;1}[\mathrm{i}]$ with $\|u\|_\infty < 1$, $\|v\|_\infty < 1$, $\|\omega_0\|_\infty \leqslant 1$ and assume that $\delta \geqslant 4$ and $4\,k - 1 + (2\,k+2)\,2^{-p} < 2^{\delta} - 2^{\delta-p}$. Then $(\tilde{u}, \tilde{v}) = \mathbf{DirectButterfly}(u, v, \omega) \in \mathcal{F}_{p,k;1}[\mathrm{i}]^2$ and $\|\tilde{u} - (u + v\,\omega)\|_\infty, \|\tilde{v} - (u - v\,\omega)\|_\infty < (2\,k+3)\,2^{-kp}$.*

Proof. Let a, b, c, d be as in **ComplexMultiply** with $u = \omega$ and v as arguments. From Proposition 13, it follows that $a \in \mathcal{F}_{p,k;2k-1+k2^{-p}}$ and $|a - \Re\omega\,\Re v| < 2\,k\,2^{-kp}$, and we have similar bounds for b, c and d. From Proposition 12, we next get $z \in \mathcal{F}_{p,k;4k-2+(2k+1)2^{-p}}[\mathrm{i}]$ and $\|z - \omega\,v\|_\infty < (2\,k+1)\,2^{-kp}$. Applying Proposition 12 again, we obtain $u', v' \in \mathcal{F}_{p,k;4k-1+(2k+2)2^{-p}}[\mathrm{i}]$, $\|u' - (u + v\,\omega)\|_\infty < (2\,k+2)\,2^{-kp}$ and $\|v' - (u - v\,\omega)\|_\infty < (2\,k+2)\,2^{-kp}$. The conclusion now follows from Proposition 11. □

The following generalization of Proposition 9 shows that we may take $\delta = 4$ as long as $k \leqslant 2$, $\delta = 5$ as long as $k \leqslant 5$, and $\delta = 6$ as long as $k \leqslant 10$ for the inverse butterfly operation.

PROPOSITION 15. *Let $u, v, \omega \in \mathcal{F}_{p,k;1}[\mathrm{i}]$ with $\|u_0\|_\infty < 1$, $\|v_0\|_\infty < 1$, $\|\omega_0\|_\infty \leqslant 1$ and assume that $\delta \geqslant 4$ and $6\,k - 2 + (4\,k+1)\,2^{-p} < 2^{\delta} - 2^{\delta-p}$. Then $(\tilde{u}, \tilde{v}) = \mathbf{InverseButterfly}(u, v, \omega) \in \mathcal{F}_{p,k;1}[\mathrm{i}]^2$ and $\|\tilde{u} - (u+v)\|_\infty, \|\tilde{v} - (u-v)\,\omega\|_\infty < 3\,k\,2^{-kp}$.*

Proof. Proposition 12 yields u', $z \in \mathcal{F}_{p,k;2+2^{-p}}[\mathrm{i}]$ and $\|u' - (u+v)\|_\infty < 2^{-kp}$, and $\|z - (u-v)\|_\infty < 2^{-kp}$. Proposition 11 then gives us $\tilde{u} \in \mathcal{F}_{p,k;1}[\mathrm{i}]$, $\|\tilde{u} - (u+v)\|_\infty < 2 \cdot 2^{-kp}$. Let a, b, c, d be as in **ComplexMultiply** with ω and z as arguments. From Proposition 13, it follows that $a \in \mathcal{F}_{p,k;3k-1+2k2^{-p}}$, $|a - \Re\omega\,\Re(u-v)| < (3\,k - 1 + (2\,k+1)\,2^{-p})\,2^{-kp}$, and we have similar bounds for b, c and d. From Proposition 12 again, we get $v' \in \mathcal{F}_{p,k;6k-2+(4k+1)2^{-p}}[\mathrm{i}]$ and $\|v' - (u-v)\,\omega\|_\infty < (3\,k - 1 + (2\,k+2)\,2^{-p})\,2^{-kp}$. Finally Proposition 11 gives us $\tilde{v} \in \mathcal{F}_{p,k;1}[\mathrm{i}]$, $\|\tilde{v} - (u-v)\,\omega\|_\infty < (3\,k - 1 + (2\,k+3)\,2^{-p})\,2^{-2p}$. □

Remark 16. Within Mathemagix, inverse butterflies are systematically used in all inverse FFT implementations, so as to benefit from in-place algorithms without *bit reverse copies*. Nevertheless, if the precision becomes of critical importance, then we recommend to use the direct FFT transform with ω replaced by ω^{-1}. In our code for $k=3$ and $k=4$, we have decided to keep $\delta=4$ by performing one additional normalization of z in **InverseButterfly**.

Remark 17. The following strategy sometimes makes it possible to decrease δ by one: instead of normalizing the entries of an FFT to be of norm $\|\cdot\|_\infty < 1$, we rather normalize them to be of norm $\|\cdot\|_\infty < 1 - C\,k\,2^{-p}$ for some small constant C. This should allow us to turn the conditions $4\,k-1+(2\,k+2)\,2^{-p} < 2^\delta - 2^{\delta-p}$ and $6\,k-2+(4\,k+1)\,2^{-p} < 2^\delta - 2^{\delta-p}$ in the Propositions 14 and 10 into $4\,k-1 \leqslant 2^\delta$ and $6\,k-2 \leqslant 2^\delta$. One could also investigate what happens if we additionally assume $\|\cdot\|_\infty < 1/2 - C\,k\,2^{-p}$.

It is interesting to compute the number of operations in $\mathbb{F}$ which are needed for our various fixed point operations, the allowed operations in $\mathbb{F}$ being addition, subtraction, multiplication, FMA and FMS. For larger precisions, it may also be worth it to use Karatsuba's method [20] for the complex multiplication, meaning that we compute $\Re u\,\Im v + \Im v\,\Re u$ using the formula $(\Re u + \Im u)\,(\Re v + \Im v) - \Re u\,\Re v - \Im u\,\Im v$. This saves one multiplication at the expense of three additions/subtractions and an additional increase of δ. The various operation counts are summarized in Table 3.

k	1	2	3	4	5	6	$k \geqslant 7$
Normalize	0	4	8	12	16	20	$4\,k-4$
Add/subtract	1	2	3	4	5	6	k
Multiply	1	5	15	30	50	75	$5\binom{k}{2}$
Butterfly	8	48	110	192	294	416	$10\,k^2+12\,k-16$
Butterfly-Karatsuba	10	49	104	174	259	359	$15/2\,k^2+35/2\,k-16$

Table 3. Operation counts in terms of basic arithmetic operations in $\mathbb{F}$.

4.4. Timings

For our C++ implementation inside Mathemagix we introduced two additional template types

```
template<typename C, typename V> fixed_hexuple;
template<typename C, typename V> fixed_octuple;
```

with similar semantics as `fixed_quadruple`. When instantiating for `C=double` and the default variant `V`, these types correspond to $\mathcal{F}_{p,3}$ and $\mathcal{F}_{p,4}$ with $p = 48$. In the future, we intend to implement a more general template type which also takes k as a parameter.

Table 4 shows our timings for FFTs over the above types. Although the butterflies of the split-radix algorithm are a bit different from the classical ones, it is pleasant to observe that our timings roughly reflect operation counts of Table 3. For comparison, let us mention that the time necessary to perform (essentially) the same computations with MPFR are roughly the same as in Table 2, even in octuple precision. In addition (c.f. Table 2), we observe that the performance of our `fixed_hexuple` FFT is of the same order as FFTW's `long double` implementation.

ν	8	9	10	11	12	13	14	15	16
`double`	0.54	1.1	2.5	6.8	16	37	85	220	450
`fixed_quadruple`	2.8	6.4	15	33	71	160	380	820	1700
`fixed_hexuple`	7.6	17	38	84	180	400	870	1800	3900
`fixed_octuple`	18	42	93	200	450	980	2100	4500	9600

Table 4. Mathemagix in size $n=2^\nu$, in micro-seconds.

Thanks to the genericity of our C++ implementation, it is possible to directly compute FFTs on a SIMD type such as `fixed_hexuple<avx_double>`, which corresponds to packing four elements of type `fixed_hexuple<double>` into an AVX data type. Combined to classical cache-friendly approaches, this allows to compute multi-dimensional FFTs in an efficient way, which closer reflects operation counts of Table 3. When computing an FFT, say over `fixed_hexuple<double>`, the input data is in fact packed in a way that most of the computation reduces to one FFT over `fixed_hexuple<avx_double>` of size divided by four. Nevertheless, some of the C++ code needs to be fine-tuned in order to fully benefit from SIMD instructions, which turns out to constitute an important part of the implementation work.

5. Variants and perspectives

Polynomial representations. In this paper, we have chosen to represent multi-precision fixed point numbers as sums $x = x_0 + \cdots + x_{k-1}$. Alternatively, we may regard such numbers as polynomials

$$x = x_0 + x_1\, 2^{-p} + \cdots + x_{k-1}\, 2^{-(k-1)p}$$

in the "indeterminate" 2^{-p}, with $x_0, \ldots, x_{k-2} \in \mathbb{Z}\, 2^{-p}$ and $|x_0|, \ldots, |x_{k-1}| < 2^{\delta}$. It can be checked that the algorithms of this paper can be adapted to this setting with no penalty (except for multiplication which may require one additional instruction). Roughly speaking, every time that we need to add a "high" part h and a "low" part l of two long products, it suffices to use a fused-multiply-add instruction $l\, 2^p + h$ instead of a simple addition $l + h$.

For naive multiplication, the polynomial representation may become slightly more efficient for $k \geqslant 3$, when the hardware supports an SIMD instruction for rounding floating point numbers to the nearest integer. In addition, when k becomes somewhat larger, say $k \geqslant 8$, then the polynomial representation makes it possible to use pair-odd Karatsuba multiplication [12, 16] without any need for special carry handling (except that δ has to be chosen a bit larger). Finally, the restriction $(k-1)\, p \leqslant -E_{\min} - \mu$ is no longer necessary, so the approach can in principle be extended to much larger values of k.

Integer arithmetic. The current focus of vendors of wide SIMD arithmetic is on fast floating point arithmetic, whereas integer arithmetic is left somewhat behind. For instance, INTEL's AVX-512 technology features efficient arithmetic (including FMA) on vectors of eight double precision numbers. On the other hand, SIMD integer multiplications are limited to 16 bit and 32 bit integers.

In order to develop an efficient GMP-style library for SIMD multiple precision integers, it would be useful to have an instruction for multiplying two vectors of 64 bit integers and obtain the lower and higher parts of the result in two other vectors of 64 bit integers. For multiple precision additions and subtractions, it is also important to have vector variants of the "add with carry" and "subtract with borrow" instructions.

Assuming the above instructions, a $64\,k$ bit unsigned truncated integer multiplication can be done in SIMD fashion using approximately $4 \binom{k}{2}$ instructions. Furthermore, in this setting, there is no need for normalizations. However, signed multiplications are somewhat harder to achieve. The best method for performing a signed integer multiplication $x\, y$ with $|x|, |y| < 2^{64k-1}$ is probably to perform the unsigned multiplication $(x + C)\,(y + C)$ with $C = 2^{64k-1}$ and use the fact that $x\, y = (x + C)\,(y + C) - (x + y)\, C + C^2$. We expect that the biggest gain of using integer arithmetic comes from the fact that we achieve a $64\,k$ instead of a $p\,k$ bit precision (for small k, we typically have $p = 48$).

One potential problem with GMP-style SIMD arithmetic is that asymptotically faster algorithms, such as Karatsuba multiplication (and especially the even-odd variant), are harder to implement. Indeed, we are not allowed to ressort to branching for carry handling. One remedy would be to use expansions $x = x_0 + x_1\, 2^{64s} + \cdots + x_{k/s-1}\, 2^{64k-64s}$ with coefficients $0 \leqslant x_i < 2^{64s-\delta}$ for a suitable number δ of nail bits and a suitable "multiplexer" $s > 1$.

Faster splitting. An interesting question is whether hardware support for certain specific operations might speed up the fixed point arithmetic in this paper. One operation which is relatively expensive is normalization. It would be nice to have an instruction which takes $x \in \mathbb{F}$ and $e \in \{E_{\min}, \ldots, E_{\max}\}$ on input and which computes numbers $h, l \in \mathbb{F}$ with $x = h + l$, $h \in \mathbb{Z}\, 2^e$ and $|l| < \mathbb{Z}\, 2^e$. In that case, normalization would only require two instructions instead of four. Besides speeding up the naive algorithms, it would also become less expensive to do more frequent normalizations, which makes the use of asymptotically more efficient multiplication algorithms more tractable at lower precisions and with fewer nail bits. More efficient normalization is also important for efficient shifting of mantissas, which is the main additional ingredient for the implementation of multiple precision arithmetic.

Asymptotically efficient algorithms. A theoretically important question concerns the asymptotic complexity of FFTs at large precisions. Let $\mathsf{I}(b)$ denote the bit complexity of b bit integer multiplication. It has recently been proved [14] that $\mathsf{I}(b) = O(b \log b\, 8^{\log^* b})$, where $\log^* b = \min_k \left\{ \log \overset{k\times}{\ldots} \log b \leqslant 1 \right\}$. Under mild assumptions on b and n it can also be shown [14] that an FFT of size n and bit precision b can be performed in time $\mathsf{F}_b(n) = O(\mathsf{I}(b\, n))$. For $b \leqslant n$ and large n, this means that $\mathsf{F}_b(n)$ scales linearly with b. In a future paper, we intend to investigate the best possible (and practically achievable) constant factor involved in this bound.

Bibliography

[1] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, EC-10(3):389–400, 1961.

[2] D. H. Bailey, R. Barrio, and J. M. Borwein. High precision computation: Mathematical physics and dynamics. *Appl. Math. Comput.*, 218:10106–10121, 2012.

[3] R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Software*, 4:57–70, 1978.

[4] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[5] S. Chakraborty, U. Frisch, W. Pauls, and Samriddhi S. Ray. Nelkin scaling for the Burgers equation and the role of high-precision calculations. *Phys. Rev. E*, 85:015301, 2012.

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

[7] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18(3):224–242, 1971.

[8] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Software*, 33(2), 2007. Software available at http://www.mpfr.org.

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

[10] GCC, the GNU Compiler Collection. Software available at http://gcc.gnu.org, from 1987.

[11] T. Granlund et al. GMP, the GNU multiple precision arithmetic library. http://www.swox.com/gmp, from 1991.

[12] G. Hanrot and P. Zimmermann. A long note on Mulders' short product. *J. Symbolic Comput.*, 37(3):391–401, 2004.

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

[14] D. Harvey, J. van der Hoeven, and G. Lecerf. Even faster integer multiplication, 2014. http://arxiv.org/abs/1407.3360.

[15] Yozo Hida, Xiaoye S. Li, and D. H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proc. 15th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE, 2001.

[16] J. van der Hoeven and G. Lecerf. On the complexity of multivariate blockwise polynomial multiplication. In J. van der Hoeven and M. van Hoeij, editors, *Proc. 2014 International Symposium on Symbolic and Algebraic Computation*, pages 211–218. ACM, 2012.

[17] J. van der Hoeven, G. Lecerf, B. Mourrain, et al. Mathemagix, from 2002. http://www.mathemagix.org.

[18] J. van der Hoeven, G. Lecerf, and G. Quintin. Modular SIMD arithmetic in Mathemagix, 2014. http://hal.archives-ouvertes.fr/hal-01022383.

[19] S. G. Johnson and M. Frigo. A modified split-radix FFT with fewer arithmetic operations. *IEEE Trans. Signal Process.*, 55(1):111–119, 2007.

[20] A. Karatsuba and J. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[21] C. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR2005-38, LIP, ENS Lyon, 2005.

[22] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[23] T. Nagai, H. Yoshida, H. Kuroda, and Y. Kanada. Fast quadruple precision arithmetic library on parallel computer SR11000/J2. In *Computational Science - ICCS 2008, 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part I*, pages 446–455, 2008.

[24] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proc. 10th Symposium on Computer Arithmetic*, pages 132–145. IEEE, 1991.