Fast basecases for arbitrary-size multiplication

Albin Ahlbäck
CNRS, LIX (UMR 7161)
Palaiseau, France
ahlback@lix.polytechnique.fr

Fredrik Johansson
Inria, IMB (UMR 5251)
Bordeaux, France
fredrik.johansson@gmail.com

***Abstract*—Multiple precision libraries typically use assembly-optimized loops for basecase operations on variable-length operands. We consider the alternative of generating lookup tables with hardcoded routines for many fixed sizes, e.g. for all multiplications up to 16 by 8 words. On recent ARM64 and x86-64 CPUs, we demonstrate up to a 2x speedup over GMP for basecase-sized multiplication and a 20% speedup for Karatsuba-sized operands. We pay special attention to the computation of approximate products and demonstrate up to a 3x speedup over GMP/MPFR for floating-point multiplication.**

***Index Terms*—integer multiplication, multiple precision, bignum arithmetic, floating-point arithmetic, performance**

I. Introduction

We consider arithmetic on natural numbers in the standard form $\sum_{i=0}^{n-1} c_i \beta^i$ with $0 \le c_i < \beta$, where we assume a full-word radix $\beta = 2^B$, $B = 64$, on a typical 64-bit machine. A popular software implementation allowing operands with variable size n is the low-level `mpn` layer of GMP [6].

The most performance-critical operation is arguably multiplication which for $n \gg 1$ is much more expensive than addition and also serves as the backbone for fast implementations of higher operations such as division [17]. Given operands $a = \sum_{i=0}^{m-1} a_i \beta^i$ and $b = \sum_{i=0}^{n-1} b_i \beta^i$ where we may assume $m \ge n \ge 1$, the schoolbook multiplication algorithm [14, §4.3.1] forms the product

$$c = \sum_{i=0}^{m+n-1} c_i \beta^i = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_i b_j \beta^{i+j}$$

using mn single-word multiplications (with double-word output) interleaved with additions with carry-in and carry-out.

The asymptotic complexity of multiplication is improved by using Karatsuba, Toom-Cook or FFT algorithms [8], but such methods only beat the schoolbook method for large m and their performance depends on having an efficient schoolbook multiplication as the "basecase" for recursive smaller multiplications.[1] In GMP, the general-purpose multiplication routine `mpn_mul` dispatches to the internal schoolbook multiplication routine `mpn_mul_basecase` for all sizes below the machine-dependent Karatsuba threshold.

Albin Ahlbäck has been supported by an ERC-2023-ADG grant for the ODELIX project (number 101142171). Fredrik Johansson has been supported by the French ANR-20-CE-48-0014 NuSCAP.

[1]Some FFT multiplication algorithms are an exception, where it is preferable to do all internal computations with single-word precision if possible. However, the widely used (e.g. by GMP) Schönhage-Strassen FFT algorithm specifically recurses to multi-word integers.

Basecase GMP routines are generally implemented in assembly, using techniques such as partial unrolling and instruction reordering to optimize out-of-order execution. However, most basecase routines are implemented as loops to support variable m and n, and the overhead of this approach for few-word operands can be considerable. In this work, we study the alternative of generating tables of specialized routines for all size pairs (m, n) roughly up to the Karatsuba threshold (roughly $m \approx n \approx 16$ for balanced multiplication).

A. Overview of our contribution

Section II describes our approach to generate efficient basecase multiplication routines for ARM64 and x86-64 assembly. We benchmark these routines for fixed (m, n), demonstrating significant speedups (roughly a factor two in some cases) over the basecase multiplication in GMP 6.3.0.

Section III discusses the impact of using such basecases in a general-purpose multiplication routine allowing variable sizes and sizes beyond the basecase range, i.e. as a drop-in replacement for GMP's `mpn_mul`.

Section IV explores the analogous optimization of short (truncated) products, with application to dot products and matrix multiplication in multiple precision floating-point arithmetic. We study properties of approximate high products in detail to obtain a general algorithmic framework with rigorous error analysis. We demonstrate significant speedups (roughly a factor three in some cases) over MPFR 4.2.1 [5].

Our implementations are freely available under the LGPL 3 license and have been contributed to the development version 3.2.0-dev of FLINT [19].

B. Restrictions and related work

Our goal is to optimize serial, scalar operations in a way that remains efficient for few-word and mixed-size operands, for the benefit of applications such as FLINT which rely heavily on low-level GMP functions and the associated data model.

The technique of hardcoding arithmetic routines for specific numbers of words is common in cryptography where one works over $\mathbb{Z}/N\mathbb{Z}$ for some fixed bit size of N, and where side-channel resistance is a significant concern aside from raw performance. Examples of cryptography-oriented multiple precision libraries and code generators include [1], [4]. However, there seem to have been few attempts to base general-purpose, variable-size arithmetic routines on such basecases.

We do not study SIMD techniques in this work, which under favorable conditions can achieve even greater speedups. For example, using AVX512-IFMA and conversion to a carry-save representation, Edamatsu and Takahashi [3] report breaking even with GMP's multiplication around 1024 bits and achieving speedup factors around 2-3 for operands larger than 3072 bits while Didier et al. [2] report speedup factors around 4-5 over GMP for batched 1024-bit operations. For extreme (e.g. million-digit) precision, state of the art NTT multiplication uses SIMD [21]. Some operations such as matrix multiplication can benefit even further by performing batched operations in a residue number system. A common limitation in these examples is that one needs either quite large operands, data conversions or batched operations on many fixed-size numbers for SIMD to pay off due to the restrictions of present-day instruction sets (e.g. lack of vectorized carry handling); in the setting of multiple precision arithmetic, SIMD and non-SIMD approaches therefore remain complementary.

II. Multiplication implementation details

When implementing multiplication routines for ARM64 and modern x86-64 processors, we note the following. Modern x86-64 processors are equipped with the ADX instruction set, allowing two carry chains to be executed concurrently. While this instruction set is useful, x86-64 suffers from its two-address instruction format with only 15 usable general-purpose registers. On the contrary, ARM64 has a three-address instruction format with 29 usable general-purpose registers, which speaks in favor of ARM64.

For straight-line programs, i.e. functions without loops or branches, the decoder of the processor remains an important aspect. The modern Zen 4 architecture is only able to decode four instructions per cycle from the instruction cache, or six instructions per cycle assuming optimal usage of the operation cache. Now, consider the $\mathcal{O}(n)$ operation $r \leftarrow a - 2^e b$ on an x86-64 system displayed in Fig. 1. As this algorithm grows by seven instructions per word, the Zen 4 decoder limits the upper performance of this algorithm between $7/6$ to $7/4$ cycles per word, somewhat far from the optimal one cycle per word. In contrast, Apple M1 is not limited by its decoder in these aspects, with the ability to decode eight instructions from the instruction cache per cycle [13].

```
shrx    e, s1, s0      // Assume s1 = b[ix]
mov     (i+1)*8(b), s1
shlx    f, s1, s2      // f = 64 - e
lea     (s0, s2), s2
mov     i*8(a), s0
sbb     s2, s0
mov     s0, i*8(r)
```

Fig. 1. The i-th step of a straight-line program for the operation $r \leftarrow a - 2^e b$ for $0 < e < 64$ on x86-64 architecture.

Basecase multiplication, however, is typically constrained by the number of multiplication ports and the dependency chains arising from the carry chains. Specifically, the decoder should not be a problem for x86-64 architectures since the routine consists of chains of mulx, adcx, adox instructions, where an optimal routine will consume one cycle for a section of these three instructions.

For full multiplication on ARM64, we chose a semi-unrolled approach, whose algorithm is displayed in Fig. 2 and was handwritten in assembly.

```
function mul_$m$_$n$
    Input: $c$: pointer to at least $m + n$ allocated words,
           $a = \sum_{i=0}^{m-1} a_i \beta^i$, $b = \sum_{i=0}^{n-1} b_i \beta^i$,
           $m \in \mathbb{Z}_{\geq 2}$: compile-time constant,
           $n \in \mathbb{Z}_{\geq 2}$
    Output: $c \leftarrow a \cdot b$
    $(x_0, \ldots, x_{m-1}) \leftarrow (a_0, \ldots, a_{m-1})$
    $\langle c_0, r_0, \ldots, r_{m-1} \rangle \leftarrow \langle x_0, \ldots, x_{m-1} \rangle \cdot b_0$
    for $j \leftarrow 1$ to $n - 1$ do
        for $i \leftarrow 0$ to $m - 1$ do
            $s_i \leftarrow x_i \cdot b_j \mod \beta$            // mul
        end
        $s_m \leftarrow \lfloor x_m \cdot b_j / \beta \rfloor$              // umulh
        $\langle c_j, s_1, \ldots, s_{m-1}, r_{m-1} \rangle \leftarrow$
            $\langle s_0, \ldots, s_{m-1}, s_m \rangle + \langle r_0, \ldots, r_{m-1}, 0 \rangle$
        for $i \leftarrow 0$ to $m - 2$ do
            $r_i \leftarrow \lfloor x_i \cdot b_j / \beta \rfloor$              // umulh
        end
        $\langle r_0, \ldots, r_{m-1} \rangle \leftarrow$
            $\langle s_1, \ldots, s_{m-1}, r_{m-1} \rangle + \langle r_0, \ldots, r_{m-2}, 0 \rangle$
    end
    $(c_n, \ldots, c_{m+n-1}) \leftarrow (r_0, \ldots, r_{m-1})$
end
```

Fig. 2. Semi-unrolled basecase multiplication suited for ARM64 architectures with the notation $\langle r_0, \ldots, r_n \rangle = \sum_{i=0}^{n} r_i \beta^i$ where $0 \leq r_i < \beta$.

For full multiplication on x86-64, the two-address instruction format requires complete unrolling to avoid intermediate loads, stores and moves; the implemented algorithm for x86-64 can be seen in Fig. 3 and was generated from a script.

A. Block decomposition

Eventually m and n become too large for a fully unrolled schoolbook multiplication to be worthwhile. For one thing, efficiency will degrade when we run out of registers and need to start writing to the stack; more importantly, extremely large unrolled loops will start to incur instruction cache misses.

In this case, one option is to fall back to a loop-based implementation like mpn_mul_basecase. An alternative is to split the operands blockwise and call the fully unrolled basecases for smaller sizes. There are various possible splittings of an (m, n) product, for example:

- Peeling: $(m_0, n_0) + (m - m_0, n - n_0)$ where (m_0, n_0) is the largest available fully unrolled block,
- Balanced: $(m, \lfloor n/2 \rfloor) + (m, \lceil n/2 \rceil)$ or $(\lfloor m/2 \rfloor, n) + (\lceil m/2 \rceil, n)$, or
- Karatsuba: three products of size about $(m/2, n/2)$.

```
function mul_$m$_$n$
    Input: $c$: pointer to at least $m+n$ allocated words,
            $a = \sum_{i=0}^{m-1} a_i\beta^i$, $b = \sum_{i=0}^{n-1} b_i\beta^i$,
            $m, n \in \mathbb{Z}_{\geq 3}$: compile-time constants
    Output: $c \leftarrow a \cdot b$
    $r_z \leftarrow 0$
    $(c_0, r_0, \ldots, r_{m-1}) \leftarrow a \cdot b_0$
    for $j \leftarrow 1$ to $n-1$ do
        $(r_{\text{scr}}, r_m) \leftarrow a_0 \cdot b_j$                // mulx
        $c_j \leftarrow r_0 + r_{\text{scr}}$                   // adcx + mov
        $r_1 \leftarrow r_1 + r_m + \text{CF}$                // adcx
        for $i \leftarrow 1$ to $m-2$ do
            $(r_0, r_{\text{scr}}) \leftarrow a_i \cdot b_j$            // mulx
            $r_i \leftarrow r_i + r_0 + \text{OF}$            // adox
            $r_{i+1} \leftarrow r_{i+1} + r_{\text{scr}} + \text{CF}$         // adcx
        end
        $(r_0, r_m) \leftarrow a_{m-1} \cdot b_j$              // mulx
        $r_{m-1} \leftarrow r_{m-1} + r_0 + \text{OF}$          // adox
        $r_m \leftarrow r_m + (r_z + \text{CF}) + (r_z + \text{OF})$
        $(r_0, \ldots, r_m) \leftarrow (r_1, \ldots, r_m, r_0)$
    end
    $(c_n, \ldots, c_{m+n}) \leftarrow (r_1, \ldots, r_m)$
end
```

Fig. 3. A basecase multiplication routine which in its unrolled and expanded form avoids intermediate loads, stores or moves. Here, `CF` and `OF` stands for the carry flag and overflow flag in the x86 architecture, respectively, and `adcx` and `adox` are instructions in the ADX instruction set.

For moderate (m, n), one can exhaustively test splitting strategies to find the optimal method for a given machine.

B. Implementation results on x86-64

On x86-64, our basecase function table covers all sizes $m, n \leq 16$. We use fully unrolled schoolbook routines up to size $m = 16$, $n = 8$, while larger cases are written as C functions that call the smaller routines. For example, we use the following decompositions for the balanced ($m = n$) cases:

- For $9 \leq n \leq 11$: peeling as $(n, 8)$ plus repeated ($n - 8$ times) $(n, 1)$ multiplications (here, we rely on GMP's `mpn_addmul_1` for fused $(m, 1)$ multiply-adds).
- For $n = 12, 14, 16$, Karatsuba with $(6, 6)$, $(7, 7)$ and $(8, 8)$ basecases.
- For $n = 13, 15$, peeling to reduce to the $(n-1, n-1)$ Karatsuba product.

Table I shows timings for our x86-64 implementation on an AMD Ryzen 7 PRO 5850U (Zen 3 architecture) with a base frequency of 1.90 GHz. To estimate cycle counts, we disabled CPU frequency boosting and sampled the `rdtsc` counter over a loop of 1024 function calls, taking the best time out of 1000 total runs. The baseline overhead of calling a no-op function in a similar loop is roughly 8 cycles per call.

We observe a significant speedup over GMP's `mpn_mul_basecase`, close to a factor two for very short products (3-5 words) and up to 30% for larger balanced products. The equivalent number of cycles per word in the schoolbook algorithm, i.e. cycles / (mn), is an interesting metric. Our largest fully unrolled schoolbook basecases ($(m, 8)$, $1 \leq m \leq 16$) achive roughly 1.36 cycles per word; the subsequent block-based versions ($n \geq 9$) perform slightly worse until we reach the $(16, 16)$ Karatsuba routine.

For reference, GMP switches from basecase to Karatsuba multiplication at 20 words on the Zen 3 architecture.

TABLE I
TIMINGS FOR BASECASE MULTIPLICATION ON X86-64 (ZEN 3).

Words		GMP (`mpn_mul_basecase`)		Ours (`flint_mpn_mul_`m`_`n)		
m	n	Cycles	C/(mn)	Cycles	C/(mn)	Speedup
Balanced ($m = n$)						
1	1	8	8.00	8	8.00	1.00
2	2	12	3.00	9	2.25	1.33
3	3	33	3.67	15	1.67	2.20
4	4	46	2.88	26	1.62	1.77
5	5	64	2.56	37	1.48	1.73
6	6	85	2.36	52	1.44	1.63
7	7	106	2.16	69	1.41	1.54
8	8	128	2.00	88	1.38	1.45
9	9	160	1.98	117	1.44	1.37
10	10	192	1.92	149	1.49	1.29
11	11	231	1.91	187	1.55	1.24
12	12	267	1.85	214	1.49	1.25
13	13	313	1.85	264	1.56	1.19
14	14	357	1.82	282	1.44	1.27
15	15	416	1.85	331	1.47	1.26
16	16	467	1.82	333	1.30	1.40
Unbalanced (some examples)						
4	2	26	3.25	14	1.75	1.86
8	1	16	2.00	14	1.75	1.14
8	4	67	2.09	47	1.47	1.43
12	1	24	2.00	21	1.75	1.14
12	4	94	1.96	80	1.67	1.18
12	8	181	1.89	130	1.35	1.39
16	1	29	1.81	27	1.69	1.07
16	4	123	1.92	110	1.72	1.12
16	8	234	1.83	174	1.36	1.34
16	12	347	1.81	286	1.49	1.21

C. Implementation results on ARM64

On ARM64, our basecase functions fix the size m and admit n as a variable. Our lookup table covers the cases $1 \leq m \leq 15$, and only employs schoolbook multiplication.

Table II shows timings for our ARM64 implementation on the Apple M1 processor. A similar strategy to obtaining the clock cycle was used, but here the special register `CNTVCT_EL0` was used to determine the virtual counter and `CNTFRQ_EL0` was used to determine its frequency.

In contrast to the results for x86-64, the speedup on ARM64 is more significant and sustained. In particular, the performance measure cycles per word is about 1.00 for all sizes which is optimal.

However, it should be noted that GMP does not implement a native basecase multiplication routine for ARM64. Instead, it falls back to `mpn_mul_1` joined with a chain of `mpn_addmul_1`, resulting in more function calls. Nonetheless, the results obtained for our implementation is as good as one can expect.

TABLE II
TIMINGS FOR BASECASE MULTIPLICATION ON ARM64 (APPLE M1).

Words		GMP (`mpn_mul_basecase`)		Ours (`flint_mpn_mul_`m`_`n)		
m	n	Cycles	C/(mn)	Cycles	C/(mn)	Speedup
Balanced $(m = n)$						
1	1	9	9.00	3	3.00	3.00
2	2	13	3.25	3	0.75	4.33
3	3	22	2.44	9	1.00	2.44
4	4	32	2.00	16	1.00	2.00
5	5	45	1.80	24	0.96	1.88
6	6	60	1.67	36	1.00	1.67
7	7	77	1.57	50	1.02	1.54
8	8	109	1.70	64	1.00	1.70
9	9	136	1.68	81	1.00	1.68
10	10	169	1.69	101	1.01	1.67
11	11	181	1.50	120	0.99	1.51
12	12	229	1.59	146	1.01	1.57
13	13	254	1.50	169	1.00	1.50
14	14	279	1.42	200	1.02	1.40
15	15	299	1.33	228	1.01	1.31

III. GENERAL-PURPOSE MULTIPLICATION

Given a table of basecase multiplication functions for all $n \le m \le m_{\max}$ ($m_{\max} = 16$ in our implementation on x86-64), we can define a general-purpose multiplication routine compatible with GMP's `mpn_mul` as follows:[2]

```
mp_word_t mul(mp_word_t * c,
   const mp_word_t * a, mp_size_t m,
   const mp_word_t * b, mp_size_t n)
{
  if (m <= m_max)
    return mul_tab[m][n](c, a, b);
  else
    return mul_fallback(c, a, m, b, n);
}
```

Here, the *fallback* subroutine dispatches to Karatsuba, Toom-Cook and other subquadratic algorithms (or some form of schoolbook multiplication for very unbalanced products).

The FLINT function `flint_mpn_mul` is defined as an inline function of this form. The inline definition allows an optimizing C/C++ compiler to eliminate the comparisons and table lookup overheads whenever m and n are known or bounded. C/C++ code making consecutive calls with fixed m and n (e.g. in a loop) is then potentially as fast as code calling the fixed-size basecase functions directly.

Table III compares the performance of GMP's `mpn_mul` with `flint_mpn_mul` *without* inlining of the latter.

We observe a speedup for basecase-sized operands similar to that when calling the basecase functions directly. For balanced multiplications between roughly 20 and 90 words, GMP is using Karatsuba multiplication on top of `mpn_mul_basecase`. Here our implementation uses GMP's Karatsuba code on top of our own basecase multiplication; we observe speedup factors between 1.1 and 1.3 compared to GMP. It is likely that such speedups will extend into the domain of Toom-Cook multiplication ($n, m \gtrsim 90$), but we do not investigate this effect in the present study.

[2]Besides writing the output to `c`, the function returns the most significant word c_{m+n-1} for compatibility with GMP.

TABLE III
TIMINGS FOR MULTIPLICATIONS ON X86-64 (ZEN 3) AND ARM64 (APPLE M1), COMPARING GMP'S `mpn_mul` AGAINST OUR `flint_mpn_mul` IN NUMBER OF CLOCK CYCLES.

Words		x86-64 (Zen 3)			ARM64 (Apple M1)		
m	n	GMP	Ours	Speedup	GMP	Ours	Speedup
1	1	17	11	1.55	21	8	2.62
2	2	20	13	1.54	18	6	3.00
3	3	38	15	2.53	27	9	3.00
4	2	31	14	2.21	20	8	2.50
4	4	50	26	1.92	36	16	2.25
8	8	131	90	1.46	114	67	1.70
12	4	96	81	1.19	70	49	1.43
16	16	454	333	1.36	368	279	1.32
20	20	675	579	1.17	541	393	1.38
31	31	1481	1191	1.24	1138	908	1.25
32	32	1543	1192	1.29	1207	931	1.30
33	33	1635	1393	1.17	1293	1019	1.27
40	40	2179	1896	1.15	1733	1297	1.34
63	63	4939	3830	1.29	3756	2982	1.26
64	64	4876	3824	1.28	3778	3016	1.25
65	65	5123	4234	1.21	3952	3159	1.25
96	96	9362	8207	1.14	7134	5517	1.29

A. Non-constant sizes

Consecutive multiplications of a fixed size (m, n) represents an idealized benchmark. While it remains realistic for certain applications, e.g. modular arithmetic with a fixed modulus or numerical arithmetic with a fixed precision, other applications may generate operands of varying size. An advantage of fully unrolled basecases is the lack of internal branches that risk being mispredicted. On the other hand, the CPU may mispredict the jump into the function table and a too large function table may saturate the L1 instruction cache (I-cache). Roughly speaking, the penalty of either a jump misprediction or an L1 cache miss is on the order of 10 cycles. To analyze the impact, we consider the following synthetic benchmarks:

- **Factorial**: defining $P(a, b) = \prod_{n=a}^{b} n$ recursively via $P(a, b) = P(a, \lfloor \frac{a+b}{2} \rfloor) P(\lfloor \frac{a+b}{2} \rfloor + 1, b)$ whenever $b - a \ge 2$, we compute $b! = P(1, b)$ for 10^6 uniformly randomly chosen $1 \le b \le N$. This results in a skewed distribution of small and large multiplications (both balanced and unbalanced) alternating in a tree pattern.
- **Random**: perform 10^7 multiplications with input lengths $m, n \in \{1, \ldots, N\}$, chosen uniformly randomly. This benchmark is designed to be adversarial for large function tables.

We used Valgrind [18] to simulate the percentage of condition branch mispredictions, indirect jump address mispredictions and L1 I-cache misses. Results are shown in Table IV. The timings in seconds indicate the actual running time on the native CPU without emulation in Valgrind.

On x86-64, we observe that indirect jump address mispredictions and I-cache misses are slightly increased on the Factorial benchmark and significantly increased on the Random benchmark. The speed advantage of our basecase routines remains significant enough to outperform GMP despite these drawbacks, although the speedup over GMP becomes quite small on the Random benchmark.

TABLE IV

MULTIPLICATION PERFORMANCE, CONDITIONAL BRANCH MISPREDICTION RATES "C", INDIRECT JUMP ADDRESS MISPREDICTION RATES "J" AND INSTRUCTION CACHE MISS RATES "I" ON SYNTHETIC BENCHMARKS.

	GMP (`mpn_mul`)				Ours (`flint_mpn_mul`)			
N	Time	C	J	I	Time	C	J	I
Factorial, x86-64 (Zen 3)								
100	0.37 s	4.5%	1.1%	0%	0.25 s	5.6%	5.1%	0.00%
500	2.20 s	5.5%	2.9%	0%	1.41 s	6.4%	10.7%	0.13%
1000	5.11 s	6.1%	3.6%	0%	3.60 s	7.1%	12.6%	0.15%
2000	12.57 s	6.6%	4.1%	0%	9.78 s	7.7%	13.3%	0.11%
Random, x86-64 (Zen 3)								
8	0.32 s	18.3%	22.3%	0%	0.18 s	20.9%	48.4%	0.00%
16	0.55 s	10.0%	18.2%	0%	0.43 s	14.3%	33.1%	3.05%
32	1.39 s	10.5%	12.7%	0%	1.32 s	10.7%	16.7%	0.41%
64	4.48 s	11.5%	11.6%	0%	4.29 s	12.9%	14.3%	0.12%
Factorial, ARM64 (M1)								
100	0.50 s	4.3%	0%	0%	0.25 s	6.6%	5.3 %	0.00%
500	2.51 s	4.8%	0%	0%	1.40 s	6.8%	9.6 %	0.03%
1000	5.65 s	5.1%	0%	0%	3.38 s	6.9%	10.7%	0.08%
2000	13.52 s	5.1%	0%	0%	8.82 s	7.2%	11.1%	0.08%
Random, ARM64 (M1)								
8	0.30 s	11.4%	0.00%	0.00%	0.23 s	11.2%	41.7%	0.01%
16	0.50 s	10.9%	0.00%	0.00%	0.43 s	10.5%	41.6%	0.00%
32	1.31 s	9.6%	0.00%	0.00%	1.13 s	10.0%	13.9%	0.00%
64	4.16 s	8.3%	0.20%	0.02%	3.82 s	9.8%	4.2%	0.06%

On ARM64, the I-cache miss rate remains small thanks to the semi-unrolled basecases which allow for a function table with much smaller code size than on x86-64.

Particularly on x86-64, we speculate that a different table design may improve overall performance in more complex applications where integer multiplications are interleaved with calls to other functions competing for the same I-cache.

IV. SHORT MULTIPLICATION

Let $a = \sum_{i=0}^{n-1} a_i \beta^i$ and $b = \sum_{i=0}^{n-1} b_i \beta^i$ be two n-word integers. We define the high and low product of a and b as $H_n(a,b) = \lfloor ab/\beta^n \rfloor \beta^n$ and $L_n(a,b) = ab - H_n(a,b)$, respectively. We can compute L_n in roughly half the time of the full schoolbook multiplication for ab. Alternatively, we can approximate H_n (or ab itself) with an error around β^n in about the same time as computing L_n. Such short products are interesting for division algorithms, modular arithmetic, and floating-point arithmetic. We focus here on high products.

As illustrated in Fig. 4, we define an approximate high product $H'_n(a,b) = \sum_{i=n-1}^{2n-1} c'_i \beta^i$ by

$$H'_n(a,b) \;=\; \sum_{i+j \ge n-1} a_i b_j \beta^{i+j}, \tag{1}$$

and a refined approximation $H''_n(a,b) = \sum_{i=n-1}^{2n-1} c''_i \beta^i$ by

$$H''_n(a,b) \;=\; H'_n(a,b) + \sum_{i+j=n-2} H_1(a_i,b_j)\beta^{i+j}. \tag{2}$$

Both H'_n and H''_n have $n-1$ low zero words (which an implementation might omit writing) and $n+1$ high words which may be nonzero. The following theorem is proven by counting the omitted terms from the full product, using the facts that $a_i b_j < \beta^2$ and $L_1(a_i,b_j) < \beta$.

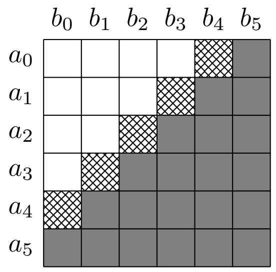

Fig. 4. High multiplication between $a = \sum_{i=0}^{n-1} a_i \beta^i$ and $b = \sum_{j=0}^{n-1} b_j \beta^j$ for $n = 6$. The word products $a_i b_j$ contributing to $H'_n(a,b)$ are shown in gray, and the refinement terms $H_1(a_i,b_j)$ contributing to $H''_n(a,b)$ are shown crosshatched.

Theorem 1. *We have $H'_1 = H''_1 = ab$. If $n, \beta \in \mathbb{Z}_{\ge 2}$, then*

$$ab - \beta^n \;<\; H_n(a,b) \;\le\; ab, \tag{3}$$

$$ab - (n-1)\beta^n \;<\; H'_n(a,b) \;\le\; ab, \tag{4}$$

$$ab - (2n-3)\beta^{n-1} \;<\; H''_n(a,b) \;\le\; ab. \tag{5}$$

The following corollary shows how one can use the high products H' and H'' for fixed-point arithmetic where one simply discards the low n words.

Corollary 2. *Assume that $2n < \beta$ and that we are given integers $a, b \in [0, \beta^n)$ representing n-word fixed-point numbers in $[0,1)$. If we approximate the product using $H_n(a,b)/\beta^n$, $\lfloor H'_n(a,b)/\beta^n \rfloor$ and $\lfloor H''_n(a,b)/\beta^n \rfloor$, then the errors are strictly smaller than 1, n and 2 ULPs, respectively.*

For floating-point arithmetic where radix 2 is more common than radix β, it is useful to consider normalized high products which given normalized significands $a, b \in [\beta^n/2, \beta^n)$ return $2^e h \in I$ with $I = [\beta^{2n}/2, \beta^{2n})$ such that $h \approx ab$. The high products H, H' and H'' all give an h for which either $h \in I$ or $2h \in I$, so we always have $e \in \{0, 1\}$.

Corollary 3. *If $4n < \beta$ and a, b are normalized, then $\lfloor 2^e H''_n(a,b)/\beta^n \rfloor \cdot 2^{nB-e}$ with $e = 1$ if $H''_n(a,b) < \beta^{2n}/2$, $e = 0$ otherwise, gives a normalized nB-bit floating-point approximation of ab with less than 2 ULPs error.*

As the next corollary illustrates, the "control word" $C = c''_{n-1}$ of H''_n allows us to compute n-word products with certified correct rounding, where it is extremely rare for random inputs that the certification fails. For instance, with $\beta = 2^{64}$ and $n = 32$, the failure probability is about 3×10^{-18}.

Corollary 4. *If $C < \beta - (2n-3)$, then $H''_n(a,b) - C\beta^{n-1} = H_n(a,b)$.*

This idea is used in MPFR for floating-point multiplication [20]. However, MPFR uses the high product H' which for n-word precision requires zero-padding n-word inputs to $n+1$ words and evaluating H'_{n+1}. It is potentially better to work with H''_n which avoids such padding.

A. Block decomposition

We can express H'_n or H''_n recursively as a combination of smaller high and full products in various ways; see Fig. 5. To

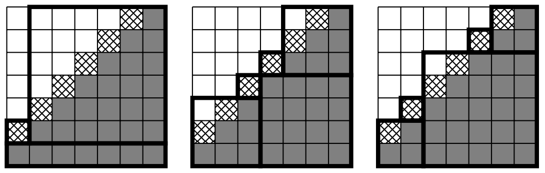
Fig. 5. Examples of block decompositions for high products. From left to right: peeling (6), balanced splitting (7) and Mulders-style splitting (10).

this end, denote $a_{r:s} = \sum_{r \le i < s} a_i \beta^{i-r}$ for a segment of an operand. The following theorem gives an example of a peeling strategy as well as a balanced splitting strategy.

Theorem 5. *For $n \ge 2$,*

$$\begin{aligned} H''_n(a,b) &= H''_{n-1}(a_{0:n-1}, b_{1:n})\beta \\ &\quad + a_{n-1} b \beta^{n-1} + H_1(a_{n-2}, b_0)\beta^{n-2}, \end{aligned} \tag{6}$$

$$\begin{aligned} H''_n(a,b) &= a_{m:n} b_{m:n} \beta^{2m} \\ &\quad + H''_m(a_{k:n}, b_{0:m})\beta^k + H''_m(a_{0:m}, b_{k:n})\beta^k \\ &\quad + \left(\begin{cases} H_1(a_m, b_m), & n \text{ even} \\ H_1(a_k, b_m) + H_1(a_m, b_k), & n \text{ odd} \end{cases} \right) \beta^{n-2}, \end{aligned} \tag{7}$$

where $k = \lceil n/2 \rceil - 1$ and $m = \lfloor n/2 \rfloor - 1$. The analogous formulas with H' instead of H'' hold if the β^{n-2} terms are omitted.

For peeling-based squaring, a symmetric version of (6) which recurses to H''_{n-2} is preferable, since this allows reducing to a smaller squaring rather than a general multiplication.

For Karatsuba-sized n, rather than computing H'_n or H''_n precisely, it is more efficient to use Mulders's strategy [7], [9], [16] to compute some approximation

$$H'_n(a,b) \le \tilde{H}'_n(a,b) \le ab \tag{8}$$

or

$$H''_n(a,b) \le \tilde{H}''_n(a,b) \le ab \tag{9}$$

by allowing a full (k,k) product to extend above the diagonal in Fig. 4.

Theorem 6. *Let $k := k(n)$ be an arbitrary parameter chosen from the range $\lfloor n/2 \rfloor < k \le n$, and let $m = n - k$. If $k = n$, define $\tilde{H}''_n(a,b) = ab$. Otherwise, define $\tilde{H}''_n$ recursively by*

$$\begin{aligned} \tilde{H}''_n(a,b) &= a_{m:n} b_{m:n} \beta^{2m} \\ &\quad + \tilde{H}''_m(a_{k:n}, b_{0:m})\beta^k + \tilde{H}''_m(a_{0:m}, b_{k:n})\beta^k \\ &\quad + \big(H_1(a_{k-1}, b_{m-1}) + H_1(a_{m-1}, b_{k-1})\big)\beta^{n-2}, \end{aligned} \tag{10}$$

Then $\tilde{H}''_n$ satisfies (9). *The analogous statement for $\tilde{H}'$ holds if one omits the β^{n-2} term in* (10).

The optimal value of k as a function of n is discussed in [7]. In practice, one precomputes a table of the best measured k up to the FFT range where $k = n$ becomes optimal.

We note that error bounds for H' and H'' as approximations of ab (e.g. Theorem 1, corollaries 2, 3 and 4) are valid also for the Mulders variants $\tilde{H}'$ and $\tilde{H}''$ which are never less accurate.

B. Implementation results

We have implemented fully unrolled basecases for H''_n based on the algorithm in Fig. 6 for $n \le 9$ (x86-64) and $n \le 8$ (ARM64). In the following, we only discuss the results for x86-64. For $n \ge 10$, we have considered both a generic loop-based routine in the style of mpn_mul_basecase and block decomposition variants. On a Zen 3 machine, we found optimal results using the $n = 9$ basecase together with peeling for $10 \le n \le 13$, balanced block decomposition for $14 \le n < 40$, and the Mulders decomposition for $\tilde{H}''_n$ for $n \ge 40$. The multiplication routine discussed in section III is used internally for full products.

We have added several new functions to FLINT. The main function flint_mpn_mulhigh_n computes H''_n (or $\tilde{H}''_n$ for large n), writing the n words $c''_n, \ldots, c''_{2n-1}$ to the output array and returning the word c''_{n-1}. The new nfloat type holds floating-point numbers with nB-bit precision; it uses the multiplication of Corollary 3 based on a normalized variant of flint_mpn_mulhigh_n with separate basecases for $n \le 9$.

Timings are shown in Table V. Comparing to Table I, we observe that our high products are always faster than or equal to full products, with a speedup of up to around a factor 1.8 ($n = 9$).

For reference, MPFR's mpfr_mul computes correctly rounded floating-point products, switching between special-case code for precision up to 128 bits [15], mpn_mul, and the internal function mpfr_mulhigh_n which computes H'_n or $\tilde{H}'_n$. Thanks to our basecase optimizations, flint_mpn_mulhigh_n is roughly 3× faster than mpfr_mulhigh_n for small to medium n despite effectively giving an extra word of accuracy, and nfloat_mul is faster than mpfr_mul (here rounding to zero) by similar factors.

```
function mulhigh_n
    Input: $c$: pointer to at least $n$ allocated words,
           $a = \sum_{i=0}^{n-1} a_i \beta^i$, $b = \sum_{i=0}^{n-1} b_i \beta^i$,
           $n \ge 2$: compile-time constant,
    Output: $c \leftarrow \lfloor H''_n(a,b)/\beta^n \rfloor$ and
            returns $c''_{n-1}(a,b)$
    $\langle r_0, r_1 \rangle \leftarrow a_{n-1} \cdot b_0 + \lfloor a_{n-2} \cdot b_0 / \beta \rfloor$
    for $i \leftarrow 1$ to $n-1$ do
        $\langle r_0, \ldots, r_{i+1} \rangle \leftarrow \langle r_0, \ldots, r_i, 0 \rangle +$
            $\langle a_{n-i-1}, \ldots, a_{n-1} \rangle \cdot b_i + \lfloor a_{n-i-2} \cdot b_i / \beta \rfloor$
    end
    $(c_0, \ldots, c_{n-1}) \leftarrow (r_1, \ldots, r_n)$
    return $r_0$
end
```

Fig. 6. Basecase algorithm for precise approximate high product.

We also time double-double and quad-double multiplication (c_dd_mul and c_qd_mul) as implemented in the QD library [10], giving roughly $2 \times 53 = 106$ and $4 \times 53 = 212$ bits of precision respectively. The quad-double multiplication uses a high product similar to H''_4, requiring 13 total subproducts of

TABLE V
TIMINGS FOR n-WORD OR nB-BIT HIGH PRODUCT ON X86-64 (ZEN 3).

n, nB		Integer			Floating-point		
$B = 64$		`mpfr_mulhigh_n`	`flint_mpn_mulhigh_n`		`mpfr_mul`	`nfloat_mul`	
		Cycles	Cycles	Speedup	Cycles	Cycles	Speedup
1	64	14	7	2.00	41	14	2.93
2	128	25	8	3.13	70	16	4.38
3	192	39	12	3.25	102	22	4.64
4	256	55	18	3.06	114	32	3.56
5	320	72	25	2.88	137	43	3.19
6	384	90	34	2.65	151	53	2.85
7	448	101	44	2.30	171	63	2.71
8	512	118	54	2.19	197	70	2.81
9	576	135	65	2.08	229	85	2.69
10	640	152	87	1.74	272	117	2.32
16	1024	285	240	1.19	494	278	1.78
20	1280	426	393	1.08	636	435	1.46
32	2048	956	968	0.99	1214	1024	1.19
40	2560	1449	1456	1.00	1853	1519	1.22
64	4096	3613	3270	1.10	3971	3305	1.20
$B = 53$					`c_dd_mul` `c_qd_mul`		
2	106				16		
4	212				154		

which 10 products are computed in 106-bit precision (requring a multiplication and a fused multiply-add) while 3 diagonal corrections use ordinary 53-bit products for the high parts [11]. Our 128-bit `nfloat` multiplication is roughly as fast as `c_dd_mul` and our 256-bit `nfloat` multiplication is 5 times faster than `c_qd_mul`. An important caveat is that we have not attempted to SIMD-vectorize the double-double and quad-double operations in this benchmark, which should be more effective than for the other types.

C. Application to dot products

Many operations (e.g. matrix multiplication) are conveniently expressed in terms of dot products $\sum_{i=1}^{N} x_i y_i$.

For multiple precision floating-point dot products, we consider the algorithm of Johansson [12]. The original implementation (available in FLINT as the function `arf_approx_dot`) allows operands with mixed precision and uses a mixture of `mpn_mul` and `mpfr_mulhigh_n` for the multiplications. Our version shown in Fig. 7 uses high multiplications exclusively and has been streamlined by assuming that the word precision n is the same for all input operands; however, like the original, the precision of each multiplication varies optimally with the magnitude of the term.

Theorem 7. *The final error in Algorithm 7 is at most* $N(n+2)$ *ULPs in* s*, i.e. the absolute error is at most* $N(n+2)\mu$ *where* $\mu = 2^e \beta^{-n-1}$.

Proof. We consider the three branches to compute the term t.

In the first branch, we compute an $(n+1)$-word approximation of $a_i b_i$ and then perform a truncating right shift by $\delta \geq 1$ bits. Using Theorem 1, we can show that the error in $t\mu$ is at most $n\mu$. Indeed, if $n = 1$, the product is exact and we get at most μ error from the shift. If $n \geq 2$, the

```
function n-float_dot
    Input: precision $n \geq 1$; nonzero $n$-word binary
           floating-point numbers $x_1, \ldots, x_N$,
           $y_1, \ldots, y_N$.
    Output: $(n+1)$-word approximation of $\sum_{i=1}^{N} x_i y_i$
    Write $x_i = (-1)^{v_i} 2^{e_i} a_i \beta^{-n}$, $\beta^n/2 \leq a_i \leq \beta^n - 1$
    Write $y_i = (-1)^{w_i} 2^{f_i} b_i \beta^{-n}$, $\beta^n/2 \leq b_i \leq \beta^n - 1$
    // Bounding exponent (plus sign bit)
    $e \leftarrow \max_{1 \leq i \leq N}(e_i + f_i) + \lceil \log_2(N+1) \rceil + 1$
    // $(n+1)$-word accumulator
    $s \leftarrow 0$
    for $i \leftarrow 1$ to $N$ do
        $\delta \leftarrow e - (e_i + f_i)$
        $d \leftarrow \lfloor \delta / B \rfloor$
        if $d = 0$ then // $1 \leq \delta < B$
            // $t$ has $n+1$ words
            $t \leftarrow \lfloor H''_n(a_i, b_i) / (\beta^{n-1} 2^{\delta}) \rfloor$
        else if $1 \leq d \leq n$ then
            // $a', b', t$ have $(n+1) - d$ words
            $(a', b') \leftarrow (\lfloor a_i / \beta^{d-1} \rfloor, \lfloor b_i / \beta^{d-1} \rfloor)$
            $t \leftarrow$
                $\lfloor H'_{(n+1)-d}(a', b') / (\beta^{(n+1)-d} 2^{\delta \bmod B}) \rfloor$
        else
            $t \leftarrow 0$
        end
        // Two's complement addition
        $s \leftarrow s + (-1)^{v_i + w_i} t \mod \beta^{n+1}$
    end
    // Interpret $s$ as signed,
        $s \in (-\beta^{n+1}/2, \beta^{n+1}/2)$
    return $s 2^e \beta^{-n-1}$
end
```

Fig. 7. Algorithm for dot product of floating-point numbers with n-word precision.

approximate product introduces an error of at most $(2n-3)\mu$ prior the shift by $\delta \geq 1$; with the shift, the error is at most $(2n-3)\mu/2 + \mu \leq n\mu$.

In the second branch, truncating a_i and b_i contributes at most 2μ to the total error. By Theorem 1, the subsequent product H' contributes error at most $(n-1)\mu$, and the truncation in the right shift contributes at most μ. Thus the error is at most $(2 + (n-1) + 1)\mu = (n+2)\mu$.

In the third branch, the omitted product is smaller than μ.

The error bound is largest in the second branch; multiplying by the number of terms N gives the result.

□

In the end, we will typically want to round the final result to a normalized floating-point number with bit precision $p \leq nB$. For random input vectors with no significant cancellation, the algorithm in Fig. 7 gives a result with nearly full accuracy as long as $N^2 n \ll \beta$. Indeed, if $|\sum_i x_i y_i| \gtrsim \max_{1 \leq i \leq N} |x_i y_i|$, so that $|s| \gtrsim \beta^{n+1}/N$, Theorem 7 shows that the *relative* error in s is $\lesssim N^2 n \beta^{-n-1}$. and we can hence determine the correct rounding of the dot product with failure probability $\sim (N^2 n)/\beta$. For example, this is of order 10^{-9} if $\beta = 2^{64}$, $N = 10000$, $n = 1000$.

Theorem 7 remains valid if we substitute more accurate implementations of the high products, as long as we always compute lower bounds for the full product (this assumption is necessary to guarantee that we do not overflow the accumulator). In particular, we can use H'' instead of H' in the second branch, and we can replace H'' by a Mulders product $\tilde{H}''$.

D. *Implementation results for matrix multiplication*

Table VI shows timings (real time on a Zen 3 CPU, with frequency boost enabled) for single-threaded floating-point matrix multiplication done using N^2 dot products, i.e. without using any other fast matrix multiplication techniques. We compare the following implementations of dot products:

- QD: a simple add-multiply loop using the C++ interface of the QD library.
- MPFR: a loop calling `mpfr_mul` and `mpfr_add`.
- ARF (GMP): the original dot product code of [12], `arf_approx_dot`, using `mpn_mul` for full multiplications.
- ARF (New): `arf_approx_dot` when using our full multiplication code (`flint_mpn_mul`) instead of `mpn_mul`.
- NFLOAT: the function `_nfloat_vec_dot` implementing the algorithm in Fig. 7, using our high multiplication code (`flint_mpn_mulhigh_n`).

We observe that simply replacing the GMP multiplication in `arf_approx_dot` with our faster basecases gives a nontrivial ($\approx 1.5\times$) speedup at a few words of precision. The streamlined algorithm for NFLOAT benefits from our basecase high multiplication and achieves a $\approx 2\times$ speedup. We leave for future work incorporating basecase high multiplications in the more general algorithm of [12].

At high precision, both the ARF and NFLOAT algorithms perform much better with exponentially scaled entries than with uniform entries, since many terms can be computed with reduced precision. This highlights that it is useful to have an efficient high multiplication not only for the target number of words n, but for every size $n' \leq n$. In low precision, the performance is somewhat worse with non-uniform entries due to increased branching.

TABLE VI
TIME (MILLISECONDS) ON X86-64 (ZEN 3) TO MULTIPLY TWO 100×100 FLOATING-POINT MATRICES USING REPEATED DOT PRODUCTS.

nB	QD	MPFR	ARF (GMP)	ARF (New)	NFLOAT
		Uniformly random entries in $[-1, 1]$			
106	5.3				
212	38				
64		29	5.9	5.9	4.3
128		44	7.0	7.0	5.4
192		57	23	19	10
256		64	28	22	12
384		72	40	29	21
512		89	50	39	26
1024		159	137	111	73
2048		343	302	303	250
4096		1059	1001	1007	820
		Entries scaled by random exponents 2^{-k}, $0 \leq k < nB$			
64		29	8.0	8.0	6.8
128		44	9.3	9.3	8.5
192		56	23	20	17
256		62	28	23	17
384		72	32	26	19
512		90	38	31	21
1024		161	57	47	31
2048		346	106	89	64
4096		1092	232	219	169

REFERENCES

[1] AWS Labs. s2n-bignum. https://github.com/awslabs/s2n-bignum/, 2024.

[2] Laurent-Stéphane Didier, Nadia Mrabet, Léa Glandus, and Jean-Marc Robert. Truncated multiplication and batch software SIMD AVX512 implementation for faster Montgomery multiplications and modular exponentiation. *IACR Communications in Cryptology*, October 2024.

[3] Takuya Edamatsu and Daisuke Takahashi. Accelerating large integer multiplication using Intel AVX-512IFMA. In *International Conference on Algorithms and Architectures for Parallel Processing*, 2019.

[4] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, page 1202–1219. IEEE, May 2019.

[5] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13–es, 2007.

[6] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.3.0 edition, 2023. http://gmplib.org/.

[7] Guillaume Hanrot and Paul Zimmermann. A long note on Mulders' short product. *Journal of Symbolic Computation*, 37(3):391–401, March 2004.

[8] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics*, 193(2):563, 2021.

[9] David Harvey and Paul Zimmermann. Short division of long integers. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 7–14. IEEE, 2011.

[10] Y. Hida, X. S. Li, and D. H. Bailey. Library for double-double and quad-double arithmetic. *NERSC Division, Lawrence Berkeley National Laboratory*, 2007. http://crd-legacy.lbl.gov/~dhbailey/mpdist/.

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

[12] Fredrik Johansson. Faster arbitrary-precision dot product and matrix multiplication. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, June 2019.

[13] Dougall Johnson. applecpu. https://github.com/dougallj/applecpu, 2023.

[14] Donald Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1998.

[15] Vincent Lefevre and Paul Zimmermann. Optimized Binary64 and Binary128 arithmetic with GNU MPFR. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, page 18–26. IEEE, July 2017.

[16] Thom Mulders. On short multiplications and divisions. *Applicable Algebra in Engineering, Communication and Computing*, 11:69–88, 2000.

[17] Niels Möller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, 2011.

[18] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[19] The FLINT team. *FLINT: Fast Library for Number Theory*, 2024. https://flintlib.org.

[20] The MPFR team. The MPFR library: algorithms and proofs. https://www.mpfr.org/algorithms.pdf, 2024. [Accessed 20-12-2024].

[21] Joris van der Hoeven and Grégoire Lecerf. Implementing number theoretic transforms. working paper or preprint, December 2024.