Faster FFTs in medium precision

Joris van der Hoeven and Grégoire Lecerf
Laboratoire d’informatique de l’École polytechnique, UMR 7161 CNRS
9120 Palaiseau, France
{vdhoeven, lecerf}@lix.polytechnique.fr

Abstract—In this paper, we show how to speed up 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

I. INTRODUCTION

Multiple precision arithmetic [1] 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 [4] and MPFR [5] 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 [6], [7], 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. $\lfloor \log_2 |x| \rfloor \geq \lfloor \log_2 |y| \rfloor + 52$ 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 [6]–[8]. The approach has been further extended in [9] to higher precisions. Specific algorithms are also described in [10] for triple-double precision, and in [11] for quad-quad-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) [12], which is only numerically accurate if all input coefficients are of the same order of magnitude [13, Section 3.4]. 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. Our actual implementation of fixed point arithmetic relies on hardware floating point arithmetic. The working precision of our fixed point arithmetic will be of the form $kp$, where $k \in \{2, 3, \ldots, \}$, $p = \mu - \delta$ for a small integer $\delta$ (typically, $\delta = 4$), and $\mu$ 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 $\delta$ 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 [4], 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 [14].

Let \( F_k(n) \) denote the time spent to compute an FFT of length \( n \) at a precision of \( \pi \) bits. For an optimal implementation at precision \( k \mu \), we would need \( F_{k\mu}(n) \approx kF_{\mu}(n) \). However, the naive implementation of a product at precision \( k \mu \) requires at least \( (k+1)\mu \) machine multiplications, so \( F_{k\mu}(n) \approx (k+1)2F_{\mu}(n) \) 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++ libraries of MATHEMAGIX [15] (which is currently limited to a single thread). For small \( k \), our timings seem to indicate that \( F_{k\mu}(n) \approx 2(k+1)F_{\mu}(n) \). For \( k = 2 \), we compared our implementation with FFTW3 [16] (in which case we only obtained \( F_{2\mu}(n) \approx 200F_{\mu}(n) \) when using the built-in type __float128 of GCC) and a home-made double-double implementation along the same lines as [8] (in which case we got \( F_{2\mu}(n) \approx 10F_{\mu}(n) \)).

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 [17]. 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. Another interesting application of fast quadruple precision FFTs is large integer multiplication [1, Section 3.3.1].

Our paper is structured as follows. Section II contains a detailed presentation of fixed point arithmetic for the important precision \( 2p \). In Section III, we show how to apply this arithmetic to the computation of FFTs, and we provide timings. In Section IV, we show how our algorithms generalize to higher precisions \( kp \) with \( k \geq 2 \), 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 \( \circ(x \ast y) \) the rounding of \( x \ast y \) according to the chosen rounding mode. If \( e \) is the exponent of \( x \ast y \) and \( E_{\max} > e \geq E_{\min} + \mu \) (i.e. in absence of overflow and underflow), then we notice that \( |\circ(x \ast y) - x \ast y| < 2^{e-\mu} \).

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 \( \circ(xy + z) \) and \( \circ(xy - z) \) the roundings of \( xy + z \) and \( xy - z \) according to the chosen rounding mode.

## II. ALMOST QUADRUPLE PRECISION

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

### A. Representation of fixed point numbers

We denote by \( F_{p,2} \) 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

\[
x_0 \in \mathbb{Z}2^{-p} \quad (1)
\]

\[
|x_0| < 2^\delta \quad (2)
\]

\[
|x_1| < 2^{\delta - p}. \quad (3)
\]

It will be convenient to write \( x = [x_0, x_1] \) for numbers of the above form. Since \( F_{p,2} \) contains all numbers \( x = k2^{-2p} \) with \( k \in \mathbb{Z} \) and \( |x| < 1 \), this means that \( F_{p,2} \) can be thought of as a fixed point type of precision \( 2p \).

**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 \( x_1 \in \mathbb{Z}2^{-2p} \).

![Diagram](image)

**Figure 1.** Representation of the decomposition \( x = [x_0, x_1] = x_0 + x_1 \).

### B. 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 \((x)\)**

\[
a := \circ(x + 3/2 \cdot 2^{e+\mu}); \quad \text{return} \ \circ(a - 3/2 \cdot 2^{e+\mu})
\]

**Proposition 1** Given \( x \in \mathbb{F} \) and \( e \in \{E_{\min}, \ldots, E_{\max} - \mu\} \) such that \( |x| < 2^{e+\mu-2} \), the algorithm Split 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 \( 5/4 \cdot 2^{e+\mu} \leq a \leq 7/4 \cdot 2^{e+\mu} \).
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 $\tilde{x} = a - 3/2 \cdot 2^{e+\mu}$. 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 2** Assuming that the rounding mode is set towards zero, the condition $|x| < 2^{e+\mu-1}$ suffices in Proposition 1. Indeed, this weaker condition implies $2^{e+\mu} < A < 2 \cdot 2^{e+\mu}$, and therefore $2^{e+\mu} < a < 2 \cdot 2^{e+\mu}$, with the rest of the proof unchanged. The same remark holds for Proposition 4 below and allows several of the subsequent bounds to be slightly improved. As a consequence, this sometimes allows us to decrease $\delta$ 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.

**C. Normalization**

Let $1 \leq B \leq 2^e$. We will write $F_{p,2}$ for the subset of $F_{p,2}$ of all numbers $x = [x_0, x_1]$ with $|x_1| < B \cdot 2^{p-1} \leq 2^{p-1}$.

Numbers in $F_{p,2}$ are said to be in normal form.

**Remark 3** Both $x = [2^{-p}, -2^{-p}, 1/2]$ 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 [18]).

**Algorithm Normalize**

\[ c := \text{Split}_e(x); \text{return } [c(x_0 + c), c(x_1 - c)] \]

**Proposition 2** Given $x \in F_{p,2}$ with $|x_0| < 2^e - 2^{p+\delta}$, the algorithm Normalize returns $\tilde{x} \in F_{p,2}$ with $|\tilde{x} - x| < 2^{-2^{p+\delta}}$.

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

**D. Addition and subtraction**

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

**Algorithm Add**

\[ [c(x_0 + y_0), c(x_1 + y_1)] \]

**Algorithm Subtract**

\[ [c(x_0 - y_0), c(x_1 - y_1)] \]

**Proposition 3** Let $x \in F_{p,2,B}$ and $y \in F_{p,2,C}$ with $S := B + C + 2^{-p} \leq 2^e$. If $|x_0 + y_0| < 2^e$, then we have $a = Add(x, y) \in F_{p,2,S}$ with $|a - (x + y)| < 2^{-2p}$. If $|x_0 - y_0| < 2^e$, then we have $b = \text{Subtract}(x, y) \in F_{p,2,S}$ with $|b - (x - y)| < 2^{-2p}$.

**Proof:** Since both $x_0$ and $y_0$ are multiples 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, $|a_1 - (x_1 + y_1)| < 2^{2e-2\mu-\delta} - 2^{-2p}$ and $|a_1| < |x_1 + y_1| + 2^{-2p} \leq |x_1| + |y_1| + 2^{-2p} < 2^{p+e-2}$. Finally $|a_0 - (x_0 + y_0)| + |a_1 - (x_1 + y_1)| \leq |a_1 - (x_1 + y_1)| < 2^{-2p}$.

For the subtraction, we replace $y$ by $-y$.

**E. Multiplication**

Our multiplication algorithm of fixed point numbers is based on a subalgorithm LongMul, which computes the exact product of two numbers $x, y \in F$ in the form of a sum $xy = h + l$, with the additional constraint that $h \in \mathbb{Z}_2$. Without this additional constraint (and in absence of overflow and underflow), $h$ and $l$ can be computed using the classical “Two Product” algorithm: $h := o(xy)$, $l := o(xy - h)$. Our LongMul algorithm exploits the FMA and FMS instructions in a similar way.

**Algorithm LongMul**

\[ (a, h, l) := \text{LongMul}(x_0, y_0); h := o(xy - h); \text{return } (h, l) \]

**Algorithm Multiply**

\[ (h, l) := \text{Multiply}(x_0, y_0); l := o(x_0y_1 + l); \text{return } (l, l) \]

**Proposition 4** Let $x, y \in F$ and $e \in \{E_{\min} + \mu, \ldots, E_{\max} - \mu\}$ be such that $|xy| < 2^{e+\mu - 2}$. Then the algorithm LongMul computes a pair $(h, l)$ such that $|h + l - xy| < 2^{-e-\mu}$ and $|l| \leq 2^e$. In addition, if $xy \in \mathbb{Z}_2$, then $h + l = xy$ and $|l| < 2^e$.

**Proof:** In a similar way as in the proof of Proposition 1, one shows that $h \in \mathbb{Z}_2$ and $|h - xy| < 2^e$. It follows that $|l| \leq 2^e$ and $|l - (xy - h)| < 2^{-2\mu}$. If $xy \in \mathbb{Z}_2$, then the subtraction $xy - h$ is exact, whence $|l| = |h - xy| < 2^e$.

**Proposition 5** Let $x \in F_{p,2,B}$ and $y \in F_{p,2,C}$ with $|x| < B$, $|y| < C$ and $B + C + 2^{-p} \leq 2^e$. Then we have $r = \text{Multiply}(x, y) \in F_{p,2,B}BC + 2^{-2p}$ and $|r - xy| < (BC + 2)2^{-2p}$.

**Proof:** Let us write $l_1$, $l_1$, and $l_1$ for the successive values of $l$ taken during the execution of the algorithm. Since $|x_0y_0| < BC < 2^{e-2}$, Proposition 4 implies that $h \in \mathbb{Z}_2$, $|l_1| < 2^{-p}$, and $h + l_1 = x_0y_0$. Using that $|x| < B2^{-p}$ and $|y| < C2^{-p}$, we next obtain $|l_1| \leq (BC + 1 + 2\cdot 2^{-p})2^{-p}$ and $|l_1| \leq (2BC + 1 + 2\cdot 2^{-p})2^{-p}$. We also get that $|l_1 - (l_1 + x_0y_1)| \leq 2^{-2p}$ and $|l_1 - (l_1 + x_1y_0)| \leq 2^{-2p}$. Finally we obtain:

\[ |r - xy| \leq |r - x_0y_0 - x_1y_0 - x_1y_1| \leq |h + l_1 - x_0y_0| + |l_1 - l_1 - x_0y_1| + |x_1y_1| \leq 2 \cdot 2^{-2p} + BC2^{-2p}. \]
F. C++ implementation inside MATHEMAGIX

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

```cpp
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 [19] for details). When instantiating for `C=double` and the default variant `V`, the type `fixed_quadruple<` corresponds to `F_p` with `p = 48` (see the file `numeric/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 [19]. 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 `numeric/avx.hpp`.

III. Fast Fourier Transforms

In this section we describe how to use the fixed point arithmetic functions to compute FFTs. The number of tail 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 [12].

A. Complex arithmetic

We implement complex analogues `CNormalize`, `CAdd`, `CSubtract` and `CMultiply` of `Normalize`, `Add`, `Subtract` and `Multiply` in a naïve way. We have fully specified `CMultiply` below, as an example. The three other routines proceed componentwise, by applying the real counterparts on the real and imaginary parts. Here `ℜu` and `ℑu` represent the real and imaginary parts of `u` respectively. The norm `∥u∥∞` of the complex number `u` is defined as `max(|ℜu|, |ℑu|).

Algorithm `CMultiply`(u, v)

```cpp
a := Multiply(ℜu, ℑv); b := Multiply(ℜu, 3ℑv);
c := Multiply(ℑu, ℑv); d := Multiply(3ℑu, ℑv)
```

return `Subtract(a, b) + Add(c, d)`

B. Butterflies

The basic building block for fast discrete Fourier transforms is the complex butterfly operation. Given a pair `(u, v) ∈ C^2` and a precomputed root of unity `ω ∈ C`, the butterfly operation computes a new pair `(u + ωv, u − ωv)`. For inverse transforms, one rather computes the pair `(u + v, (u − v)ω)` instead. For simplicity, and without loss of generality, we may assume that the approximation of `ω` in `F_p` satisfies `∥ω∥∞ ≤ 1.`

Algorithm `DirectButterfly`(u, v, ω)

```cpp
z := CMultiply(ω, v); u′ := CAdd(u, z);
v′ := CSubtract(u, z); \tilde{u} := CNormalize(u′)
\tilde{v} := CNormalize(v′); return (\tilde{u}, \tilde{v})
```

Algorithm `InverseButterfly`(u, v, ω)

```cpp
u′ := CAdd(u, v); z := CSubtract(u, v)
```

Proposition 6 Let `u, v, ω ∈ F_p` with `∥u0∥∞ < 1, ∥u∥∞ < 1, ∥ω∥∞ ≤ 1` and assume that `δ ≥ 4`. Then `(\tilde{u}, \tilde{v}) = DirectButterfly(u, v, \omega) ∈ F_p` and `∥\tilde{u} − (u + ωv)∥∞, ∥\tilde{v} − (u − vω)∥∞ < 9 · 2^{-2p}`.

Proof: Let `a, b, c, d` be as in `CMultiply` with `u = ω` and `v` as arguments. From Proposition 5, it follows that `a ∈ F_p` and `|a − ℜω\ℜv| < 3 · 2^{-2p}`, and we have similar bounds for `b, c` and `d`. Setting `s = Subtract(a, b)`, Proposition 3 yields `s ∈ F_p` and `|s − (a − b)| < 2^{-2p}`, whence `|s − (\ℜω\ℜv − 3\ℑv)| < 7 · 2^{-2p}. After obtaining a similar bound for `Add(c, d), it follows that z ∈ F_p` and `∥z − \omega v∥∞ < 7 · 2^{-2p}. Applying Proposition 3 once more, we obtain `u′, v′ ∈ F_p` and `∥u′ − (u + vω)∥∞ < 8 · 2^{-2p}` and `∥v′ − (u − vω)∥∞ < 8 · 2^{-2p}`. The conclusion now follows from Proposition 2.

Proposition 7 Let `u, v, ω ∈ F_p` with `∥u0∥∞ < 1, ∥u∥∞ < 1, ∥\omega∥∞ ≤ 1` and assume that `δ ≥ 4`. Then `(\tilde{u}, \tilde{v}) = InverseButterfly(u, v, \omega) ∈ F_p` and `∥\tilde{u} − (u + vω)∥∞, ∥\tilde{v} − (u − vω)∥∞ < 10 · 2^{-2p}`.

Proof: Proposition 3 yields `u′, z ∈ F_p` and `∥u′ − (u + vω)∥∞ < 2^{-2p}` and `∥z − (u − vω)∥∞ < 2^{-2p}`. Proposition 2 then gives us `\tilde{u} ∈ F_p` and `∥\tilde{u} − (u + \omega v)∥∞ < 2^{-2p}. Let a, b, c, d be as in `CMultiply` with `ω` and `z` as arguments. From Proposition 5, it follows that `a ∈ F_p` and `|a − ℜω\ℜv| < (4 + 2^{-p}) · 2^{-2p}`, and we have similar bounds for `b, c` and `d`. From Proposition 3 again, we get `v′ ∈ F_p` and `∥v′ − (u − vω)∥∞ < (9 + 2^{-2p}) · 2^{-2p}. Then Proposition 2 gives us `\tilde{v} ∈ F_p` and `∥\tilde{v} − (u − v)\omega∥∞ < 10 · 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 [13], which can be adapted to the present context thanks to the above propositions.

C. 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 [20] version 4.9.1.

For the sake of comparison, Table I displays timings to compute FFTs over complex numbers with the FFTW library version 3.3.4 [16]. 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. Our MATHEMAGIX implementations of the algorithm in this paper are available from revision 9621. Timings are reported in Table II. We have implemented the split-radix algorithm (see [21], [22] for recent advances and historical references). 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 [8], 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 instruction sets. 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 is IEEE compliant quadruple precision type __float128, and abbreviate MPFR (113 bits) for numbers of the above form and abbreviate \( F_{p,k} = F_{p,k,2^p} \). Numbers in \( F_{p,k,1} \) are said to be in normal form. Of course, we require that \( (k-1)p \) is smaller than \( -E_{\text{min}} - \mu \). For this, we assume that \( k \leq 19 \). 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

\[ c_i := \text{Split}_{-ip}(r_i), \quad x_i := o(r_i - c_i) \]

\[ r_{i-1} := o(x_{i-1} + c_i) \]

\[ x_0 := r_0; \quad \text{return} \ [x_0, \ldots, x_{k-1}] \]

Proposition 8 Given a fixed point number \( x \in F_{p,k,1} \) with \( |x_0| < 2^{-1} - 2^{-p}, B \leq 2^{2} - 2^{-p} \), the algorithm Normalize returns \( \tilde{x} \in F_{p,k,1} \) with \( |\tilde{x} - x| < 2^{-kp'} \).

Proof: During the first step of the loop, when \( i = k-1 \), by Proposition 1 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 \( |r_{k-1}| < 2^{-(k-1)p} \), this entails \( |c_{k-1}| < (2^2 + 1)2^{1-(k-1)p} \) and therefore \( |c_{k-1}| < 2^{-(k-1)p} \). Hence \( |x_{k-2} + c_{k-1}| \leq |x_{k-2}| + |c_{k-1}| < 2^{-(k-2)p} \), so that \( r_{k-2} = o(x_{k-2} + c_{k-1}) = x_{k-2} + c_{k-1} \). Using similar arguments for \( i = k - 2, \ldots, 1 \), we obtain \( \tilde{x} \in F_{p,k,1} \) and \( |\tilde{x} - x| \leq |r_{k-1} - c_{k-1}|. \) Finally, since \( |r_{k-1} - c_{k-1}| < 2^{-(k-1)p} \) we have \( |\tilde{x}_{k-1} - (r_{k-1} - c_{k-1})| < 2^{-(k-1)p} \), which concludes the proof.
B. Ring operations

The generalizations of Add and Subtract are straightforward: Add\((x, y) = \{x_0 + y_0, \ldots, x_k + y_k\}\), and Subtract\((x, y) = \{x_0 - y_0, \ldots, x_k - y_k\}\).

**Proposition 9** Let \(x \in \mathcal{F}_{p,k,b}\) and \(y \in \mathcal{F}_{p,k,c}\) with \(S := B + C + 2^{-p} ≤ 2^p\). If \(|x_0 + y_0| < 2^p\), then \(a = \text{Add}(x, y) \in \mathcal{F}_{p,k,s}\) with \(|a - (x + y)| < 2^{-kp}\). If \(|x_0 - y_0| < 2^p - 2^{-p}\), then \(b = \text{Subtract}(x, y) \in \mathcal{F}_{p,k,s}\) with \(|a - (x - y)| < 2^{-kp}\).

**Proof:** Since both \(x_i\) and \(y_i\) are integer multiples of \(2^{-ip}\) for \(0 ≤ i < k\) and add in \(B\), the additions \(a_i = x_i + y_i\) are exact. The exponents of \(x_{k-1}\) and \(y_{k-1}\) are both strictly bounded by \(d - (k - 1)p\). Consequently, \(|a - (x + y)| ≤ |a_{k-1} - (x_{k-1} + y_{k-1})| < 2^{d-(k-1)p - p} < 2^{-kp}\) and \(|a_{k-1}| < |x_{k-1} + y_{k-1}| + 2^{-kp} ≤ |x_{k-1} + y_{k-1}| + 2^{-kp} < S^{-2(k-1)p}\). For the subtraction we replace \(y - b\).

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

**Algorithm Multiply\((x, y)\):**

\[
\begin{align*}
(\tau_0, r_k) &:= \text{LongMul}_{p}(x_0, y_0), \\
&\text{for } i \text{ from } 1 \text{ to } k - 1 \text{ do} \\
&\quad (h, r_{i+1}) := \text{LongMul}_{-(i+1)p}(x_i, y_i), \\
&\quad r_i := c(h_i + h), \\
&\text{for } j \text{ from } 1 \text{ to } i \text{ do} \\
&\quad (h, l) := \text{LongMul}_{-(i+1)p}(x_j, y_{i-j}), \\
&\quad r_i := c(h_i + h), \\
&\text{for } i \text{ from } 0 \text{ to } k - 1 \text{ do} \\
&\quad r_{k-1} := c(x_i y_{k-1-i} + r_{k-1}) \\
&\text{return } [r_0, \ldots, r_{k-1}] 
\end{align*}
\]

**Proposition 10** Let \(x \in \mathcal{F}_{p,k,b}\) and \(y \in \mathcal{F}_{p,k,c}\) with \(|x_0| < B\), \(|y_0| ≤ C\), \(BC ≤ 2^{d-2}\), and \(S := k(BC+1-2^{-p}) - 1 \leq 2^p\). Then \(r = \text{Multiply}(x, y) \in \mathcal{F}_{p,k,s}\) with \(|r - xy| < 2S^{-2kp}\).

**Proof:** Let us write \((h_{i,j}, l_{i,j})\) for the pair returned by \(\text{LongMul}_{-(i+1)p}(x_i, y_i)\), for \(0 ≤ i + j ≤ k - 2\). Since \(BC ≤ 2^{d-2}\), Proposition 4 implies that \(h_{i,j} \in Z^{2^{-(i+1)p}}, \|l_{i,j}\| < 2^{-(i+1)p}\), 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 \(r_i = \sum_{j+i=1} h_{i,j} + \sum_{i+j=1} l_{i,j}\) for all \(1 ≤ t ≤ k\), and therefore \(|r_i| < (t + 1)BC2^{-tp} + t2^{-tp} ≤ 2S^{-2kp}\) for all \(0 ≤ t ≤ k - 2\). In particular no overflow occurs and these sums are exact. Let \(s_0 = \sum_{i+j=2-k} l_{i,j}\) represent the value of \(r_{k-1}\) before entering the second loop. Then let \(s_{i+1} = c(x_i y_{k-1-i} + s_i)\) for \(0 ≤ i < k - 1\), 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 \(|s_i| < (iBC + k - 1 + i2^{-p})2^{-(k-1)p}\) for all \(0 ≤ i ≤ k\). For the precision loss, from \(|s_{i+1} - (x_i y_{k-1-i} + s_i)| ≤ 2^{-kp}\), we deduce that

\[
|r - xy| ≤ \sum_{0 ≤ i ≤ k} |s_{i+1} - (x_i y_{k-1-i} + s_i)| + \sum_{i+j ≥ k} |x_i y_j| < k2^{-kp} + BC \sum_{i+j ≥ k} 2^{-(i+j)p}.
\]

The rest of the proof follows from \(\sum_{i+j ≥ k} 2^{-(i+j)p} = 2^{-kp} \sum_{0 ≤ i ≤ k-2} (k-1-i)2^{-ip} = (k-1-k2^{-p} + 2^{-kp}/(1-2^{-p}))^22^{-kp} ≤ (k-1)(1+2^{-p})2^{-kp}\), while using \(2 ≤ k ≤ 19\) and \(p ≥ 5\).

**C. Fast Fourier Transforms**

We can use the same butterfly implementations as in Section III-B. The following generalization of Proposition 6 shows that we may take \(δ = 4\) as long as \(k ≤ 4\), and \(δ = 5\) as long as \(k ≤ 8\) for the direct butterfly operation.

**Proposition 11** Let \(u, v, ω \in \mathcal{F}_{p,k,1}\) with \(|u|_∞ < 1\), \(|v|_∞ < 1\), \(|ω|_∞ < 1\), and assume that \(δ ≥ 4\) and that \(4k - 1 + (2k + 2)2^{-p} < 2^{δ} - 2^{2}2^{-p}\). Then \((\tilde{u}, \tilde{v}) = \text{DirectButterfly}(u, v, ω) \in \mathcal{F}_{p,k,1}^{[i]}\) and \(\|\tilde{u} - (u + vω)\|_∞ < (4k + 1)2^{-2kp}\).

**Proof:** Let \(a, b, c, d\) be as in CMultiply with \(u = ω\) and \(v\) as arguments. From Proposition 10, it follows that \(a \in \mathcal{F}_{p,k,2k-1+k2^{-p}}\) and \(|a - ⨁_{ω}| < (2k - 1 + k2^{-p})2^{-kp}\), and we have similar bounds for \(b, c\) and \(d\). From Proposition 9, we next get \(z \in \mathcal{F}_{p,k,4k+2-2(2k+1)2^{-p}}\) and \(|z - ω|_∞ < (4k - 1 + k2^{-2p})2^{-2kp}\). Applying Proposition 9 again, we obtain \(u', v' \in \mathcal{F}_{p,k,4k+2-2(2k+1)2^{-p}}\), \(|u' - (u + vω)|_∞ < (4k + 1 - 4k - 2 + 2k2^{-2p})2^{-2kp}\). Then \((\tilde{u}, \tilde{v})\) is replaced by \(\text{InveButterfly}(u, v) \in \mathcal{F}_{p,k,1}^{[i]}\) and \(\|\tilde{u} - (u + v)|_∞ < (6k + 1)2^{-2kp}\).

**Remark 4** 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 \(δ = 4\) by performing one additional normalization of \(z\) in InveButterfly.

**Remark 5** The following strategy sometimes makes it possible to decrease \(δ\) by one: instead of normalizing the entries of an FFT to be of norm \(\|\cdot\|_∞ < 1\), we rather normalize
them to be of norm $\| \cdot \|_\infty < 1 - Ck2^{-p}$ for some small constant $C$. This should allow us to turn the conditions $4k - 1 + (2k + 2)2^{-p} < 2^\delta - 2^{\delta - p}$ and $6k - 2 + (4k + 1)2^{-p} < 2^\delta - 2^{\delta - p}$ in the Propositions 11 and 12 into $4k - 1 \leq 2^\delta$ and $6k - 2 \leq 2^\delta$. One could also investigate what happens if we additionally assume $\| \cdot \|_\infty < 1/2 - Ck2^{-p}$.

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 [23] for the complex multiplication, meaning that we compute $(\Re u + 3\Re v + 3\Im u)\delta - \Re u\delta + 3\Im u\delta$, which corresponds to packing four elements of type fixed_hexuple<avx_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 III. 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.

V. VARIANTS AND PERSPECTIVES

a) 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

$$x = x_0 + x_12^{-p} + \cdots + x_{k-1}2^{-(k-1)p}$$

in the “indeterminate” $2^{-p}$, with $x_0, \ldots, x_{k-2} \in \mathbb{Z}2^{-p}$ and $|x_0|, \ldots, |x_{k-1}| < 2^p$. 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 $l2^p + h$ instead of a simple addition $l + h$.

For naïve multiplication, the polynomial representation may become slightly more efficient for $k \geq 3$, when the hardware supports an SIMD instruction for rounding floating point numbers to the nearest integer. In addition, when $k$ becomes somewhat larger, say $k \geq 8$, then the polynomial representation makes it possible to use even-odd Karatsuba multiplication [24], [25] without any need for special carry handling (except that $\delta$ has to be chosen a bit larger). Finally, the restriction $(k - 1)p \leq -E_{\text{min}} - \mu$ is no longer necessary, so the approach can in principle be extended to much larger values of $k$.

b) 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 64k 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 $xy$ 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 $xy = (x+C)(y+C) - (x+y)C + C^2$. We expect that the biggest gain of using integer arithmetic comes from the fact that we achieve a 64k instead of a pk 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 resort to branching for carry handling. One remedy would be to use expansions $x = x_0 + x_1 264s + \cdots + x_k/s-1 2^{64k-64s}$ with coefficients $0 \leq x_i < 2^{64s-\delta}$ for a suitable number $\delta$ of nail bits and a suitable “multiplexer” $s > 1$.

c) 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 $e \in \{E_{\text{min}}, \ldots, E_{\text{max}}\}$ on input and which computes numbers $h, l \in \mathbb{F}$ with $x = h + l$, $h \in \mathbb{Z}^\ast$ and $|l| < \mathbb{Z}^\ast$. In that case, normalization would only require two instructions instead of four. Besides speeding up the naïve 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.

d) Asymptotically efficient algorithms: A theoretically important question concerns the asymptotic complexity of FFTs at large precisions. Let $l(b)$ denote the bit complexity of b bit integer multiplication. It has recently been proved [13] that $l(b) = O\left(b \log b \log \log \log b\right)$, where $\log^* b = \min_k \left\{ \log^{k+1} b \leq 1 \right\}$. Under mild assumptions on $b$ and $n$ it can also be shown [13] that an FFT of size $n$ and bit precision $b$ can be performed in time $F_0(n) = O\left(l(bn)\right)$. For $b \leq n$ and large $n$, this means that $F_0(n)$ scales linearly with $b$. In a future paper, we intend to investigate the best possible (and practically achievable) constant factor involved in this bound.

REFERENCES