Multiple precision floating-point arithmetic on SIMD processors

Joris van der Hoeven

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

Email: vdhoeven@lix.polytechnique.fr

April 26, 2017

Current general purpose libraries for multiple precision floating-point arithmetic such as Mpfr suffer from a large performance penalty with respect to hard-wired instructions. The performance gap tends to become even larger with the advent of wider SIMD arithmetic in both CPUs and GPUs. In this paper, we present efficient algorithms for multiple precision floating-point arithmetic that are suitable for implementations on SIMD processors.

Keywords: floating-point arithmetic, multiple precision, SIMD

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

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

1. Introduction

Multiple precision arithmetic [3] is crucial in areas such as computer algebra and cryptography, and increasingly useful in mathematical physics and numerical analysis [1]. Early multiple precision libraries appeared in the seventies [2], and nowadays GMP [7] and MPFR [6] are typically very efficient for large precisions of more than, say, 1000 bits. However, for precisions that 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 Intel's AVX technology. Indeed, it is hard to take advantage of wide SIMD instructions when implementing basic arithmetic for individual numbers of only a few words. In order to fully exploit SIMD instructions, one should rather operate on suitably represented SIMD vectors of multiple precision numbers. A second problem with current SIMD arithmetic is that CPU vendors tend to favor wide floating-point arithmetic over wide integer arithmetic, whereas faster integer arithmetic is 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 floating-point numbers.

One existing approach is based on the "**TwoSum**" and "**TwoProduct**" operations [4, 13] that 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. $\lfloor \log_2 |x| \rfloor \geqslant \lfloor \log_2 |y| \rfloor + 53$ or $y = 0$). The **TwoProduct** operation can be implemented

using only two instructions when hardware offers the *fused-multiply-add* (FMA) and *fused-multiply-subtract* (FMS) instructions, as is for instance the case for AVX2 enabled processors. The **TwoSum** operation 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 well known that double machine precision arithmetic can be implemented reasonably efficiently in terms of the **TwoSum** and **TwoProduct** algorithms [4, 13, 15]. The approach has been further extended in [17, 14] to higher precisions. Specific algorithms are also described in [12] for triple-double precision, and in [9] for quadruple-double precision. But these approaches tend to become inefficient for large precisions.

An alternative approach is to represent floating-point numbers by products $m\,b^e$, where m is a fixed-point mantissa, e an exponent, and $b \in 2^{\mathbb{N}}$ the base. This representation is used in most of the existing multi-precision libraries such as GMP [7] and MPFR [6]. However, the authors are only aware of sequential implementations of this approach. In this paper we examine the efficiency of this approach on SIMD processors. As in [10], we systematically work with vectors of multiple precision numbers rather than with vectors of "digits" in base b. We refer to [19, 5] for some other recent approaches.

Our paper is structured as follows. In section 2, we detail the representation of fixed-point numbers and basic arithmetic operations. We follow a similar approach as in [10], but slightly adapt the representation and corresponding algorithms to allow for larger bit precisions of the mantissa. As in [10], we rely on standard IEEE-754 compliant floating-point arithmetic that is supported by most recent processors and GPUs. For processors with efficient SIMD integer arithmetic, it should be reasonably easy to adapt our algorithms to this kind of arithmetic. Let μ be the bit precision of our machine floating-point numbers minus one (so that $\mu = 52$ for IEEE-754 double precision numbers). Throughout this paper, we represent fixed-point numbers in base 2^p by k-tuplets of machine floating-point numbers, where p is slightly smaller than μ and $k \geqslant 2$.

The main bottleneck for the implementation of floating-point arithmetic on top of fixed-point arithmetic is shifting. This operation is particularly crucial for addition, since every addition requires three shifts. Section 3 is devoted to this topic and we will show how to implement reasonably efficient shifting algorithms for SIMD vectors of fixed-point numbers. More precisely, small shifts (of less than p bits) can be done in parallel using approximately $4\,k$ operations, whereas arbitrary shifts require approximately $(\log_2 k + 4)\,k$ operations.

In section 4, we show how to implement arbitrary precision floating-point arithmetic in base $b = 2$. Our approach is fairly standard. On the one hand, we use the "left shifting" procedures from section 3 in order to normalize floating-point numbers (so that mantissas of non zero numbers are always "sufficiently large" in absolute value). On the other hand, the "right shifting" procedures are used to work with respect to a common exponent in the cases of addition and subtraction. We also discuss a first strategy to reduce the cost of shifting and summarize the corresponding operation counts in Table 2. In section 5, we perform a similar analysis for arithmetic in base $b = 2^p$. This leads to slightly less compact representations, but shifting is reduced to multiple word shifting in this setting. The resulting operation counts can be found in Table 3.

The operation counts in Tables 2 and 3 really represent the worst case scenario in which our implementations for basic arithmetic operations are required to be "black boxes". Multiple precision arithmetic can be made far more efficient if we allow ourselves to open up these boxes when needed. For instance, any number of floating-pointing numbers can be added using a single function call by generalizing the addition algorithms from sections 4 and 5 to take more than two arguments; this can become almost thrice as efficient as the repeated use of ordinary additions. A similar approach can be applied to entire algorithms such as the FFT [10]: we first shift the inputs so that they all admit the same exponent and then use a fixed-point algorithm for computing the FFT. We intend to come back to this type of optimizations in a forthcoming paper.

So far, we only checked the correctness of our algorithms using a prototype implementation. Our operation count analysis indicates that our approach should outperform others as soon as $k \geqslant 5$ and maybe even for $k = 3$ and $k = 4$. Another direction of future investigations concerns correct rounding and full compliance with the IEEE standard, taking example on Mpfr [6].

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 $\mu \geqslant 8$ 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 $E_{\max} = 1023$.

In this paper, and contrary to [10], the rounding mode is always assume to be "round to nearest". Given $x, y \in \mathbb{F}$ and $* \in \{+, -, \cdot\}$, we denote by $\circ(x * y)$ the rounding of $x * y$ to the nearest. For convenience of the reader, we denote $\bullet(x * y) = \circ(x * y)$ whenever the result $\circ(x * y) = x * y$ is provably exact in the given context. 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 $|\circ(x * y) - x * y| \leqslant 2^{e-\mu-1}$. For efficiency reasons, the algorithms in this paper do not attempt to check for underflows, overflows, and other exceptional cases.

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 $x\,y - z$ to the nearest.

Acknowledgment. We are very grateful to the third referee for various suggestions and for drawing our attention to several more or less serious errors in an earlier version of this paper.

2. Fixed-point arithmetic

Let $p \in \{6, \ldots, \mu - 2\}$ and $k \geqslant 2$. In this section, we start with a survey of efficient fixed-point arithmetic at bit precision $k\,p$. We recall the approach from [10], but use a slightly different representation in order to allow for high precisions $k > 19$. We adapted the algorithms from [10] accordingly and explicitly state the adapted versions. Allowing p to be smaller than μ corresponds to using a redundant number representation that makes it possible to efficiently handle carries during intermediate computations. We denote by $\delta = \mu - p \geqslant 2$ the number of extra carry bits (these bits are sometimes called "nails", following GMP [7]). We refer to section 3.5 for a table that recapitulates the operation counts for the algorithms in this section.

2.1. Representation of fixed-point numbers

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

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

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

$$\begin{array}{rcll} x_i & \in & \mathbb{Z}\,2^{-p} & \text{for } 0 \leqslant i < k \\ |x_0| & < & 2^\delta & \\ |x_i| & < & C & \text{for } 0 < i < k. \end{array}$$

We write $x = [x_0, \ldots, x_{k-1}]$ for numbers of the above form and abbreviate $\mathcal{F}_{p,k} = \mathcal{F}_{p,k;2^\delta}$. Numbers in $\mathcal{F}_{p,k;4/5}$ are said to be in *carry normal form*.

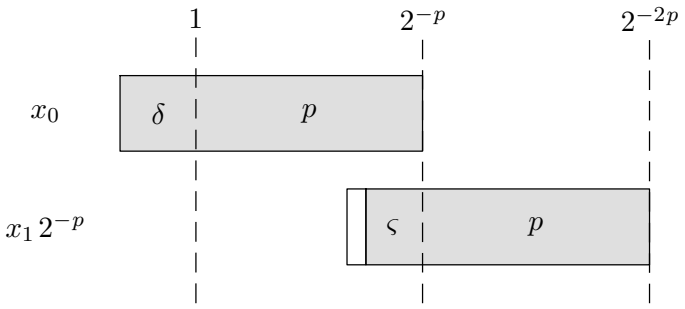

Figure 1. Schematic representation of a fixed-point number $x = [x_0, x_1] = x_0 + x_1\,2^{-p}$, where $\varsigma = \lceil \log_2 C \rceil$.

Remark 1. The paper [10] rather uses the representation $x = x_0 + \cdots + x_{k-1}$ with $x_i \in \mathbb{Z}\,2^{-(i+1)p}$ and $|x_i| < C\,2^{-ip}$. This representation is slightly more efficient for small k, since it allows one to save one operation in the implementation of the multiplication algorithm. However, it is limited to small values of k (typically $k \leqslant 19$), since $(k-1)\,p$ must required to be smaller than $-E_{\min} - \mu$. The representation (1) also makes it easier to implement efficient multiplication algorithms at high precisions k, such as Karatsuba's algorithm [11] or FFT-based methods [16, 18, 8]. We intend to return to this issue in a forthcoming paper.

Remark 2. Another minor change with respect to [10] is that we also require $x_i \in \mathbb{Z}\,2^{-p}$ to hold for the last index $i = k-1$. In order to meet this additional requirement, we need two additional instructions at the end of the multiplication routine in section 2.6 below.

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 $\bullet(a - {}^3\!/_2 \cdot 2^{e+\mu})$

Proposition 1 from [10] becomes as follows for "rounding to nearest":

PROPOSITION 3. *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 $|\tilde{x} - x| \leqslant 2^{e-1}$.*

2.3. Carry propagation

Numbers can be rewritten in carry normal form using carry propagition. This can be achieved as follows (of course, the loop being unrolled in practice):

Algorithm CarryNormalize(x)

$r_{k-1} := x_{k-1}$
for i **from** $k-1$ **down to** 1 **do**
  $c_i := \mathbf{Split}_0(r_i)$
  $\tilde{x}_i := \bullet(r_i - c_i)$
  $r_{i-1} := \bullet(x_{i-1} + c_i\,2^{-p})$
$\tilde{x}_0 := r_0$
return $[\tilde{x}_0, \ldots, \tilde{x}_{k-1}]$

A straightforward adaptation of [10, Proposition 2] yields:

PROPOSITION 4. *Given a fixed-point number $x \in \mathcal{F}_{p,k;C}$ with $|x_0| < 2^\delta - 2^{\delta-p}$ and $C \leqslant 2^\delta - 2^{\delta-p}$, the algorithm* **CarryNormalize** *returns $\tilde{x} \in \mathcal{F}_{p,k;1/2+2^{-p}}$ with $\tilde{x} = x$.*

2.4. Addition and subtraction

Non normalized addition and subtraction of fixed-point numbers is straightforward:

Algorithm Add(x, y)
return $[\bullet(x_0 + y_0), ..., \bullet(x_{k-1} + y_{k-1})]$

Algorithm Subtract(x, y)
return $[\bullet(x_0 - y_0), ..., \bullet(x_{k-1} - y_{k-1})]$

Proposition 9 from [10] now becomes:

Proposition 5. *Let $x \in \mathcal{F}_{p,k;C}$ and $y \in \mathcal{F}_{p,k;D}$ with $S := C + D + 2^{-p} \leqslant 2^\delta$. If $|x_0 + y_0| < 2^\delta$, then $a = \mathbf{Add}(x, y)$ satisfies $a = x + y \in \mathcal{F}_{p,k;S}$. If $|x_0 - y_0| < 2^\delta$, then $b = \mathbf{Subtract}(x, y)$ satisfies $b = x - y \in \mathcal{F}_{p,k;S}$.*

2.5. Double length multiplication

The multiplication algorithm of fixed-point numbers is based on a subalgorithm $\mathbf{LongMul}_e$ that computes the exact product of two numbers $x, y \in \mathbb{F}$ in the form of a sum $x\,y = h + \ell$, 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 well known "Two Product" algorithm: $h := \circ(x\,y)$, $\ell := \bullet(x\,y - h)$. The $\mathbf{LongMul}_e$ algorithm exploits the FMA and FMS instructions in a similar way.

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

Proposition 4 from [10] becomes as follows for "rounding to nearest":

Proposition 6. *Let $x, y \in \mathbb{F}$ and $e \in \{E_{\min} + \mu, ..., E_{\max} - \mu\}$ be such that $|x\,y| < 2^{\mu+e-2}$ and $x\,y \in \mathbb{Z}\,2^{e-\mu}$. Then the algorithm $\mathbf{LongMul}_e(x, y)$ computes a pair $(h, \ell) \in \mathbb{F}^2$ with $h \in \mathbb{Z}\,2^e$, $h + \ell = x\,y$, and $|\ell| \leqslant 2^{e-1}$.*

2.6. General fixed-point multiplication

For C large enough as a function of k, one may use the following algorithm for multiplication (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**
$\quad(h, r_{i+1}) := \mathbf{LongMul}_{-p}(x_0, y_i)$
$\quad r_i := \bullet(r_i\,2^p + h)$
$\quad$**for** j **from** 1 **to** i **do**
$\quad\quad(h, \ell) := \mathbf{LongMul}_{-p}(x_j, y_{i-j})$
$\quad\quad r_i := \bullet(r_i + h)$
$\quad\quad r_{i+1} := \bullet(r_{i+1} + \ell)$
$r_{k-1} := \bullet(r_{k-1}\,2^p)$
for i **from** 0 **to** $k-1$ **do**
$\quad r_{k-1} := \circ(x_i\,y_{k-1-i} + r_{k-1})$
$r_{k-1} := \mathbf{Split}_{-p}(r_{k-1})$
return $[r_0, ..., r_{k-1}]$

Notice that we need the additional line $r_{k-1} := \mathbf{Split}_{-p}(r_{k-1})$ at the end with respect to [10] in order to ensure that $r_{k-1} \in \mathbb{Z}\,2^{-p}$. Adapting [10, Proposition 10], we obtain:

PROPOSITION 7. *Let $x \in \mathcal{F}_{p,k;C}$ and $y \in \mathcal{F}_{p,k;D}$ with $|x_0| < C$, $|y_0| \leqslant D$, $C\,D \leqslant 2^{\delta-2}$, and $S := {}^{k}\!/_{2}\,(C\,D + 1 + 2^{2-p}) \leqslant 2^{\delta}$. Then $r = \mathbf{Multiply}(x, y) \in \mathcal{F}_{p,k;S}$ with $|r - x\,y| < S\,2^{-kp}$.*

3. FAST PARALLEL SHIFTING

In this section, we discuss several routines for shifting fixed-point numbers. All algorithms were designed to be naturally parallel. More precisely, we logically operate on SIMD vectors $\boldsymbol{x} = (x^1, ..., x^w)$ of fixed-point numbers $x^i = [x^i_0, ..., x^i_{k-1}]$. Internally, we rather work with fixed point numbers $\boldsymbol{x} = [\boldsymbol{x}_0, ..., \boldsymbol{x}_{k-1}]$ whose "coefficients" are machine SIMD vectors $\boldsymbol{x}_j = (x^1_j, ..., x^w_j)$. All arithmetic operations can then simply be performed in SIMD fashion using hardware instructions. For compatibility with the notations from the previous section, we omit the bold face for SIMD vectors, except if we want to stress their SIMD nature. For actual implementations for a given k, we also understand that we systematically use in-place versions of our algorithms and that all loops and function calls are completely expanded.

3.1. Small parallel shifts

Let $x = [x_0, ..., x_{k-1}]$ be a fixed-point number. Left shifts $x\,2^s$ and right shifts $x\,2^{-s}$ of x with $0 \leqslant s \leqslant p$ can be computed by cutting the shifted coefficients $x_i\,2^s$ resp. $x_i\,2^{-s}$ using the routine $\mathbf{Split}_{-p}$ and reassembling the pieces. The shift routines behave very much like generalizations of the routine for carry normalization.

Algorithm SmallShiftLeft(x, s)

$u := 2^{s-p}$
$\ell_0 := \bullet(u\,x_0)$
for i **from** 1 **to** $k-1$ **do**
  $h_i := \circ(u\,x_i + {}^{3}\!/_{2} \cdot 2^{\delta})$
  $h_i := \bullet(h_i - {}^{3}\!/_{2} \cdot 2^{\delta})$
  $\ell_i := \bullet(u\,x_i - h_i)$
  $r_{i-1} := \bullet(\ell_{i-1}\,2^p + h_i)$
$r_{k-1} := \bullet(\ell_{k-1}\,2^p)$
return $[r_0, ..., r_{k-1}]$

Algorithm SmallShiftRight(x, s)

$u := 2^{-s}$
$h_{k-1} := \circ(u\,x_{k-1} + {}^{3}\!/_{2} \cdot 2^{\delta})$
$h_{k-1} := \bullet(h_{k-1} - {}^{3}\!/_{2} \cdot 2^{\delta})$
for i **from** $k-2$ **down to** 0 **do**
  $h_i := \circ(u\,x_i + {}^{3}\!/_{2} \cdot 2^{\delta})$
  $h_i := \bullet(h_i - {}^{3}\!/_{2} \cdot 2^{\delta})$
  $\ell_i := \bullet(u\,x_i - h_i)$
  $r_{i+1} := \bullet(\ell_i\,2^p + h_{i+1})$
return $[h_0, r_1, ..., r_{k-1}]$

PROPOSITION 8. *Let $s \in \{0, ..., p\}$. Given a fixed-point number $x \in \mathcal{F}_{p,k;C}$ with $C \leqslant 2^{\mu-s-2}$ and $|x_0| < 2^{\delta-s} - C\,2^{-p} - 2^{-p-1-s}$, the algorithm* **SmallShiftLeft** *returns $r \in \mathcal{F}_{p,k;{}^{1}\!/_{2}+C2^{s-p}+2^{-p-1}}$ with $r = x\,2^s$.*

Proof. In the main loop, we observe that $h_i := \mathbf{Split}_{-p}(u\,x_i)$ and $u\,x_i \in 2^{s-2p}\,\mathbb{Z}$. The assumption $C \leqslant 2^{\mu-s-2}$ guarantees that Proposition 3 applies, so that $h_i \in \mathbb{Z}\,2^{-p}$ and $|h_i - u\,x_i| \leqslant 2^{-p-1}$. The operation $\ell_i := \bullet(u\,x_i - h_i)$ is therefore exact, so that $u\,x_i = h_i + \ell_i$, $2^p\,\ell_i \in 2^{s-p}$, and $|\ell_i| \leqslant 2^{-p-1}$. Since $|u\,x_i| < C\,2^{s-p}$, we also have $|h_i| < C\,2^{s-p} + 2^{-p-1}$. Now $|\ell_{i-1}\,2^p| \leqslant {}^{1}\!/_{2}$ if $i > 1$ and $|\ell_{i-1}\,2^p| < 2^{\delta} - C\,2^{s-p} - 2^{-p-1}$ if $i = 1$. Combined with the facts that $|h_i| < C\,2^{s-p} + 2^{-p-1}$ and $h_i, \ell_{i-1}\,2^p \in \mathbb{Z}\,2^{-p}$, it follows that the operation $r_{i-1} := \bullet(\ell_{i-1}\,2^p + h_i)$ is also exact. Moreover, $|r_{i-1}| < {}^{1}\!/_{2} + C\,2^{s-p} + 2^{-p-1}$ if $i > 1$ and $|r_{i-1}| < 2^{\delta}$ if $i = 1$. The last operation $r_{k-1} := \bullet(\ell_{k-1}\,2^p)$ is clearly exact and $|r_{k-1}| = |\ell_{k-1}\,2^p| \leqslant {}^{1}\!/_{2}$. Finally,

$$\begin{aligned}
\sum_{i=0}^{k-1} r_i\,2^{-pi} &= \sum_{i=1}^{k-1} r_{i-1}\,2^{-p(i-1)} + r_{k-1}\,2^{-p(k-1)} \\
&= \sum_{i=1}^{k} \ell_{i-1}\,2^{p-p(i-1)} + \sum_{i=1}^{k-1} h_i\,2^{-p(i-1)} \\
&= \ell_0\,2^p + \sum_{i=1}^{k-1} \ell_i\,2^{-p(i-1)} + \sum_{i=1}^{k-1} h_i\,2^{-p(i-1)}
\end{aligned}$$

$$= x_0\, 2^s + \sum_{i=1}^{k-1} x_i\, 2^{s-pi}$$
$$= 2^s \sum_{i=0}^{k-1} x_i\, 2^{-pi}.$$

This proves that $r = x\, 2^s$. □

PROPOSITION 9. *Let $s \in \{0, ..., p\}$. Given a fixed-point number $x \in \mathcal{F}_{p,k;C}$ with $C \leqslant 2^{\delta+s-2}$ and $|x_0| < \min(2^{\delta+s-2}, 2^\delta)$, the algorithm* **SmallShiftRight** *returns $r \in \mathcal{F}_{p,k;1/2+C2^{-s}+2^{-p-1}}$ with $|r - x\, 2^{-s}| \leqslant 2^{-pk-1}$.*

Proof. Similar to the proof of Proposition 8. □

3.2. Large parallel shifts

For shifts by $s = \sigma\, p$ bits with $\sigma < k$, we may directly shift the coefficients x_i of the operand. Let $\sigma = \sigma_0 + \sigma_1\, 2 + \cdots + \sigma_{\ell-1}\, 2^{\ell-1}$ be the binary representation of σ with $\sigma_0, ..., \sigma_{\ell-1} \in \{0, 1\}$. Then we decompose a shift by $\sigma\, p$ bits as the composition of ℓ shifts by either 0 or $2^i p$ bits for $i = 0, ..., \ell - 1$, depending on whether $\sigma_i = 0$ or $\sigma_i = 1$. This way of proceeding has the advantage of being straightforward to parallelize, assuming that we have an instruction to extract a new SIMD vector from two given SIMD vectors according to a mask. On INTEL processors, there are several "blend" instructions for this purpose. In the pseudo-code below, we simply used "if expressions" instead.

Algorithm LargeShiftLeft(x, σ)

```
d := 1
while d < k do
    b := (σ and d) ≠ 0
    for i from 0 to k − 1 − d do
        x_i := (if b then x_{i+d} else x_i)
    for i from k − d to k − 1 do
        x_i := (if b then 0 else x_i)
    d := 2 d
return [x_0, ..., x_{k−1}]
```

Algorithm LargeShiftRight(x, σ)

```
d := 1
while d < k do
    b := (σ and d) ≠ 0
    for i from k − 1 down to d do
        x_i := (if b then x_{i−d} else x_i)
    for i from d − 1 down to 0 do
        x_i := (if b then 0 else x_i)
    d := 2 d
return [x_0, ..., x_{k−1}]
```

The following propositions are straightforward to prove.

PROPOSITION 10. *Let $s \in \{0, p, ..., (k-1)\, p\}$. Given a fixed-point number $x \in \mathcal{F}_{p,k;C}$ with $x_0 = \cdots = x_{s/p-1} = 0$, the algorithm* **LargeShiftLeft** *returns $r \in \mathcal{F}_{p,k;C}$ with $r = x\, 2^s$.*

PROPOSITION 11. *Let $s \in \{0, p, ..., (k-1)\, p\}$. Given a fixed-point number $x \in \mathcal{F}_{p,k;C}$ with $|x_0| < C$, the algorithm* **LargeShiftRight** *returns $r \in \mathcal{F}_{p,k;C}$ with $|r - x\, 2^{-s}| < C\,(1 + 2^{1-p})\, 2^{-pk}$.*

Combining with the algorithms from the previous subsection, we obtain the routines **ShiftLeft** and **ShiftRight** below for general shifts. Notice that we shift by at most $k\, p$ bits. Due to the fact that we allow for nail bits, the maximal error is bounded by $(C+1)\, 2^{-pk}$.

Algorithm ShiftLeft(x, s)

```
σ := min(⌊s / p⌋, k − 1)
s := min(p, s − σ p)
x := LargeShiftLeft(x, σ)
x := SmallShiftLeft(x, s)
return x
```

Algorithm ShiftRight(x, s)

```
σ := min(⌊s / p⌋, k − 1)
s := min(p, s − σ p)
x := LargeShiftRight(x, σ)
x := SmallShiftRight(x, s)
return x
```

3.3. Uniform parallel shifts

The routines **LargeShiftLeft** and **LargeShiftRight** were designed to work for SIMD vectors $\boldsymbol{x} = (x^1, ..., x^w)$ of fixed-point numbers and shift amounts $\boldsymbol{\sigma} = (\sigma^1, ..., \sigma^w)$. If the number of bits by which we shift is the same $\sigma = \sigma^1 = \cdots = \sigma^w$ for all entries, then we may use the following routines instead:

Algorithm UniformShiftLeft($\boldsymbol{x}, \sigma$)

for i **from** 0 **to** $k-1$ **do**
  $j := i + \sigma$
  $x_i := (\textbf{if } j < k \textbf{ then } x_j \textbf{ else } 0)$
return $[x_0, ..., x_{k-1}]$

Algorithm UniformShiftRight($\boldsymbol{x}, \sigma$)

for i **from** $k-1$ **down to** 0 **do**
  $j := i - \sigma$
  $x_i := (\textbf{if } j \geqslant 0 \textbf{ then } x_j \textbf{ else } 0)$
return $[x_0, ..., x_{k-1}]$

3.4. Retrieving the exponent

The IEEE-754 standard provides an operation logb for retrieving the exponent e of a machine number $x \in \mathbb{F}$: if $x \neq 0$, then $2^e \leqslant |x| < 2^{e+1}$. It is natural to ask for a similar function on $\mathcal{F}_{p,w}$, as well as a similar function logw in base 2^p (that returns the exponent e with $2^{pe} \leqslant |x| < 2^{pe+1}$ for every $x \in \mathcal{F}_{p,k}$ with $x \neq 0$). For $x = 0$, we understand that $\mathrm{logb}(x) = \mathrm{logw}(x) = -\infty$.

The functions **LogB** and **LogW** below are approximations for such functions logb and logw. More precisely, for all $x \in \mathcal{F}_{k,p;C}$ in carry normal form with $|x| < 1$, we have

$$\textbf{LogB}(x) - 1 \leqslant \mathrm{logb}(x) \leqslant \textbf{LogB}(x) \tag{2}$$
$$\textbf{LogW}(x) - 1 \leqslant \mathrm{logw}(x) \leqslant \textbf{LogW}(x). \tag{3}$$

The routine **LogB** relies on the computation of a number $\Sigma = \textbf{Compress}(x) \in \mathbb{F}$ with $|\Sigma - x| \leqslant |x|\, 2^{-\mu}$ that only works under the assumption that $k\,p < -E_{\min}$. It is nevertheless easy to adapt the routine to higher precisions by cutting x into chunks of $\lfloor -E_{\min}/\,p \rfloor$ machine numbers. The routine **LogW** relies on a routine **HiWord** that determines the smallest index i with $x_i \neq 0$.

Algorithm Compress($\boldsymbol{x}$)

$\Sigma := \circ(x_0 + x_1\, 2^{-p})$
for i **from** 2 **to** $k-1$ **do**
  $\Sigma := \circ(\Sigma + x_i\, 2^{-ip})$
return Σ

Algorithm HiWord($\boldsymbol{x}$)

$r := \infty$
for i **from** $k-1$ **down to** 0 **do**
  $r := (\textbf{if } x_i = 0 \textbf{ then } r \textbf{ else } i)$
return r

Algorithm LogB($\boldsymbol{x}$)

return $\mathrm{logb}(\circ(\textbf{Compress}(x)\,(1 + 2^{-\mu})))$

Algorithm LogW($\boldsymbol{x}$)

return $-1 - \textbf{HiWord}(x)$

PROPOSITION 12. *The routines* **Compress**, **HiWord**, **LogB** *and* **LogW** *are correct.*

Proof. In **Compress**, let Σ_i be the value of Σ after adding $x_i\, 2^{-ip}$ and notice that $\Sigma_i \in 2^{-pi}$ for all i. If $\Sigma_{i+1} = \Sigma_i + x_i\, 2^{-ip}$ for all i, then $\Sigma_k = x$ and **Compress** returns an exact result. Otherwise, let i be minimal such that $\Sigma_{i+1} \neq \Sigma_i + x_i\, 2^{-ip}$. Then $|\Sigma_i + x_i\, 2^{-ip}| > 2^{\mu+1-(i+1)p}$ and $|\Sigma_{i+1}| \geqslant 2^{\mu+1-(i+1)p}$, whence $|x_i\, 2^{-ip}| < {}^4\!/_5\; 2^{-\delta-1}\, |\Sigma_{i+1}|$ and $|\Sigma_{i+1} - (\Sigma_i - x_i\, 2^{-ip})| \leqslant 2^{-\mu-1}\, |\Sigma_{i+1}|$. Moreover, the exponent e of Σ_{i+1} is at least $\mu + 1 - (i+1)\, p$. For $j > i$, we have $|x_j\, 2^{-jp}| < {}^4\!/_5\; 2^{-jp}$, whence the exponent f of $x_j\, 2^{-jp}$ is at most $-(i+1)\, p - 1$. This means that $\circ(\Sigma_{i+1} + x_j\, 2^{-jp}) = \Sigma_{i+1}$, whence $\Sigma_{i+1} = \Sigma_{i+2} = \cdots = \Sigma_k$ and $|\Sigma_k - x| \leqslant |\Sigma_{i+1} - (\Sigma_i - x_i\, 2^{-ip})| + |x_{i+1}\, 2^{-(i+1)p} + \cdots + x_{k-1}\, 2^{-(k-1)p}| < 2^{-\mu-1}\, |\Sigma_k| + 2^{-(i+1)p} \leqslant 2^{-\mu}\, |\Sigma_k|$. The bound (2) directly follows.

The algorithm **HiWord** is clearly correct. Assume that $x \neq [0, ..., 0]$ and let i be minimal with $x_i \neq 0$. Then we have $|x_i\, 2^{-ip}| \geqslant 2^{-(i+1)p}$, whereas $|x_{i+1}\, 2^{-(i+1)p} + \cdots + x_{k-1}\, 2^{-(k-1)p}| < {}^5\!/_6\, 2^{-(i+1)p}$, so that ${}^1\!/_6\, 2^{-(i+1)p} < |x|$. If $i > 0$, then we also get $|x_i\, 2^{-ip}| < {}^4\!/_5\, 2^{-ip}$, whence $|x| < {}^5\!/_6\, 2^{-ip}$. If $i = 0$, then $|x| < 1$ by assumption. This shows that (3) holds as well. □

3.5. Operation counts

In Table 1 below, we have shown the number of machine operations that are required for the fixed-point operations from this and the previous section, as a function of k. Of course, the actual time complexity of the algorithms also depends on the latency and throughput of machine instructions. We count an assignment $z := (\textbf{if } b \textbf{ then } x \textbf{ else } y)$ as a single instruction.

k	2	3	4	5	6	7	8	9	10	11	12
Carry normalize	4	8	12	16	20	24	28	32	36	40	44
Add/subtract	2	3	4	5	6	7	8	9	10	11	12
Multiply	8	18	33	53	78	108	143	183	228	278	333
Small shift	7	11	15	19	23	27	31	35	39	43	47
Large shift	3	8	10	18	21	24	27	40	44	48	52
General shift	12	21	27	39	45	53	60	77	85	92	100
Uniform shift	4	6	8	10	12	14	16	18	20	22	24
Bit exponent	3	4	5	6	7	8	9	10	11	12	13
Highest word	3	4	5	6	7	8	9	10	11	12	13

Table 1. Operation counts in terms of machine instructions (the results are not necessarily normalized).

4. Floating-point arithmetic in base 2

Let p, k and δ be as in section 2. We will represent floating-point numbers as products

$$x \;=\; m\, b^e,$$

where $m \in \mathcal{F}_{p,k}$ is the *mantissa* of x and $e \in \mathcal{E} := \{-2^\mu, ..., 2^\mu\}$ its *exponent* in base $b \in 2^{\mathbb{N}}$. We denote by $\mathcal{F}_{p,k}\, b^{\mathcal{E}}$ the set of all such floating-point numbers. We assume that the exponents in $\mathcal{E}$ can be represented exactly, by machine integers or numbers in $\mathbb{F}$. As in most existing multiple precision libraries, we use an extended exponent range. For multiple precision arithmetic, it is indeed natural to support exponents of at least as many bits as the precision itself.

In this section, we assume that $b = 2$. The main advantage of this base is that floating-point numbers can be represented in a compact way. The main disadvantage is that normalization involves expensive shifts by general numbers of bits that are not necessarily multiples of p. Notice that $b = 2$ is also used in the Mpfr library for multiple precision floating-point arithmetic [6].

4.1. Normalization

Given ${}^1\!/_2 \leqslant C \leqslant 2^\delta$, we denote $\mathcal{F}_{p,k;C}\, 2^{\mathcal{E}} = \{m\, 2^e : m \in \mathcal{F}_{p,k;C}, e \in \mathcal{E}\}$. Numbers in $\mathcal{F}_{p,k;{}^4\!/_5}\, 2^{\mathcal{E}}$ are again said to be in *carry normal form* and the routine **CarryNormalize** extends to $\mathcal{F}_{p,k}\, 2^{\mathcal{E}}$ by applying it to the mantissa. We say that a floating-point number $x = m\, 2^e$ with $m \in \mathcal{F}_{p,k}$ is in *dot normal form* if we either have $m = 0$ or ${}^1\!/_2 - 2^{-p} \leqslant |m| < 1$. If x is also in carry normal form, then we simply say that x is in *normal form*. In absence of overflows, normalizations can be computed using the routine **Normalize** below. In the case when the number to be normalized is the product of two normalized floating-point numbers, we need to shift the mantissa by at most two bits, and it is faster to use the variant **QuickNormalize**.

Algorithm Normalize($m\,2^e$)

$m' := \mathbf{CarryNormalize}(m)$
$s := -1 - \mathbf{LogB}(m')$
$m' := \mathbf{ShiftLeft}(m', s)$
$e' := e - s$
$e' := \max(e', -2^\mu)$
$e' := (\mathbf{if}\ m'_0 = 0\ \mathbf{then}\ -2^\mu\ \mathbf{else}\ e')$
return $m'\,2^{e'}$

Algorithm QuickNormalize($m\,2^e$)

$s := -1 - \mathbf{LogB}([m_0, m_1])$
$m' := \mathbf{SmallShiftLeft}(m, s)$
$e' := e - s$
$e' := \max(e', -2^\mu)$
return $m'\,2^{e'}$

PROPOSITION 13. *Given $m\,2^e \in \mathcal{F}_{p,k}\,2^{\mathcal{E}}$ with $|m| < 1$ and $e > p\,k - 2^\mu$, the algorithm* **Normalize** *returns a normal number $m'\,2^{e'} \in \mathcal{F}_{p,k}\,2^{\mathcal{E}}$ with $m'\,2^{e'} = m\,2^e$. If $|m| \geqslant 2^{\delta+2-p}$, then so does* **QuickNormalize**.

Proof. If $m = 0$, then **CarryNormalize** returns [0, ..., 0] and it is easy to verify that the proposition holds. Assume therefore that $m \neq 0$, so that $s < k\,p$. By Proposition 4, we have $\mathbf{CarryNormalize}(m) \in \mathcal{F}_{p,k;1/2+2^{-p}}$. Using Proposition 8, it follows that $m' \in \mathcal{F}_{p,k;C}$ with $C = 1/2 + (1/2 + 2^{-p})\,2^{(s \operatorname{rem} p)-p} + 2^{-p-1} \leqslant 3/4 + 2^{-p} < 4/5$, whence m' is carry normal. We also have $m\,2^e = m'\,2^{e-s}$ and $2^{1-s}/(1 + 2^{-\mu}) \leqslant |m'| < 2^{-s}$, whence $1/2 - 2^{-p} \leqslant |m'| < 1$. This also implies $|m'| \neq 0$ and $e' = e - s$, since $|m' - m'_0| \leqslant C\,2^{-p} + \cdots + C\,2^{-(k-1)p}$. We conclude that $m'\,2^{e'}$ is dot normal and its value coincides with $m\,2^e$. If $|m| \geqslant 2^{\delta+2-p}$, then it can be checked that $\varphi := \mathbf{Compress}([m_0, m_1])$ still satisfies $|\varphi - m| \leqslant |m|\,2^{-\mu}$ and that $s \leqslant p - 2$. From Proposition 8, it again follows that $m' \in \mathcal{F}_{p,k;C}$ with $C = 1/2 + (1/2 + 2^{-p})\,2^{-2} + 2^{-p-1} < 4/5$, whence m' is carry normal. The remainder of the correctness of **QuickNormalize** follows as before. □

4.2. Arithmetic operations

Addition and subtraction of normalized floating-point numbers are computed as usual, by putting the mantissa under a common exponent, by adding or subtracting the shifted mantissas, and by normalizing the result. By increasing the common exponent by two, we also make sure that the most significant word of the addition or subtraction never provokes a carry.

Algorithm Add($m_1\,2^{e_1}, m_2\,2^{e_2}$)

$e := \max(e_1, e_2) + 2$
$m'_1 := \mathbf{ShiftRight}(m_1, e - e_1)$
$m'_2 := \mathbf{ShiftRight}(m_2, e - e_2)$
$m := \mathbf{Add}(m'_1, m'_2)$
return $\mathbf{Normalize}(m\,2^e)$

Algorithm Subtract($m_1\,2^{e_1}, m_2\,2^{e_2}$)

$e := \max(e_1, e_2) + 2$
$m'_1 := \mathbf{ShiftRight}(m_1, e - e_1)$
$m'_2 := \mathbf{ShiftRight}(m_2, e - e_2)$
$m := \mathbf{Subtract}(m'_1, m'_2)$
return $\mathbf{Normalize}(m\,2^e)$

Floating-point multiplication is almost as efficient as its fixed-point analogue, using the following straightforward:

Algorithm Multiply($m_1\,2^{e_1}, m_2\,2^{e_2}$)

$m := \mathbf{Multiply}(m_1, m_2)$
$e := e_1 + e_2$
return $\mathbf{QuickNormalize}(m\,2^e)$

Remark 14. Strictly speaking, for normal numbers $m_1\,2^{e_1}$ and $m_2\,2^{e_2}$, it is still necessary to prove that $|\mathbf{Multiply}(m_1, m_2)| < 1$. This conclusion can only be violated if $|m_1|$ and $|m_2|$ are very close to one. Now it can be shown that a carry normal mantissa m with $m < 1$ and $1 - 2^{(p-1)-kp} < m < 1$ is necessarily of the form $m = [1, 0, ..., 0, -\varepsilon]$. For numbers of this form, it is easy to check that $|\mathbf{Multiply}(m_1, m_2)| < 1$.

4.3. Shared exponents

If we are computing with an SIMD vector $\boldsymbol{m}\,2^{\boldsymbol{e}} = (m^1\,2^{e^1}, ..., m^w\,2^{e^w})$ of floating-point numbers, then another way to speed up shifting is to let all entries share the same exponent $e = e^1 = \cdots = e^w$. Of course, this strategy may compromise the accuracy of some of the computations. Nevertheless, if all entries are of similar order of magnitude, then the loss of accuracy is usually limited to a few bits. Moreover, by counting the number of leading zeros of the mantissa of a particular entry $m^i\,2^e$, it is possible to monitor the loss of accuracy, and redo some of the computations if necessary.

When sharing exponents, it becomes possible to use **UniformShiftLeft** and **UniformShiftRight** instead of **LargeShiftLeft** and **LargeShiftRight** for multiple word shifts. The routine logb should also be adjusted: if the individual exponents given by the vector $\boldsymbol{e} = (e^1, ..., e^w)$, then logb should now return the vector $(e, ..., e)$ with $e = \max(e^1, ..., e^w)$. Modulo these changes, we may use the same routines as above for basic floating-point arithmetic.

4.4. Operation counts

In Table 2, we give the operation counts for the various variants of the basic arithmetic operations $+$, $-$ and $\times$ that we have discussed so far. For comparison, we have also shown the operation count for the algorithm from [14] that is based on floating-point expansions (notice that $p = \mu$ is somewhat better for this algorithm, since our algorithms rather use $p \approx \mu - 4$).

Due to the cost of shifting, we observe that additions and subtractions are the most costly operations for small and medium precisions $k \leqslant 8$. For larger precisions $k \geqslant 12$, multiplication becomes the main bottleneck.

	k	2	3	4	5	6	7	8	9	10	11	12
$\pm$	Individual exponents	51	84	108	150	177	204	231	288	318	348	378
	Shared exponents	55	79	103	127	151	175	199	223	247	271	295
$\times$	Individual exponents	31	35	54	78	107	141	180	224	273	327	386
	Shared exponents	32	36	55	79	108	142	181	225	274	328	387
$\times$	FP expansions	138	193	261	342	436	543	663	796	942	1101	1273

Table 2. Operation counts for arithmetic floating-point operations in base 2.

5. Floating-point arithmetic in base 2^p

As we can see in Table 2, additions and subtractions are quite expensive when working in base $b = 2$. This is due to the facts that shifting is expensive with respect to this base and that every addition involves three shifts. For this reason, we will now examine floating-point arithmetic in the alternative base $b = 2^p$. One major disadvantage of this base is that normalized non zero floating-point numbers may start with as many as $p - 1$ leading zero bits. This means that k should be increased by one in order to achieve a similar accuracy. We notice that the base $b = 2^p$ is also used in the native `mpf` layer for floating-point arithmetic in the Gmp library [7].

Carry normal forms are defined in the same way as in base $b = 2$. We say that a floating-point number $x = m\,2^{ep}$ with $m \in \mathcal{F}_{p,k}$ is in *dot normal form* if $|m| < 4/5$ and either $m = 0$ or $m_0 \neq 0$. We say that x is in *normal form* if it is both in carry normal form and in dot normal form.

5.1. Addition and subtraction

In section 4.2, we increased the common exponent of the summands of an addition by two in order to avoid carries. When working in base 2^p, a similar trick would systematically shift out the least significant words of the summands. Instead, it is better to allow for carries, but to adjust the routine for normalization accordingly, by temporarily working with mantissas of $k + 1$ words instead of k.

Algorithm Normalize($m\,2^{ep}$)

$m' := [0, m_0, ..., m_{k-1}]$
$m' := \mathbf{CarryNormalize}(m')$
$\sigma := \mathbf{HiWord}(m')$
$m' := \mathbf{LargeShiftLeft}(m', \sigma)$
$e' := e + 1 - \sigma$
$e' := \max(e', -2^{\mu})$
$e' := (\mathbf{if}\ m'_0 = 0\ \mathbf{then}\ -2^{\mu}\ \mathbf{else}\ e')$
return $[m'_0, ..., m'_{k-1}]\,2^{e'p}$

Modulo this change, the routines for addition and subtract are now as follows:

Algorithm Add($m_1\,2^{e_1 p}, m_2\,2^{e_2 p}$)

$e := \max(e_1, e_2)$
$m'_1 := \mathbf{LargeShiftRight}(m_1, e - e_1)$
$m'_2 := \mathbf{LargeShiftRight}(m_2, e - e_2)$
$m := \mathbf{Add}(m'_1, m'_2)$
return $\mathbf{Normalize}(m\,2^{ep})$

Algorithm Subtract($m_1\,2^{e_1 p}, m_2\,2^{e_2 p}$)

$e := \max(e_1, e_2)$
$m'_1 := \mathbf{LargeShiftRight}(m_1, e - e_1)$
$m'_2 := \mathbf{LargeShiftRight}(m_2, e - e_2)$
$m := \mathbf{Subtract}(m'_1, m'_2)$
return $\mathbf{Normalize}(m\,2^{ep})$

5.2. Multiplication

The dot normalization of a product becomes very particular when working in base 2^p since this can always be accomplished using a large shift by either 0 or p or $2\,p$ bits. Let **LargeShiftLeft*** be the variant of **LargeShiftLeft** obtained by replacing the condition $d < k$ by $d \leqslant 2$. In order to achieve an accuracy of about $(k-1)\,p$ bits at least, we extend the mantissas by one machine coefficient before multiplying them. The routine **QuickNormalize** suppresses the extra entry.

Algorithm QuickNormalize($m\,2^{ep}$)

$m' := \mathbf{CarryNormalize}(m)$
$\sigma := \mathbf{HiWord}([m_0, m_1, m_2])$
$m' := \mathbf{LargeShiftLeft}^*(m', \sigma)$
$e' := e - \sigma$
$e' := \max(e', -2^{\mu})$
return $[m'_0, ..., m'_{k-1}]\,2^{e'p}$

Algorithm Multiply($m_1\,2^{e_1 p}, m_2\,2^{e_2 p}$)

$m'_1 := [(m_1)_0, ..., (m_1)_{k-1}, 0]$
$m'_2 := [(m_2)_0, ..., (m_2)_{k-1}, 0]$
$m := \mathbf{Multiply}(m'_1, m'_2)$
$e := e_1 + e_2$
return $\mathbf{QuickNormalize}(m\,2^{ep})$

5.3. Operation counts

In Table 2, we give the operation counts for floating-point addition, subtraction and multiplication in base 2^p. In a similar way as in section 4.3, it is possible to share exponents, and the table includes the operation counts for this strategy. This time, additions and subtractions are always cheaper than multiplications.

	$k-1$	2	3	4	5	6	7	8	9	10	11	12
$\pm$	Individual exponents	31	49	67	92	107	122	147	183	201	219	237
	Shared exponents	31	43	55	67	79	91	103	115	127	139	151
$\times$	Individual exponents	40	61	87	118	154	195	241	292	348	409	475
	Shared exponents	41	62	88	119	155	196	242	293	349	410	476
$\times$	FP expansions	138	193	261	342	436	543	663	796	942	1101	1273

Table 3. Operation counts for arithmetic floating-point operations in base 2^p.

Bibliography

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

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

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

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

[5] N. Emmart, J. Luitjens, C. C. Weems, and C. Woolley. Optimizing modular multiplication for nvidia's maxwell gpus. In Paolo Montuschi, Michael Schulte, Javier Hormigo, Stuart Oberman, and Nathalie Revol, editors, *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, pages 47–54. IEEE, 2016.

[6] 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.

[7] T. Granlund et al. GMP, the GNU multiple precision arithmetic library. http://gmplib.org, from 1991.

[8] D. Harvey, J. van der Hoeven, and G. Lecerf. Even faster integer multiplication. *Journal of Complexity*, 36:1–30, 2016.

[9] 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.

[10] J. van der Hoeven and G. Lecerf. Faster FFTs in medium precision. In *22nd Symposium on Computer Arithmetic (ARITH)*, pages 75–82, June 2015.

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

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

[13] 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.

[14] Jean-Michel Muller, Valentina Popescu, and Ping Tak Peter Tang. A new multiplication algorithm for extended precision using floating-point expansions. In Paolo Montuschi, Michael Schulte, Javier Hormigo, Stuart Oberman, and Nathalie Revol, editors, *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, pages 39–46, 2016.

[15] 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.

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

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

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

[19] D. Takahashi. Implementation of multiple-precision floating-point arithmetic on Intel Xeon Phi coprocessors. In *Computational Science and Its Applications – ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part II*, pages 60–70, Cham, 2016. Springer International Publishing.