Modular SIMD arithmetic in MATHEMAGIX

JORIS VAN DER HOEVEN and GRÉGOIRE LECERF, CNRS & École polytechnique, Laboratoire LIX
GUILLAUME QUINTIN, Université de Limoges, Laboratoire XLIM

Modular integer arithmetic occurs in many algorithms for computer algebra, cryptography, and error correcting codes. Although recent microprocessors typically offer a wide range of highly optimized arithmetic functions, modular integer operations still require dedicated implementations. In this article, we survey existing algorithms for modular integer arithmetic and present detailed vectorized counterparts. We also describe several applications, such as fast modular Fourier transforms and multiplication of integer polynomials and matrices. The vectorized algorithms have been implemented in C++ inside the free computer algebra and analysis system MATHEMAGIX. The performance of our implementation is illustrated by various benchmarks.

CCS Concepts: • **Mathematics of computing → Mathematical software performance**; *Computations on matrices*; *Computations on polynomials*; • **Theory of computation → Vector/streaming algorithms;**

Additional Key Words and Phrases: Modular integer arithmetic, fast Fourier transform, integer product, polynomial product, matrix product, Mathemagix

ACM Reference Format:
Joris van der Hoeven, Grégoire Lecerf, and Guillaume Quintin. 2016. Modular SIMD arithmetic in MATHEMAGIX. ACM Trans. Math. Softw. 43, 1, Article 5 (August 2016), 37 pages.
DOI: http://dx.doi.org/10.1145/2876503

1. INTRODUCTION

During the past decade, major manufacturers of microprocessors have changed their focus from ever-increasing clock speeds to putting as many cores as possible on one chip and to lower power consumption. One approach followed by leading constructors such as INTEL and AMD is to put as many x86 compatible processors on one chip. Another approach is to rely on new simplified processing units, which allows us to increase the number of cores on each chip. Modern *Graphics Processing Units* (GPUs) have been designed according to this strategy.

As powerful as multicore architectures may be, this technology also comes with its drawbacks. Besides the increased development cost of parallel algorithms, the main disadvantage is that the high degree of concurrency allowed by multicore architecture often constitutes an overkill. Indeed, many computationally intensive tasks ultimately boil down to classical mathematical building blocks such as matrix multiplication or *fast Fourier transforms* (FFTs).

Authors' addresses: J. van der Hoeven and G. Lecerf, Laboratoire d'informatique de l'École polytechnique, LIX, UMR 7161 CNRS, Campus de l'École polytechnique, 1 rue Honoré d'Estienne d'Orves, Bâtiment Alan Turing, CS35003, 91120 Palaiseau, France; emails: {vdhoeven, lecerf}@lix.polytechnique.fr; G. Quintin, Laboratoire XLIM, UMR 7252 CNRS, Université de Limoges, 123, avenue Albert Thomas, 87060 Limoges Cedex, France; email: guillaume.quintin@unilim.fr.

ACM 0098-3500/2016/08-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2876503

In many cases, the implementation of such building blocks is better served by simpler parallel architectures, and more particularly by the *Single Instruction, Multiple Data* (SIMD) paradigm. Limited support for this paradigm has been integrated into the x86 processor families with the INTEL MMX and *Streaming SIMD Extensions* (SSE) technologies. More recently, the advent of INTEL *Advanced Vector eXtensions* (AVX) and AVX-512 has further empowered this model of computation. Historically speaking, vector computers have played an important role in "High-Performance Computing" (HPC). An interesting question is whether more powerful SIMD instructions can also be beneficial to other areas.

More specifically, this article concerns the efficiency of the SIMD approach for doing exact computations with basic mathematical objects such as integers, rational numbers, modular integers, polynomials, matrices, and so on. Besides high performance, we are therefore interested in the reliability of the output. For this reason, we like to call this area *High Quality Computing* (HQC).

One important mathematical building block for HQC is fast modular arithmetic. For instance, products of multiple-precision integers can be computed efficiently using FFTs over finite fields of the form $\mathbb{F}_p$ with $p = k2^l + 1$, where p typically fits within 32 or 64 bits and l is as large as possible under this constraint [Pollard 1971]. Similarly, integer matrices can be multiplied efficiently using such FFTs or Chinese remaindering.

The aim of this article is twofold. First, we adapt known algorithms for modular integer arithmetic to the SIMD model. We carefully compare various possible implementations as a function of the bit-size of the modulus and the available SIMD instructions. In the second part of the article, we apply our implementations to several fundamental algorithms for exact arithmetic, namely to the multiplication of integers, integer matrices, modular polynomials and modular polynomial matrices. We show that our implementations based on modular arithmetic outperform existing implementations in many cases, sometimes by one order of magnitude.

The descriptions of the algorithms are self-contained, and we provide implementation details for the SSE and AVX instruction sets. All implementations were done in C++ and our software is freely distributed within the computer algebra and analysis system MATHEMAGIX (http://www.mathemagix.org) [van der Hoeven and Lecerf 2013a; van der Hoeven et al. 2002; van der Hoeven and Lecerf 2013b; Lecerf 2010; van der Hoeven et al. 2011], from revision $\geqslant$9985 available at http://gforge.inria.fr/projects/mmx).

1.1. Related Work

There is a large literature on modular integer arithmetic. The main operation to be optimized is multiplication modulo a fixed integer $p > 1$. This requires an efficient reduction of the product modulo p. The naive way of doing this reduction would be to compute the remainder of the division of the product by p. However, divisions tend to be more expensive than multiplications in current hardware. The classical solution to this problem is to pre-compute the inverse p^{-1} so all divisions by p can be transformed into multiplications.

In order to compute the product c of a and b modulo p, it suffices to compute the integer floor part q of the rational number abp^{-1} and to deduce $c = ab - qp$. The most obvious algorithm is to compute q using floating point arithmetic. However, this approach suffers from a few drawbacks. First, in order to ensure portability, the processor should comply to the IEEE-754 standard, which ensures that rounding errors are handled in a predictable way. This is fortunately the case for most recent processors. Second, the size of the mantissa can be twice as large as the size of the modulus. Back-and-forth conversions from and to integers may then be required, which are expensive on some

platforms. For more details, we refer the reader to Fog [2012b, Chapter 14], Alverson [1991], and Baker [1992]. Many recent processors contain *fused multiply-add* instructions (FMA), which facilitate taking full advantage of floating point operations; these instructions have already been used before for the "TwoProduct" algorithm [Ogita et al. 2005] and modular arithmetic [Thomé 2012, Chapitre 14].

Barrett's algorithm [Barrett 1987] provides an alternative to the floating point approach. The idea is to rescale the floating point inverse of p^{-1} and to truncate it into an integer type. An important variant has been designed by Möller and Granlund [2011], who also propose an optimized algorithm for computing the pre-inverse in any size by means of integer operations only. For some early comparisons between integer and floating point operations, and a branch-free variant, we refer to the article of Granlund and Montgomery [1994]. This approach is also discussed by Fog [2012c, Section 16.9] for particular processors. For multiple-precision integers, algorithms were also given by Hasenplaugh et al. [2007], Bardis et al. [2010], and Möller and Granlund [2011].

Another alternative approach is to precompute the inverse p^{-1} of the modulus p as a 2-adic integer. This technique, which is essentially equivalent to *Montgomery's algorithm* [Montgomery 1985], only uses integer operations but requires p to be odd. Furthermore, modular integers need to be encoded and decoded (with a cost similar to one modular product), which is not always convenient. Implementations have been discussed by Kaya Koç et al. [1996]. An SSE version for 32-bit integers can be found in an article of Meng et al. [2010]. A generalization to even moduli was proposed by Kaya Koç [1994]: It relies on computing separately modulo 2^k and $\tilde{p}$ (such that $\tilde{p}$ is odd and $p = 2^k\tilde{p}$) via Chinese remaindering. Unfortunately, this leads to a significant overhead for small moduli. Comparisons between Barrett's and Montgomery's product were given by Bosselaers et al. [1994] for values of k corresponding to a few machine words. A recent survey can be found in the article of Nedjah and de Macedo Mourelle [2006] with a view towards hardware implementation.

Modular arithmetic and several basic mathematical operations on polynomials and matrices have been implemented before on several types of parallel hardware (multicore architectures and GPUs); see, for instance, Bernstein et al. [2009a, 2009b], Giorgi et al. [2010], and Haque and Maza [2012]. Although the present article focuses on x86 compatible processor families and their recent extensions, the algorithms we describe are intended to be useful for all platforms supporting SIMD technology.

Our applications to matrix product over large polynomials or integers are central tasks to many algorithms in computer algebra [Aho et al. 1974; Knuth 1997; von zur Gathen and Gerhard 2003; Bini and Pan 2012]. For a sample of recent related implementations, the reader might consult works of Moreno Maza and Xie [2010], Harvey and Sutherland [2014], and Giorgi and Lebreton [2014].

1.2. Our Contributions

One difficulty with modular arithmetic is that the most efficient algorithms are strongly dependent on the bit-size of the modulus p, as well as the availability and efficiency of specific instructions in the hardware. This is especially true in the case of SIMD processors, for which the precise instruction sets still tend to evolve rather quickly. In this article, we have attempted to be as exhaustive as possible: We implemented all classical strategies, while hand optimizing all algorithms as a function of the available SIMD instructions.

The fast sequential algorithms for modular reduction from the previous section all involve some branching in order to counterbalance the effect of rounding. Our first contribution is to eliminate all branching to make vectorization possible and efficient. Our second contribution is a complete implementation of the various approaches as a

function of the available SIMD instructions. For Barrett's and Montgomery's algorithms, we consider both the SSE 4.2 and AVX 2 instruction sets, for all types of supported integers. For the floating point approach, we have implementations with and without SSE 4.1, AVX, and FMA extensions. We finally provide detailed benchmarks for the performance of the different approaches in practice. The observed speedups with respect to the *scalar versions* (i.e., without vectorization) nearly reflect theoretical expectations.

High-performance libraries such as GMP [Granlund et al. 1991], MPIR [Hart and the MPIR Team 2010], MPFR [Fousse et al. 2007], and FLINT [Hart and the FLINT Team 2008] for computing with large integers, floating point numbers in arbitrary precision, polynomials, and matrices are written in C and mostly aim at providing users with very fast mathematical operations. In fact, the internal representations and algorithms operating on them are not expected to be accessed nor replaced *a posteriori*. Instead, the C++ libraries of MATHEMAGIX provide users with high-level interfaces to data structures and mathematical objects but also with interfaces to internal representations and algorithms. Users can assemble algorithms for specific purposes, fine-tune thresholds, and easily replace an algorithm by another one *a posteriori*. In Section 4, we explain the main design principles. In particular, we show how our modular numbers are implemented and how our approach makes it simple to develop SIMD variants of algorithms that benefit from hardware vectorization features.

A third contribution of this article is the application of our modular arithmetic to an SIMD version of the fast Fourier transform algorithm in a prime field of the form $\mathbb{Z}/p\mathbb{Z}$, where $p - 1$ is a multiple of a large power of two. Our implementations outperform existing software and we report on timings and comparisons for integer, polynomial and matrix products.

Besides low-level software design considerations, our main research goals concern algorithms for solving polynomial systems, for effective complex analysis, and error correcting codes. MATHEMAGIX includes several of our recent algorithms [Bank et al. 2015; Berthomieu et al. 2011, 2013; van der Hoeven and Lecerf 2013c]. From our experience, a strict bottom-up approach to mathematical software design prevents users from implementing high-level algorithms efficiently. In other words, assembling algorithms from a strict mathematical point of view does not always lead to the best performance. Interfaces to algorithms and the ability to reassemble them for higher level operations with a sufficient level of genericity turns out to be very useful. For instance, multiplying integer matrices by performing integer products and additions in sequence according to the schoolbook algorithm can be improved if one has access to relatively generic implementations of the low-level integer product sub-functions. This specific example is adressed in our last section.

1.3. Conventions

Throughout this article, timings are measured on a platform equipped with an INTELCORE i7-4770 CPU @ 3.40GHz and 8GB of 1600MHz DDR3. It runs the JESSIE GNU DEBIAN operating system with a LINUX kernel version 3.14 in 64-bit mode. Care has been taken for avoiding CPU (*Central Processing Unit*) throttling issues while measuring timings. Nevertheless, timings often vary in a nonnegligible range sometimes over 20%, and we thus measure average timings. We compile mainly with GCC [GCC 1987] version 4.9.2. For pseudocode, we use the operators of the C99 language with their exact meanings and the usual integer types from `inttypes.h`. Note also that integer promotions do not invalidate the algorithms and proofs in this article. We refer the reader to the C99 (ISO/IEC 9899:1999) standard Sections 6.2.5 and 6.3.1.1 for further details [British Standards Institution 2003]. For simplicity, we assume that the operator $>>$ always implements the *right arithmetic shift*.

1.4. Brief Survey of SIMD Technology

For completeness, we conclude this introduction with recalling basic facts on SIMD technology. This technology can be seen as a type of parallelism where multiple processing units simultaneously execute the same instruction on multiple data. We can think of an SIMD processor as a single CPU that operates on registers that are really vectors of scalar data. The main basic characteristics of SIMD processors are the following:

—The scalar data types that are supported;
—The total bit-size b of vector registers;
—The number v of vector registers;
—For each scalar data type T, the instruction set for vector operations over T;
—The instruction set for other operations on SIMD registers, such as communication with the main memory or permutations of entries.

Modern SIMD processors typically support both floating point types (`float` and `double`) and integer types (8-, 16-, 32-, and 64-bit integers). For a scalar type T of bit-size n, we operate on vectors of b/n coefficients in T.

Currently, the most common SIMD technologies are SSE (with $b = 128$ and $v = 16$), AVX (with $b = 256$ and $v = 16$), and AVX-512 (with $b = 512$ and $v = 32$). For instance, an AVX-enabled processor can execute an operation on a vector of 16 integers of bit-size 16 in unit time. It should be noted that these bit-sizes $b \leqslant 512$ are rather modest in comparison to their historical ancestors, such as the CDC STAR-100 and CRAY vector computers (CDC was a trademark of CONTROL DATA CORPORATION). One advantage of shorter vectors is that it remains feasible to provide instructions for permuting the entries of vectors in quite general ways (i.e., "communications" remain relatively inexpensive, whenever needed).

In this article, our SIMD algorithms are described using *compiler intrinsics*. Most of the time, the semantics of the SIMD types and instructions in this article are quite straightforward. Let us give a few examples for SSE-enabled processors:

—The types `__m128i` and `__m128d` correspond to packed 128-bit integer and floating point vectors.
—The instruction `_mm_add_epi64` corresponds to the addition of two vectors of 64-bit signed integers of type `__m128i` (so the vectors are of length 2).
—The predicate `_mm_cmpgt_epi64` corresponds to a component-wise $>$ test on two vectors of 64-bit signed integers of type `__m128i`. The Boolean results `true` and `false` are encoded by the signed integers `-1` and `0`.
—The instruction `_mm_mul_pd` corresponds to the multiplication of two vectors of double precision floating point numbers of type `__m128d` (so the vectors are of length 2).

When needed, typically for comparisons, the `_epi` suffix is replaced by `_epu` to indicate that packed integers are unsigned. We notice that all floating point arithmetic conforms to the IEEE-754 standard. In particular, results of floating point operations are obtained through correct rounding of their exact mathematical counterparts, where the global rounding mode can be set by the user. Some processors also provide fused multiply add (FMA) and subtract instructions `_mm_fmadd_pd` and `_mm_fmsub_pd`, which are useful in Section 3. Another less obvious but useful instruction that is provided by some processors is:

—`_mm_blendv_pd` $(\vec{x}, \vec{y}, \vec{m})$ returns a vector $\vec{z}$ with $z_i = y_i$ whenever the most significant bit of m_i is set and $z_i = x_i$ otherwise (for each i). For floating point numbers, it should be noticed that the most significant bit of m_i corresponds to the sign bit.

The efficiency of SIMD instructions is usually measured in terms of latency and throughput: The *latency* is the number of CPU clock cycles needed to make the result of an instruction available to an other instruction; The *reciprocal throughput*, sometimes called *issue latency*, is the (fractional) number of cycles needed to actually perform the computation—for brevity, we drop "reciprocal." In this article, we shall not investigate the question of finding the best possible assembly code for each operation, which mostly reduces to finding the best reordering for a given processor in simple cases. Instead, we delegate this optimization to the compiler.

For the sake of illustration, we provide unfamiliar readers with typical examples for our aforementioned processor of family number 6 and model number 60 (3C in hexadecimal):

—For vectors of 8 packed 32-bit integers (signed or not), with the AVX 2 instruction set: Additions and subtractions have latency 1 and throughput 0.5; multiplications have latency 5 and throughput 1; computing the entrywise maximum of two vectors has latency 1 and throughput 0.5.
—For vectors of 4 packed `double`, with the AVX instruction set: Additions and subtractions have latency 3 and throughput 1; multiplications have latency 5 and throughput 1; FMA have latency 5 and throughput 0.5; `_mm256_blendv_pd` has latency 2 and throughput 1 (this intrinsic is the 256-bit version of `_mm_blendv_pd`).

For more details about SSE and AVX intrinsics, we refer to the INTEL intrinsic guides [Intel Corporation 2013a, 2013b] and also to Fog [2012a] for useful comments and practical recommendations.

Let us mention that a standard way to benefit from SIMD technology is to rely on *auto-vectorization* features of compilers. Languages such as C and C++ have not yet been extended to support SIMD technology in a standard and more explicit manner. In particular, current SIMD types are not even *bona fide* C/C++ types. Programming via intrinsics contains a certain number of technical pitfalls, such as memory alignment and latency issues. It is therefore left to the programmer to wrap memory allocation functions or to rely on specific features of the compiler.

2. MODULAR OPERATIONS VIA INTEGER TYPES

If a and b are nonnegative integers, then $a \operatorname{quo} b$ and $a \operatorname{rem} b$ represent the *quotient* and the *remainder* in the long division of a by b, respectively. We let `U` denote an unsigned integer type of bit-size n assumed to be at least 2 for convenience. This means that all the integers in the range from 0 to $2^n - 1$ can be represented in `U` by their binary expansion. The corresponding type for signed integers of bit-size n is written `I`: all integers from -2^{n-1} to $2^{n-1} - 1$ can be represented in `I`. The type `L` represents unsigned integers of size at least $2n$. Let p be a nonnegative integer of bit-size at most m with $m \leqslant n$. For efficiency reasons, we consider that n and m are quantities known at compilation time, whereas the actual value of p is only known at execution time. In fact, deciding which elementary modular arithmetic algorithm to use at runtime according to the bit-size of p is, of course, too expensive. For convenience, we identify the case $p = 0$ to computing modulo 2^n. Modulo p integers are stored as their representative in $\{0, \ldots, p - 1\}$.

2.1. Modular Sum

Although modular sum seems rather easy at first look, we quickly summarize the known techniques and pitfalls.

2.1.1. Unvectorized Implementations. Given x and y modulo p, in order to compute $(x + y) \operatorname{rem} p$, we can first compute $x + y$ in `U` and subtract p when an overflow occurs or when $x + y \geqslant p$, as detailed in the following function:

FUNCTION 2.1.
Input: Integers x and y modulo p.
Output: $(x+y)$ rem p.

```
U add_mod (U x, U y) {
(1) U a = x + y;
(2) if (a < x) return a - p;
(3) return (a >= p) ? a - p : a; }
```

Of course, when $m \leqslant n-1$, no overflow occurs in the sum of line 1, and line 2 is useless. If branching is more expensive than shifting and if $m \leqslant n-1$, then one can compute `I a` $= x+y-p$ and return a `+ ((a >> (n-1)) & p)`, where we recall that `>>` implements the right arithmetic shift. It is necessary to program both versions and determine which approach is the fastest for a given processor at configuration time. Negation and subtraction can be easily implemented in a similar manner.

2.1.2. Implementations with SSE 4.2 and AVX 2. Arithmetic operations on packed integers are rather well supported by SSE 4.2, uniformly for various types of integers. Let $\vec{p}$ represent the packed n-bit integer of type `__m128i`, whose entries are filled with p. In order to avoid branching in Function 2.1, one can compute `U` $a = x+y$ and return `min` $(a,\ a-p)$ when $m \leqslant n-1$. This approach can be straightforwardly vectorized for packed integers of bit-sizes 8, 16, and 32, as exemplified in the following function:

FUNCTION 2.2.
Input: Packed 32-bit integers $\vec{x}$ and $\vec{y}$ modulo $\vec{p}$, assuming $m \leqslant n-1$.
Output: $(\vec{x}+\vec{y})$ rem $\vec{p}$.

```
__m128i add_mod_31_epu32 (__m128i x, __m128i y) {
(1) __m128i a = _mm_add_epi32 (x, y);
(2) return _mm_min_epu32 (a, _mm_sub_epi32 (a, p)); }
```

Since the `min` operation does not exist on packed 64-bit integers, we use the following function, where $\vec{0}$ represents the packed n-bit integer of type `__m128i` filled with 0:

FUNCTION 2.3.
Input: Packed 64-bit integers $\vec{x}$ and $\vec{y}$ modulo $\vec{p}$, assuming $m \leqslant n-1$.
Output: $(\vec{x}+\vec{y})$ rem $\vec{p}$.

```
__m128i add_mod_63_epu64 (__m128i x, __m128i y) {
(1) __m128i a = _mm_sub_epi64 (_mm_add_epi64 (x, y), p);
(2) __m128i b = _mm_cmpgt_epi64 (0, a);
(3) return _mm_add_epi64 (a, _mm_and_si128 (b, p)); }
```

If $m = n$, then we can proceed as follows: $\max(x, p-y)$ equals x if and only if $x \geqslant p-y$. In that case, $(x+y)$ rem p can be obtained as $x-(p-y)$ and otherwise as $x-(p-y)+p$. In the extreme case $p = 2^n$, an overflow only occurs when $y = 0$. Nevertheless, `max` $(x,\ p-y)$ equals x when computed in `U` and $x-(p-y)$ is the correct value. These calculations can be vectorized for packed integers of bit-size 8, 16, and 32 as follows:

FUNCTION 2.4.
Input: Packed 32-bit integers $\vec{x}$ and $\vec{y}$ modulo $\vec{p}$.
Output: $(\vec{x}+\vec{y})$ rem $\vec{p}$.

```
__m128i add_mod_epu32 (__m128i x, __m128i y) {
(1) __m128i a = _mm_sub_epi32 (p, y);
(2) __m128i b = _mm_cmpeq_epi32 (_mm_max_epu32 (x, a), x);
```

Table I. Sum of Large Vectors of Integers, in CPU Clock Cycles

n	8	16	32	64	128
Scalar	1.0	1.1	1.2	2.2	3.6
SSE 4.2	0.086	0.17	0.34	0.69	N/A
AVX 2	0.042	0.086	0.18	0.34	N/A

```
(3) __m128i c = _mm_andnot_si128 (b, p);
(4) return _mm_add_epi32 (_mm_sub_epi32 (x, a), c); }
```

The maximum operator and comparisons do not exist for packed 64-bit unsigned integers so we declare the function `_mm_cmpgt_epu64 (__m128i a, __m128i b)` as:

$$\texttt{\_mm\_cmpgt\_epi64 (\_mm\_sub\_epi64 (}\vec{a}, \overrightarrow{2^{63}}\texttt{), \_mm\_sub\_epi64 (}\vec{b}, \overrightarrow{2^{63}}\texttt{))}$$

where $\overrightarrow{2^{63}}$ represents the packed 64-bit integer filled with 2^{63}. The modular addition can be realized as follows:

FUNCTION 2.5.
Input: Packed 64-bit integers $\vec{x}$ and $\vec{y}$ modulo $\vec{p}$.
Output: $(\vec{x} + \vec{y}) \operatorname{rem} \vec{p}$.

```
__m128i add_mod_epu64 (__m128i x, __m128i y) {
(1) __m128i a = _mm_add_epi64 (x, y);
(2) __m128i b = _mm_or_si128 (_mm_cmpgt_epu64 (x, a),
                              _mm_cmpgt_epu64 (a, p - 1));
(3) return _mm_sub_epi64 (a, _mm_and_si128 (b, p)); }
```

It is straightforward to adapt these functions to the AVX 2 instruction set.

2.1.3. Timings. Timings presented in this section and the next one are obtained by means of our file `algebramix/bench/vector_bench.cpp` for the scalar versions and `algebramix/bench/vector_sse_bench.cpp` and `algebramix/bench/vector_avx_bench.cpp` for vectorized versions. The C++ code corresponding to what we call *naive* and *unrolled* loops in the sequel is shown in Section 4.2. In order to precisely measure timings in clock cycles, we use the assembly instruction `rdtsc`.

Table I displays timings for arithmetic operations over integer types of all possible bit-sizes n supported by the compiler. For conciseness, we restrict this table to addition; subtraction and negation behave in a similar way. Timings are the average number of clock cycles when applying the addition on two vectors of byte-size 2,048 aligned in memory and writing the result into the first vector (what is usually called *in-place sum*). In particular, these timings include *load* and *store* instructions for moving data between registers and main memory. The row "Scalar" corresponds to disabling auto-vectorization extensions with the command line option `-fno-tree-vectorize` of the compiler and to unrolling loops every 16 entries. For better performance, the optimization options `-O3 -mtune=native` of the compiler are enabled.

In other rows of the table, we indicate timings obtained by the vectorized algorithms according to the SIMD instruction set (timings are still the average per n-bit integers). Loops are unrolled every four SIMD vectors. Naturally, when we use AVX instructions, we implicitly mean that 256-bit registers and instructions are used. When we run SSE compatible binaries on our AVX 2-enabled processor, only the first 128 bits of the registers are used (this mode is sometimes called AVX-128). Let us underline that the

Table II. Modular Sum of Large Vectors of Integers in CPU Clock Cycles

n	8		16		32		64		128	
m	7	8	15	16	31	32	63	64	127	128
Scalar	3.0	3.0	3.1	3.2	2.6	2.9	2.7	4.1	14	15
SSE 4.2	0.13	0.20	0.25	0.42	0.52	0.80	1.4	2.1	N/A	N/A
AVX 2	0.058	0.088	0.13	0.19	0.26	0.39	0.72	1.0	N/A	N/A

vectorization process is not left to the compiler. In fact, we implemented a dedicated framework in MATHEMAGIX, which is sketched in Section 4, in order to align data and unroll loops. Operations that are not supported are indicated by N/A, meaning *Non Applicable*.

Remark 2.6. Instead of operating on large vectors it often happens that several operations are performed on a small amount of memory. The most favorable case is when all data fit into CPU registers. In this case, the sole AVX sum of eight 32-bit integers takes 0.5 cycle, which is significantly faster than in Table I. This reveals the important relative cost of moving data from and to the memory. With the SSE 4.2 and AVX 2 technologies, each in-place SIMD addition in fact performs two loads, one arithmetic and one aligned store instructions. One of the two loads can be avoided by passing a memory address as an argument to the sum instruction.

Remark 2.7. Roughly speaking, in the AVX 2 context, when loops are unrolled every four SIMD vectors, one may execute all the eight vectorized load instructions first, then the four vectorized sums, followed by the four vectorized store instructions. Since 16 SIMD registers are available, the latency chain may thus be discarded in first approximation. The throughputs are 1.5 cycles for two loads and one store instructions, and 0.5 for one addition, which amounts to 2 cycles. It is even better to fuse one load with the addition, so we can perform eight 32-bit sums in about 1.5 cycles, which is essentially observed with our implementation.

Remark 2.8. Letting the compiler do auto-vectorization leads to slower timings. For instance, with 32-bit integers, we measure 0.35 cycle per addition with AVX 2 instructions. This demonstrates the advantage of our own loop unrolling mechanism, and the lack of overhead in our implementation.

Table II shows timings for in-place modular sums of vectors of 2,048 bytes, with loops suitably unrolled: every eight entries in the "Scalar" row, and every four SIMD vectors in the vectorized implementations. In absence of vectorization, 8-bit and 16-bit arithmetic operations are in fact performed via 32-bit instructions. Indeed, 8-bit and 16-bit arithmetic is handled in a suboptimal way by current processors and compilers. For our vectorized implementations, we observe significant speedups when $m \leqslant n - 1$. Nevertheless, when $m = n$, the lack of efficiency is not dramatic enough to justify using larger integers and doubling memory space.

Remark 2.9. With $n = 32$ and $m = 31$, the sum of the throughputs of the AVX 2 instructions of Function 2.2 yields an estimate of 2.5 cycles per SIMD modular sum, which leads to 0.31 cycle per entry. Our executable code is therefore very well pipelined. We further observe how packed 64-bit integers are penalized by the lack of a genuine vectorized min operator.

Remark 2.10. In the row "Scalar," if we let the compiler auto-vectorize with the AVX 2 instructions, then timings are significantly improved but remain higher than

Table III. Values for the Parameters α, s, t, and h

	$m \leqslant n-2$	$m \leqslant n-1$	$m \leqslant n$
α	2^{n-2}	2^{n-1}	2^n
s	$\max(r-2, 0)$	$r-1$	$r-1$
t	$n+1$	n	n
h	1	2	3

with our vectorized implementation. For instance, with $n = 32$ and $m = 31$, we measure an average of 0.54 cycles.

2.2. Barrett's Product

The first modular product we describe is the one classically attributed to Barrett. It has the advantage of operating on integer types with no assumption on the modulus. For any nonnegative real number x, we write $\lfloor x \rfloor$ for the largest integer less or equal to x, and $\lceil x \rceil$ for the smallest integer greater or equal to x. We use the following auxiliary quantities:

—the nonnegative integer $r := \lceil \log p / \log 2 \rceil$ of p with $2^{r-1} < p \leqslant 2^r$;
—nonnegative integers s and t such that $t \geqslant r$ and $s + t \leqslant n + r - 1$;
—the integer $q := \lfloor \frac{2^{s+t}}{p} \rfloor$ represents an approximation of a rescaled numerical inverse of p.

Since $2^{s+t}/p < 2^{n+r-1}/2^{r-1} = 2^n$, the integer q fits in U. We call q the *pre-inverse* of p. Since $s + t \leqslant 2n - 1$, the computation of q just requires one division in L. In this subsection, both integers p and q are assumed to be of type U. We describe below how to set suitable values for s and t in terms of r (see Table III).

2.2.1. Reduction. Let α be one more auxiliary quantity with $\alpha 2^t \leqslant 2^{2n}$. If a is an integer such that $0 \leqslant a < \alpha p$, then Barrett's algorithm computes $a \operatorname{rem} p$ as follows:

FUNCTION 2.11.
Input: An integer a such that $0 \leqslant a < \alpha p$, with p, q, r, s, t, and α as above.
Output: $a \operatorname{rem} p$.

```
U reduce_barrett (L a) {
(1) L b = a >> s;
(2) L c = (b * q) >> t;
(3) L d = a - c * p;
(4) while (d >= p) d = d - p;
(5) return d; }
```

PROPOSITION 2.12. *Function 2.11 is correct. The number of iterations in the "while" loop of step 4 is at most h, where $h := \lceil \frac{2^s}{2^{r-1}} + \frac{\alpha 2^r}{2^{s+t}} \rceil$ if $s \geqslant 1$ and $h := \lceil \frac{\alpha 2^r}{2^t} \rceil$ if $s = 0$. In addition, bq and cp fit in* L.

PROOF. From $a < \alpha p$, it follows that $bq \leqslant aq/2^s < \alpha pq/2^s \leqslant \alpha 2^t$. Therefore, bq has bit-size at most $2n$, and $c < \alpha$. Since $r \leqslant t$ the product cp fits in L. From

$$c - \left\lfloor \frac{a}{p} \right\rfloor = \left(\left\lfloor \frac{bq}{2^t} \right\rfloor - \frac{bq}{2^t} \right) + \frac{q}{2^t} \left(b - \frac{a}{2^s} \right) + \frac{a}{2^{s+t}} \left(q - \frac{2^{s+t}}{p} \right) + \left(\frac{a}{p} - \left\lfloor \frac{a}{p} \right\rfloor \right),$$

we obtain $-1 - \frac{q}{2^t} - \frac{a}{2^{s+t}} < c - \lfloor \frac{a}{p} \rfloor < 1$. If $s \geqslant 1$, then $\frac{q}{2^t} + \frac{a}{2^{s+t}} \leqslant h$, so $-h \leqslant c - \lfloor \frac{a}{p} \rfloor \leqslant 0$, whence the conclusion follows. If $s = 0$, then $b = a$, and we still have $-h \leqslant c - \lfloor \frac{a}{p} \rfloor \leqslant 0$. □

If z is an integer modulo p, and $(x_i)_{i\in\{1,\ldots,l\}}$ and $(y_i)_{i\in\{1,\ldots,l\}}$ are sequences of integers modulo p, then computing $(z + x_1y_1 + \cdots + x_ly_l) \bmod p$ is a central task to matrix and polynomial products. In order to minimize the number of reductions to be done, we wish to take α as large as possible. In general, in Barrett's algorithm, we can always take $\alpha = 2^n$, $t = n$, and $s = r - 1$, which leads to $h = 3$. When $r \leqslant n - 1$, we can achieve $h = 2$ if we restrict to $\alpha = 2^{n-1}$ with $t = n$ and $s = r - 1$. When $2 \leqslant r \leqslant n - 2$, it is even better to take $t = n + 1$ and $s = r - 2$ so $h = 1$ when $\alpha = 2^{n-2}$. When $r \leqslant 1$, then we let $t = n+1$ and $s = 0$ when $\alpha = 2^{n-2}$ to get $h = 1$. These possible settings are summarized in Table III.

Remark 2.13. If $r = m = n$, then the modular product can be further optimized to reach $h = 2$, by using the algorithm from Möller and Granlund [2011]. Roughly speaking, if $n = 32$, say, then the main idea consists in emulating the case $n = 33$ with 32-bit integers (one writes $q = 2^{33} + \hat{q}$, where $\hat{q}$ fits 32 bits and adapts Barrett's formulas). The vectorized version of this variant is possible but will not be investigated in the present article for brevity. In fact, in the special case when $r = m = n$ and p is odd, it is even better to use Montgomery's algorithm, considered below.

We could consider taking $\bar{q} := \lceil 2^{s+t}/p \rceil$ instead of q. From $s + t \leqslant n + r - 1$ and $2^n \geqslant 2^{r-1} + 1$ we obtain $2^{s+t} \leqslant 2^{n+r-1} \leqslant 2^{n+r-1} + 2^n - 2^{r-1} - 1 = (2^n - 1)(2^{r-1} + 1)$. Therefore, the inequalities $\frac{2^{s+t}}{p} \leqslant \frac{2^{s+t}}{2^{r-1}+1} \leqslant 2^n - 1$ imply that $\bar{q}$ fits in `U`. Using $\bar{q}$ instead of q in Function 2.11 leads to the following inequalities:

$$-1 - \frac{\bar{q}}{2^t} < c - \left\lfloor \frac{a}{p} \right\rfloor < \frac{a}{2^{s+t}} + 1 - \frac{1}{p}.$$

If $s = 0$, then the term $\frac{\bar{q}}{2^t}$ disappears. If r is sufficiently small, then $\frac{a}{2^{s+t}}$ can be bounded by $\frac{1}{p}$. Therefore, line 4 of Function 2.11 can be discarded. More precisely, if we assume that $m \leqslant (n-1)/2$ and letting $s := 0$, $t := n + r - 1$, and $\bar{q} := \lceil 2^t/p \rceil$, then we obtain the following faster function:

FUNCTION 2.14.
Input: An integer a with $0 \leqslant a < 2^{(n-1)/2}p$, and p, $\bar{q}$, $t = n + r - 1$, $m \leqslant \frac{n-1}{2}$ as above.
Output: a rem p.

```
U reduce_barrett_half (U a) {
(1) U c = (a * ((L) q̄)) >> t;
(2) return a - c * p; }
```

PROPOSITION 2.15. *Function 2.14 is correct.*

PROOF. Letting $\alpha := 2^{(n-1)/2}$, the proof follows from $-1 < c - \lfloor \frac{a}{p} \rfloor < \frac{a}{2^t} + 1 - \frac{1}{p}$ and $\frac{ap}{2^t} < \frac{\alpha p^2}{2^t} \leqslant \alpha 2^{2r-t} \leqslant \alpha 2^{r-n+1} \leqslant 1$. □

Remark 2.16. With recent compilers, 128-bit integers are available, which eases the implementation of Barrett's algorithm up to 64-bit. But if `U` is the largest type of integers supported by the compiler, then q has to be computed by alternative methods such as the classical Newton-Raphson division [Schulte et al. 1994] (see also our implementation in `numerix/modular_int.hpp` inside MATHEMAGIX, with the necessary proofs in the documentation). The computation of the pre-inverse has also been optimized in Möller and Granlund [2011] for special case $r = m = n$.

2.2.2. Several Multiplications with One Fixed Argument. Assume that we wish to compute several modular products, where one of the multiplicands is a fixed modular integer $y \in \mathbb{Z}/p\mathbb{Z}$. A typical application is the computation of FFTs, where we need to multiply

by roots of unity that only depend on the size of the transformer; these roots need to be computed only once and can be cached into memory. We reuse the same approach as implemented in NTL [Shoup 2015] and explained for instance by Harvey [2014]. We pre-compute $\psi := \lfloor (2^n y)/p \rfloor < 2^n$ and obtain a speedup within each product by y thanks to the following function:

FUNCTION 2.17.
Input: Integers x, y in $\{0, \ldots, p-1\}$, and $\psi < 2^n$, with p and ψ as above.
Output: $(xy) \operatorname{rem} p$.

```
U mul_mod_barrett (U x, U y, U ψ) {

(1) U c = (x * ((L) ψ)) >> n;
(2) L d = ((L) x) * y - ((L) c) * p;
(3) return (d >= p) ? d - p : d; }
```

PROPOSITION 2.18. *Function 2.17 is correct.*

PROOF. From $c - \lfloor \frac{xy}{p} \rfloor = (\lfloor \frac{x\psi}{2^n} \rfloor - \frac{x\psi}{2^n}) + \frac{x}{2^n}(\psi - \frac{2^n y}{p}) + (\frac{xy}{p} - \lfloor \frac{xy}{p} \rfloor)$ we obtain $-1 - \frac{x}{2^n} < c - \lfloor \frac{xy}{p} \rfloor < 1$, whence the correctness. □

Notice that Function 2.17 does not depend on q, r, s, or t. If $m \leqslant n-1$, then line 2 can be replaced by `U d = x * y - c * p`. This saves one subtraction and one test when compared to calling Function 2.11 on `x * y` and makes the shift in step 1 a bit cheaper.
If $m \leqslant n/2$, then $\bar{\psi} := \lceil (2^n y)/p \rceil$ fits in `U` since $(2^n y)/p \leqslant 2^n - 2^n/p$, and Function 2.17 can be improved along the same lines as Function 2.14:

FUNCTION 2.19.
Input: Integers x, y in $\{0, \ldots, p-1\}$, $\bar{\psi} < 2^n$, with p and $\bar{\psi}$ as above, and where $m \leqslant n/2$.
Output: $(xy) \operatorname{rem} p$.

```
U mul_mod_barrett_half (U x, U y, U ψ̄) {

(1) U c = (x * ((L) ψ̄)) >> n;
(2) return x * y - c * p; }
```

PROPOSITION 2.20. *Function 2.19 is correct.*

PROOF. Similarly to previous proofs, we have $-1 < c - \lfloor \frac{xy}{p} \rfloor < \frac{x}{2^n} + 1 - \frac{1}{p}$, whence the correctness. □

2.2.3. Implementations with SSE *4.2 and* AVX *2.* Function 2.11 could be easily vectorized if all the necessary elementary operations were available within the SSE 4.2 or AVX 2 instruction sets. Unfortunately, this is not so for all integer sizes, which forces us to examine different cases. As a first remark, in order to remove the branching involved in line 4 of Function 2.11, we replace it by `d = min (d, d - p)` as many times as specified by Proposition 2.12. In the functions below, we consider the case when $m > n/2$. The other case is more straightforward. The packed $2n$-bit integer filled with q (respectively, p) seen of type `L` is written $\vec{q}_{\mathtt{L}}$ (respectively, $\vec{p}_{\mathtt{L}}$). The constant $\overrightarrow{2^n - 1}_{\mathtt{L}}$ corresponds to the packed $2n$-bit integer filled with $2^n - 1$. We start with the simplest case of packed 16-bit integers. If $\vec{x}$ and $\vec{y}$ are the two vectors of scalar type `U` to be multiplied modulo p, then we unpack each of them into two vectors of scalar type `L`, we perform all the needed SIMD arithmetic over `L`, and then we pack the result back to vectors over `U`.

FUNCTION 2.21.
Input: Packed 16-bit integers $\vec{x}$ and $\vec{y}$ modulo $\vec{p}$, assuming $m \leqslant n-2$.
Output: $(\vec{x}\vec{y})$ rem $\vec{p}$.

```
__m128i mul_mod_14_epu16 (__m128i x, __m128i y) {
(1)  __m128i x_l = _mm_unpacklo_epi16 (x, 0);
(2)  __m128i x_h = _mm_unpackhi_epi16 (x, 0);
(3)  __m128i y_l = _mm_unpacklo_epi16 (y, 0);
(4)  __m128i y_h = _mm_unpackhi_epi16 (y, 0);
(5)  __m128i a_l = _mm_mullo_epi32 (x_l, y_l);
(6)  __m128i a_h = _mm_mullo_epi32 (x_h, y_h);
(7)  __m128i b_l = _mm_srli_epi32 (a_l, s);
(8)  __m128i b_h = _mm_srli_epi32 (a_h, s);
(9)  __m128i c_l = _mm_srli_epi32 (_mm_mullo_epi32 (b_l, q_L), t);
(10) __m128i c_h = _mm_srli_epi32 (_mm_mullo_epi32 (b_h, q_L), t);
(11) __m128i c = _mm_packus_epi32 (c_l, c_h);
(12) __m128i d = _mm_sub_epi16 (_mm_mullo_epi16 (x, y),
                                _mm_mullo_epi16 (c, p);
(13) return _mm_min_epu16 (d, _mm_sub_epi16 (d, p)); }
```

If $m \leqslant n-1$, then the computations up to line 10 are the same but the lines after are replaced by:

```
(11) __m128i d_l = _mm_sub_epi32 (a_l, _mm_mullo_epi32 (c_l, p_L));
(12) __m128i d_h = _mm_sub_epi32 (a_h, _mm_mullo_epi32 (c_h, p_L));
(13) d_l = _mm_min_epu32 (d_l, _mm_sub_epi32 (d_l, p_L));
(14) d_h = _mm_min_epu32 (d_h, _mm_sub_epi32 (d_h, p_L));
(15) __m128i d = _mm_packus_epi32 (d_l, d_h);
(16) return _mm_min_epu16 (d, _mm_sub_epi16 (d, p)); }
```

Under the only assumption that $m \leqslant n$, one needs to duplicate the lines 13 and 14 twice and to simply return d in line 16.

For packed 8-bit integers, since no packed product is natively available, we simply perform most of the computations over packed 16-bit integers as follows:

FUNCTION 2.22.
Input: Packed 8-bit integers $\vec{x}$ and $\vec{y}$ modulo $\vec{p}$, assuming $m \leqslant n-2$.
Output: $(\vec{x}\vec{y})$ rem $\vec{p}$.

```
__m128i mul_mod_6_epu8 (__m128i x, __m128i y) {
(1)  __m128i x_l = _mm_unpacklo_epi8 (x, 0);
(2)  __m128i x_h = _mm_unpackhi_epi8 (x, 0);
(3)  __m128i y_l = _mm_unpacklo_epi8 (y, 0);
(4)  __m128i y_h = _mm_unpackhi_epi8 (y, 0);
(5)  __m128i a_l = _mm_mullo_epi16 (x_l, y_l);
(6)  __m128i a_h = _mm_mullo_epi16 (x_h, y_h);
(7)  __m128i b_l = _mm_srli_epi16 (a_l, s);
(8)  __m128i b_h = _mm_srli_epi16 (a_h, s);
(9)  __m128i c_l = _mm_srli_epi16 (_mm_mullo_epi16 (b_l, q_L), t);
(10) __m128i c_h = _mm_srli_epi16 (_mm_mullo_epi16 (b_h, q_L), t);
```

```
(11) __m128i d_l = _mm_sub_epi16 (a_l, _mm_mullo_epi16 (c_l, p_L));
(12) __m128i d_h = _mm_sub_epi16 (a_h, _mm_mullo_epi16 (c_h, p_L));
(13) __m128i d = _mm_packus_epi16 (d_l, d_h);
(14) return _mm_min_epu16 (d, _mm_sub_epi8 (d, p)); }
```

If $m \leqslant n-2$ does not hold, then the same modifications as with packed 16-bit integers are applied.

For packed 32-bit integers, a similar extension using packing and unpacking instructions is not directly possible since the corresponding packing function is not available in the instruction set. Instead, we take advantage of the `_mm_mul_epu32` instruction. If $m \leqslant n-2$, then we use the following code:

FUNCTION 2.23.
Input: Packed 32-bit integers $\vec{x}$, $\vec{y}$ modulo $\vec{p}$, assuming $m \leqslant n-2$.
Output: $(\vec{x}\vec{y}) \operatorname{rem} \vec{p}$.

```
__m128i mul_mod_30_epu32 (__m128i x, __m128i y) {
(1)  __m128i a_l = _mm_mul_epu32 (x, y);
(2)  __m128i b_l = _mm_srli_epi64 (a_l, s);
(3)  __m128i c_l = _mm_srli_epi64 (_mm_mul_epi32 (b_l, q_L), t);
(4)  __m128i x_h = _mm_srli_si128 (x, 4);
(5)  __m128i y_h = _mm_srli_si128 (y, 4);
(6)  __m128i a_h = _mm_mul_epu32 (x_h, y_h);
(7)  __m128i b_h = _mm_srli_epi64 (a_h, s);
(8)  __m128i c_h = _mm_srli_epi64 (_mm_mul_epi32 (b_h, q_L), t);
(9)  __m128i a = _mm_blend_epi16 (a_l, _mm_slli_si128 (a_h, 4),4+8+64+128);
(10) __m128i c = _mm_or_si128 (c_l, _mm_slli_si128 (c_h, 4));
(11) __m128i d = _mm_sub_epi32 (a, _mm_mullo_epi32 (c, p));
(12) return _mm_min_epu32 (d, _mm_sub_epi32 (d, p)); }
```

In line 9, we could, of course, compute $\vec{a}$ as `_mm_mullo_epi32 (x, y)`, but this is a bit slower. When $m \leqslant n-1$, the same kind of modifications as before have to be done: $\vec{d}$ must be computed with vectors of unsigned 64-bit integers. Since these vectors do not support the computation of the minimum, one has to use `_mm_cmpgt_epi64`.

The case $m = n$ turns out to be more difficult, since the entries of $\vec{b}_l$ and $\vec{b}_h$ may actually need 33 bits. For the sake of conciseness we will not reproduce our solution in full details and refer the reader to our source code. In fact, assuming $s = 31$ and $t = 32$, we still use `_mm_mul_epi32` instructions, but we detect and fix the casual overflow by vectorizing the following code that uses the same notation as in Function 2.11:

```
uint32_t b = a >> s;
uint64_t c = b * ((L) q);
c = c >> t;
if (a >= ((uint64_t) 1 << 63)) c += q;
L d = a - c * p;
```

Timings reported in the next paragraphs confirm that this approach lacks of efficiency. We recommend vectorizing the algorithm proposed by Möller and Granlund [2011], as sketched in our Remark 2.13, or using Montgomery's algorithm described below if p is odd.

As a last piece of code, we show our vectorized implementation of Function 2.17, for $n = 32$ and $m \leqslant n-1$, with SSE 4.2 instructions. Again, the AVX 2 variant is very similar, and the case $m = n$ can be adapted in the same way as for Function 2.23.

Table IV. Product of Large Vectors of Integers in CPU Clock Cycles

n	8	16	32	64	128
Scalar	1.0	1.1	1.2	2.1	5.3
SSE 4.2	0.23	0.19	0.52	N/A	N/A
AVX 2	0.12	0.094	0.26	N/A	N/A

Table V. Modular Product of Large Vectors of Integers in CPU Clock Cycles

n	8				16				32			
m	2	6	7	8	6	14	15	16	14	30	31	32
Naive	8.1	8.1	8.1	8.1	11	11	11	11	9.4	22	22	22
Barrett	3.5	7.9	9.9	13	3.3	6.2	9.5	11	3.3	6.1	9.2	11.5
Padded Barrett	3.3	3.3	3.3	3.3	3.3	3.3	3.3	6.1	N/A	N/A	N/A	N/A
SSE Barrett	0.75	0.75	0.81	1.1	0.78	1.9	2.5	3.1	2.2	3.0	3.6	6.5
AVX Barrett	0.37	0.40	0.43	0.56	0.40	0.94	1.3	1.5	1.1	1.5	1.8	3.3

FUNCTION 2.24.
Input: Packed 32-bit integers $\vec{x}$, $\vec{y}$, $\vec{\psi}$ modulo $\vec{p}$, assuming $m \leqslant n-1$.
Output: $(\vec{x}\vec{y})$ rem $\vec{p}$.

```
__m128i sc_mul_mod_31_epu32 (__m128i x⃗, __m128i y⃗, __m128i ψ⃗) {
(1) __m128i c⃗_l = _mm_mul_epu32 (x⃗, ψ⃗);
(2) __m128i x⃗_h = _mm_srli_si128 (x⃗, 4);
(3) __m128i ψ⃗_h = _mm_srli_si128 (ψ⃗, 4);
(4) __m128i c⃗_h = _mm_mul_epi32 (x⃗_h, ψ⃗_h);
(5) __m128i c⃗ = _mm_blend_epi16 (c⃗_h, _mm_srli_si128 (c⃗_l, 4), 1+2+16+32);
(6) __m128i d⃗ = _mm_sub_epi32 (_mm_mullo_epi32 (x⃗, y⃗),
                               _mm_mullo_epi32 (c⃗, p⃗));
(7) return _mm_min_epu32 (d⃗, _mm_sub_epi32 (d⃗, p⃗)); }
```

2.2.4. Timings. In Table IV we display timings for multiplying machine integers, using the same conditions as in Table I. Recall that packed 8-bit integers have no dedicated instruction for multiplication: It is thus done through the 16-bit multiplication via unpacking/packing.

Remark 2.25. Letting the compiler perform auto-vectorization with 32-bit integers, we measure 0.38 cycle per multiplication, which is higher than with our own code.

Table V shows the performance of the above algorithms. It is obtained by using the same benchmark files as for addition. For the unvectorized algorithms, loops are unrolled 16 by 16 in bit-sizes 8 and 16, 8 by 8 in sizes 32 and 64, and 4 by 4 in size 128. The row "Naive" corresponds to the scalar approach using the C++ operator % to compute remainders. Up to 32-bit integers, arithmetic is handled via 32- and 64-bit registers. The row "Barrett" contains timings for the scalar implementation of Barrett's product. Notice that the case $m = n-2$ is significantly faster than the case $m = n$.

In the scalar approach, compiler optimizations and hardware operations are not always well supported for small sizes, so it makes sense to perform computations on a larger size. On our test platform, 32-bit integers typically provide the best performance.

Table VI. Modular Product of Large Vectors of Integers in CPU Clock Cycles

n	64				128			
m	30	62	63	64	62	126	127	128
Naive	21	90	90	90	95	500	780	520
Barrett	9.5	17	26	29	72	95	150	160

Table VII. Modular Product for a Fixed Multiplicand in CPU Clock Cycles

n	8		16		32		64		128	
m	7	8	15	16	31	32	63	64	127	128
Padded Barrett	3.1	3.1	3.1	3.1	4.5	4.5	4.5	7.0	50	110
SSE Barrett	0.63	0.68	0.56	2.0	2.0	5.0	N/A	N/A	N/A	N/A
AVX Barrett	0.32	0.35	0.28	0.94	1.1	2.0	N/A	N/A	N/A	N/A

The resulting timings are given in the row "padded Barrett." For 8-bit integers, the best strategy is in fact to use lookup tables yielding 2.5 cycles in average, but this strategy cannot be vectorized. Finally, the last two rows correspond to the vectorized versions of Barrett's approach, where loops are unrolled every four SIMD vectors.

For larger integers, the performance is shown in Table VI. Let us mention that, in the row "Barrett" with $n = 64$, we actually make use of `__int128` integer arithmetic.

Table VII shows the average cost of one in-place product of a vector of 2,048 bytes with a fixed multiplicand. Comparison to Table V reveals that this approach is always faster than using generic products.

Remark 2.26. Our vectorized modular operations can easily be adapted to simultaneously compute modulo several moduli, say, p_1, p_2, p_3, p_4, assuming they share the same parameters r, s, and t. Nevertheless, instead of using vectors $\vec{p}_{\mathrm{L}}$ filled with the same modulus, this requires one vector of type `L` to be filled with p_1 and p_3 and a second one with p_2 and p_4; the same consideration holds for $\vec{q}_{\mathrm{L}}$. These modifications involve caching more pre-computations and a small overhead in each operation.

Remark 2.27. As explained before, timings measured in Table V for the unvectorized algorithms are obtained with the `-fno-tree-vectorize` compiler option that disables auto-vectorization. In fact, auto-vectorization is often worthy in simple situations and for numerical types but of a lesser interest when applied to integer types and modular arithmetic. For instance, in the typical case when $n = 32$ and $m = 30$, by removing the latter compiler option and allowing AVX 2 instructions, the code is reported to be vectorized by the `-fopt-info-vec` option, but we measure only 3.7 cycles per modular product. With $n = 16$ and $m = 14$, or $n = 8$ and $m = 6$, auto-vectorization seems to be also applied but leads to 4.1 cycles, which is even higher than our unvectorized version.

Remark 2.28. In some applications, the modulus is known at compile time, and the compiler is able to produce a faster executable code. For instance, writing `dest` and `s` for pointers to the first entry of two arrays over `int32_t`, the naive code

```
const int32_t p = 469762049;
for (int32_t i = 0; i < 1024; i++)
  dest[i] = (((int64_t) dest[i]) * s1[i]) % p;
```

uses 3.7 cycles per modular product when compiled with `-O3 -mavx2 -mtune=native`. In fact, Barrett's algorithm seems to be used by the compiler, but, unfortunately, it

seems that auto-vectorization is not achieved (in other words SIMD instructions are not used). This is to be compared to the case of Table V, with $n = 32$ and $m = 30$.

2.3. Montgomery's Product

For Montgomery's algorithm [Montgomery 1985], one needs to assume that p is odd. Let m be such that $r \leqslant m \leqslant n$ and let $a < 2^m p$. We need the auxiliary quantities ρ and χ defined by $0 < \chi < 2^m$, $0 < \rho < p$, and $\rho 2^m - \chi p = 1$, which can be computed with the extended Euclidean algorithm [von zur Gathen and Gerhard 2003, Chapter 3].

2.3.1. Reduction. For convenience, we introduce $\mu := 2^m - 1$. The core of Montgomery's algorithm is the following reduction function:

FUNCTION 2.29.
Input: An integer a such that $0 \leqslant a < 2^m p$, with p odd, and m as above.
Output: $(a\rho) \operatorname{rem} p$.

```
U reduce_montgomery (U a) {
(1) U b = (a * χ) & μ;
(2) L c = a + b * p;
(3) U d = c >> m;
(4) if (c < a) return d − p;
(5) return (d >= p) ? d − p : d; }
```

PROPOSITION 2.30. *Function 2.29 is correct. If $m \leqslant n-1$, then line 4 can be discarded.*

PROOF. First, one verifies that $bp \bmod 2^m = a\chi p \bmod 2^m = -a \bmod 2^m$. Therefore $a+bp$ is a multiple of 2^m. If $m \leqslant n-1$, then no overflow occurs in the sum of line 2, and the division in line 3 is exact. We then have $d2^m \bmod p = a \bmod p$, and the correctness follows from $0 \leqslant c < 2^m p + 2^m p$ and thus $0 \leqslant d < 2p$. If $m = n$, then the casual overflow in line 2 is tested in line 4, and d is the value in U of $(a + bp)/2^m$. □

Let x be a modulo p integer. We say that x is in *Montgomery's representation* if stored as $(x2^m) \operatorname{rem} p$. The product of two modular integers x and y, of respective Montgomery's representations $\tilde{x}$ and $\tilde{y}$, can be obtained in Montgomery's representation $xy2^m \operatorname{rem} p$ by applying Function 2.29 to $\tilde{x}\tilde{y}$ since $xy2^m \bmod p = (x2^m)(y2^m)\rho \bmod p$.

If $m = n$, then the mask in line 1 can be avoided, and the shift in line 3 might be more favorable than a general shift, according to the compiler. In total, if $m = n$ or $m = n-1$, Montgomery's approach is expected to be faster than Barrett's one. Otherwise, costs should be rather similar. Of course, these considerations are rather informal and the real cost very much depends on the processor and the compiler.

Remark 2.31. As for Barrett's algorithm one could be interested in simplifying Montgomery's product when performing several products by the same multiplicand y. Writing $\varphi = (\chi y) \operatorname{rem} 2^m$, the only simplification appears in line 1, where b can be obtained as $(x\varphi) \operatorname{rem} 2^m$, which saves one product in U. Therefore Barrett's approach is expected to be always faster for this task. Precisely, if $m \leqslant n-1$, this is to be compared to Function 2.17 which performs only one high product in line 1.

Remark 2.32. Up to our best knowledge, it is not known whether Montgomery's product can be improved in a similar way as we did for Barrett's product, in the case when $m \leqslant n/2$.

2.3.2. Implementations with SSE *4.2 and* AVX *2.* In the following code, we show our vectorized implementation of Function 2.29 when $n = 32$ and $m \leqslant n-1$, with SSE 4.2 instructions. Recall that $\vec{\chi}_{\mathrm{L}}$ and $\vec{\mu}_{\mathrm{L}}$ represent packed 64-bit integers, respectively, filled with χ and μ.

Table VIII. Montgomery's Product in CPU Clock Cycles

n	8		16		32		64		128	
m	7	8	15	16	31	32	63	64	127	128
Montgomery	6.0	6.4	6.2	6.6	5.6	6.0	7.5	8.2	94	90
SSE Montgomery	0.69	1.0	2.0	3.0	3.1	4.2	N/A	N/A	N/A	N/A
AVX Montgomery	0.37	0.52	1.1	1.5	1.6	2.2	N/A	N/A	N/A	N/A

FUNCTION 2.33.
Input: Packed 32-bit integers $\vec{x}$, $\vec{y}$ modulo $\vec{p}$ in Montgomery representation, assuming $m \leqslant n-1$.
Output: $(\vec{x}\vec{y})$ rem $\vec{p}$ in Montgomery representation.

```
__m128i montgomery_mul_mod_31_epu32 (__m128i x, __m128i y) {
(1)  __m128i a_l = _mm_mul_epu32 (x, y);
(2)  __m128i b_l = _mm_and_si128 (_mm_mullo_epi32 (a_l, χ_L), μ_L);
(3)  __m128i c_l = _mm_add_epi64 (a_l, _mm_mul_epi32 (b_l, p_L));
(4)  __m128i d_l = _mm_srli_epi64 (c_l, m);
(5)  __m128i x_h = _mm_srli_si128 (x, 4);
(6)  __m128i y_h = _mm_srli_si128 (y, 4);
(7)  __m128i a_h = _mm_mul_epu32 (x_h, y_h);
(8)  __m128i b_h = _mm_and_si128 (_mm_mullo_epi32 (a_h, χ_L), μ_L);
(9)  __m128i c_h = _mm_add_epi64 (a_h, _mm_mul_epi32 (b_h, p_L));
(10) __m128i d_h = _mm_srli_epi64 (c_h, m);
(11) __m128i d = _mm_blend_epi16 (d_l, _mm_slli_si128 (d_h, 4),4+8+64+128);
(12) return _mm_min_epu32 (d, _mm_sub_epi32 (d, p)); }
```

2.3.3. Timings. Table VIII contains timings measured in the same manner as in the previous subsection by taking averages for in-place operations on vectors of 2,048 bytes. Compared to Tables V and VI, we observe that Montgomery's product is not interesting in the scalar case for 8- and 16-bit integers but it outperforms Barrett's approach in larger sizes, especially when $m = n$. On the other hand, when compared to Table VII, Montgomery's product is not competitive for several products with a fixed multiplicand, unless $n = m = 128$.

Remark 2.34. Similar implementations of Montgomery's algorithm have been carried out by Meng and Johnson [2014] and Meng et al. [2010] for SSE instructions, with applications to FFT codelet generation. In comparison, let us mention that our version of Barrett's algorithm with a fixed multiplicand is faster whenever $m \leqslant n-1$. Nevertheless, we did not tune codelets for small FFT sizes. Instead, we rely on a vectorized variant of the TFT explained in Section 5.

3. MODULAR OPERATIONS VIA NUMERIC TYPES

Instead of integer types, we can use numeric types such as `float` or `double` to perform modular operations. Let us write `F` such a type, and let $\ell \geqslant 5$ represent the size of the mantissa of `F` minus 1, that is, 23 bits for `float` and 52 bits for `double`. This number of bits corresponds to the size of the so-called *trailing significant field* of `F`, which is explicitly stored. The modular product of x mod p and y mod p begins with the computation of an integer approximation c of xy/p. Then $d = xy - cp$ is an approximation of

xy rem p at distance $O(p)$. The constant hidden behind the latter O depends on rounding modes. In this section, we conform to the IEEE-754 standard. We first analyze the case when p fills less than half of the mantissa. We next propose a general algorithm using the fused multiply-add operation.

3.1. Notations

The following notations are used until the end of the present section. We write u for the value of $1/p$ computed in `F` by applying the division operator on `1.0` and `(F) p`. Still following IEEE-754, the trailing significant field of u is written υ and its exponent e. These quantities precisely depend on the active rounding mode used to compute u. But for all rounding modes, they satisfy the following properties:

$$0 \leqslant \upsilon < 2^\ell, \qquad u = \frac{2^\ell + \upsilon}{2^e}, \qquad \left|\frac{1}{p} - u\right| < \frac{1}{2^e}. \tag{1}$$

From the latter inequality we obtain

$$2^\ell - 1 \leqslant -1 + 2^\ell + \upsilon < \frac{2^e}{p} < 1 + 2^\ell + \upsilon \leqslant 2^{\ell+1}$$

and thus

$$\frac{p}{2^e} < \frac{1}{2^\ell - 1}. \tag{2}$$

Let a be a real number of type `F`, and let `F b = a * u` be the approximation of au computed in `F`. Again, independently of the rounding mode, there exist integers β and f, for the trailing significant field and the exponent of b, such that

$$0 \leqslant \beta < 2^\ell, \qquad b = \frac{2^\ell + \beta}{2^f}, \qquad |au - b| < \frac{1}{2^f}. \tag{3}$$

From $2^\ell - 1 < 2^f au < 2^{\ell+1}$ we deduce

$$\frac{1}{2^f} < \frac{au}{2^\ell - 1}. \tag{4}$$

We also use the approximation $\bar{u}$ of `1.0 / ((F) p)` computed in `F` with rounding mode set towards infinity, so $\bar{u} \geqslant 1/p$ holds.

3.2. Reduction in Half Size

When p fills no more than half of the mantissa, that is, when $m \leqslant \ell/2$, it makes it possible to perform modular products in `F` easily. Let the `floor` function return the largest integral value less than or equal to its argument, as defined in C99.

FUNCTION 3.1.
Input: An integer a such that $0 \leqslant a < \alpha p$, where $\alpha := 2^{\ell/2}$, and $m \leqslant \ell/2$.
Output: a rem p.

```
F reduce_numeric_half (F a) {
(1) F b = a * u;
(2) F c = floor (b);
(3) F d = a - c * p;
(4) if (d >= p) return d - p;
(5) if (d < 0) return d + p;
(6) return d; }
```

Proposition 3.2. *Function 3.1 is correct for any rounding mode. In addition, if $\bar{u}$ is used instead of u in line 1, and if the rounding mode is set towards infinity, then line 4 can be discarded.*

Proof. Using Equations (1) and (2), we obtain $au = \frac{a}{p}up < \alpha(1+\frac{p}{2^e}) < 2^{\ell/2}(1+\frac{1}{2^{\ell}-1})$, hence the `floor` function actually returns $\lfloor b \rfloor$ in c. From the decomposition

$$c - \left\lfloor \frac{a}{p} \right\rfloor = (\lfloor b \rfloor - b) + (b - au) + a\left(u - \frac{1}{p}\right) + \left(\frac{a}{p} - \left\lfloor \frac{a}{p} \right\rfloor\right), \tag{5}$$

we deduce

$$-1 - \frac{1}{2^f} - \frac{\alpha p}{2^e} < c - \left\lfloor \frac{a}{p} \right\rfloor < \frac{1}{2^f} + \frac{\alpha p}{2^e} + 1.$$

From Equation (2), we have $\frac{\alpha p}{2^e} < \frac{1}{2}$, and from Equation (4) we deduce $\frac{1}{2^f} < \frac{au}{2^{\ell}-1} \leqslant \frac{\alpha p(1/p+1/2^e)}{2^{\ell}-1} \leqslant \frac{\alpha+\alpha p/2^e}{2^{\ell}-1} \leqslant \frac{2^{\ell/2}+1/2}{2^{\ell}-1} < \frac{1}{2}$. It follows that $|c - \lfloor \frac{a}{p} \rfloor| < 2$ and hence that $|c - \lfloor \frac{a}{p} \rfloor| \leqslant 1$.

If using $\bar{u}$ instead of u, and if the rounding mode is set towards infinity, then $b \geqslant a\bar{u}$, and then $0 \leqslant c - \lfloor \frac{a}{p} \rfloor \leqslant 1$, which allows us to discard line 4. □

In the same way we did for Barrett's product, and under mild assumptions, we can improve the latter function whenever $m \leqslant (\ell - 1)/2$.

Function 3.3.
Input: An integer a such that $0 \leqslant a < \alpha p$, where $\alpha := 2^{(\ell-1)/2}$ and $m \leqslant (\ell-1)/2$.
Hypothesis: The current rounding mode rounds towards infinity.
Output: a rem p.

```
F reduce_numeric_half_1 (F a) {
(1) F b = a * ū;
(2) F c = floor (b);
(3) return a - c * p; }
```

Proposition 3.4. *Function 3.3 is correct.*

Proof. We claim that

$$\frac{1}{2^f} + \frac{\alpha p - 1}{2^e} \leqslant \frac{1}{p}. \tag{6}$$

By Equations (2) and (4), the claim is satisfied as soon as $\frac{(\alpha p-1)(1+p/2^e)}{2^{\ell}-1} + \frac{\alpha p-1}{2^{\ell}-1} \leqslant 1$, which is itself implied by $(2^{\ell-1}-1)(1+\frac{1}{2^{\ell}-1})+2^{\ell-1}-1 \leqslant 2^{\ell}-1$, which is correct since it rewrites into $\frac{2^{\ell-1}-1}{2^{\ell}-1} \leqslant 1$ by expanding the product. From $0 \leqslant \frac{a}{p} - \lfloor \frac{a}{p} \rfloor \leqslant 1 - \frac{1}{p}$ and Equation (5) we deduce that

$$-1 < c - \left\lfloor \frac{a}{p} \right\rfloor < 1 + \frac{1}{2^f} + \frac{\alpha p - 1}{2^e} - \frac{1}{p} \leqslant 1,$$

which proves the correctness. □

Remark 3.5. The hypothesis on the rounding mode can be dropped if $a = 0$ or p does not divide a. This is, in particular, the case if p is prime and a is the product of two numbers in $\{0, \ldots, p-1\}$. Indeed, if $a = 0$, then the algorithm is clearly correct. If p does not divide a, then we have $\frac{1}{p} \leqslant \frac{a}{p} - \lfloor \frac{a}{p} \rfloor \leqslant 1 - \frac{1}{p}$. Combining the latter inequality

with $-\frac{1}{2^f} < b - au < \frac{1}{2^f}$ and $|au - \frac{a}{p}| < \frac{\alpha p - 1}{2^f}$ yields

$$-1 - \frac{1}{2^f} - \frac{\alpha p - 1}{2^f} + \frac{1}{p} < b - \left\lfloor \frac{a}{p} \right\rfloor < 1 + \frac{1}{2^f} + \frac{\alpha p - 1}{2^e} - \frac{1}{p}.$$

Inequality (6) finally implies $-1 < b - \lfloor \frac{a}{p} \rfloor < 1$.

3.3. Larger Modular Products via FMA

Until now, we have not exploited the whole mantissa of F. To release the assumption $m \leqslant \ell/2$ in Function 3.1, the value for d could be computed over a sufficiently large integer type. Over `double`, one can use 64-bit integers, as implemented, for instance, in NTL [Shoup 2015]. The drawbacks of this approach are the extra conversions between numeric and integer types and the fact that the vectorization is compromised since 64-bit integer products are not natively available in the SSE or AVX instruction sets. In what follows, we describe a modular product in the case when $m \leqslant \ell - 2$, using the fused multiply-add operation from the IEEE 754-2008 standard. We write `fma (x, y, z)` for the evaluation of $xy + z$ using the current rounding mode. The following function begins with the "TwoProduct" algorithm of Ogita et al. [2005]:

FUNCTION 3.6.
Input: Integers x and y modulo p, and $m \leqslant \ell - 2$.
Output: (xy) rem p.

```
F mul_mod_fma (F x, F y) {
(1) F h = x * y;
(2) F l = fma (x, y, -h);
(3) F b = h * u;
(4) F c = floor (b);
(5) F d = fma (-c, p, h);
(6) F g = d + l;
(7) if (g >= p) return g - p;
(8) if (g < 0) return g + p;
(9) return g; }
```

PROPOSITION 3.7. *Function 3.6 is correct for any rounding mode. If $\bar{u}$ is used instead of u in line 3, and if the rounding mode is set towards infinity, then line 7 can be discarded.*

PROOF. Let $a := xy$. We have $a \leqslant (p-1)^2 \leqslant 2^{2r} - 2^{r+1} + 1$, hence $|h - a| < 2^{2r-\ell} \leqslant 2^{r-2} \leqslant 2^{\ell-4}$, so $h + l = a$. We also verify that $\frac{h}{p} \leqslant \frac{2^{r-2}}{p} + \frac{a}{p} < \frac{2^{r-2}}{2^{r-1}} + p - 1 \leqslant p - \frac{1}{2}$. Using Equation (2), it follows that

$$hu = \frac{h}{p} up < \left(p - \frac{1}{2}\right)\left(1 + \frac{p}{2^e}\right) < 2^{\ell-2}\left(1 + \frac{1}{2^\ell - 1}\right) < 2^\ell,$$

which implies that $c = \lfloor b \rfloor$. From

$$c - \left\lfloor \frac{a}{p} \right\rfloor = (\lfloor b \rfloor - b) + (b - hu) + h\left(u - \frac{1}{p}\right) + \frac{h - a}{p} + \left(\frac{a}{p} - \left\lfloor \frac{a}{p} \right\rfloor\right),$$

we deduce that

$$-1 - |b - hu| - h\left|u - \frac{1}{p}\right| - \frac{|h - a|}{p} < c - \left\lfloor \frac{a}{p} \right\rfloor < |b - hu| + h\left|u - \frac{1}{p}\right| + \frac{|h - a|}{p} + 1.$$

First, we have $\frac{|h-a|}{p} \leqslant \frac{2^{r-2}}{2^{r-1}+1}$. Then, using Equation (2) again, we deduce

$$h\left|u-\frac{1}{p}\right| < \left(p-\frac{1}{2}\right)\frac{p}{2^e} \leqslant \frac{2^r-\frac{1}{2}}{2^\ell-1}.$$

By means of Equation (4), we know that $|b-hu| < \frac{hu}{2^{\ell-1}}$ and therefore

$$|b-hu| < \frac{2^{\ell-2}\left(1+\frac{1}{2^{\ell-1}}\right)}{2^\ell-1}.$$

Summing the latter bounds we obtain:

$$|b-hu| + h\left|u-\frac{1}{p}\right| + \frac{|h-a|}{p} < \frac{2^{\ell-4}}{2^{\ell-3}+1} + \frac{2^{\ell-2}-\frac{1}{2}}{2^\ell-1} + \frac{2^{\ell-2}\left(1+\frac{1}{2^{\ell-1}}\right)}{2^\ell-1}.$$

Setting $L := 2^\ell$, the right hand-side rewrites

$$\frac{L/16}{L/8+1} + \frac{L/4-\frac{1}{2}}{L-1} + \frac{L/4\left(1+\frac{1}{L-1}\right)}{L-1},$$

and we claim that this quantity is less than 1 for all $L \geqslant 2^5$. In fact, this claim simply rewrites into $15/4L^2 - 10L + 4 > 0$, which is clearly satisfied. We thus have shown that $-1 \leqslant c - \lfloor \frac{a}{p} \rfloor \leqslant 1$. In particular, this implies $|a-cp| < 2p$, whence $|h-cp| \leqslant l+2p < 2^\ell$. This proves that $d = h-cp$ and, therefore, $g = d+l = h-cp+l = a-cp$, which finally implies the correctness of Function 3.6.

Now suppose that $\bar{u}$ is used and that the rounding mode is set towards infinity. Then $\bar{u} \geqslant 1/p$, $h \geqslant a$, and $b \geqslant h\bar{u}$ so $b \geqslant a\bar{u}$ and, therefore, $0 \leqslant c - \lfloor \frac{a}{p} \rfloor \leqslant 1$. □

Remark 3.8. Let `I` be a type of signed integers of at least $n > \ell$ bits. In the above functions, the executable code generated for the `floor` instruction heavily depends on the compiler, its version, and its command line arguments. This makes timings for this numeric approach rather unpredictable. We have run tests with GCC version $\geqslant$ 4.7 and CLANG [CLANG 2007] version $\geqslant$3.4 and observed that c = `(F) ((I) b)` always generates the x86 cast instructions `cvttsd2si`, `cvtsi2sd`, whereas c = `floor (b)` is compiled into a call to the `floor` function from the math library, which catches overflows. In order to force the compiler to use a special purpose instruction such as `roundsd` from SSE 4.1, the `-O3 -msse4.1 -fno-trapping-math` arguments must be passed to GCC. In the case of CLANG, the options `-O3 -msse4.1` are sufficient.

3.4. Vectorized Implementations

For efficiency, we assume that SSE 4.1 is available. Our modular addition and subtraction functions can benefit from `_mm_blendv_pd` as follows:

FUNCTION 3.9.
Input: Packed doubles $\vec{x}$ and $\vec{y}$ modulo $\vec{p}$, assuming $m \leqslant \ell - 1$.
Output: $(\vec{x}+\vec{y}) \operatorname{rem} \vec{p}$.

```
__m128d add_mod_51 (__m128d x, __m128d y) {
(1) __m128d a = _mm_add_pd (x, y);
(2) __m128d b = _mm_sub_pd (a, p);
(3) return _mm_blendv_pd (b, a, b); }
```

Assuming that FMA instructions are available, Function 3.6 is implemented as follows:

Table IX. Floating Point Operations in CPU Clock Cycles

Operation	sum		product	
Type	`float`	`double`	`float`	`double`
Scalar	1.1	1.0	1.1	1.0
SSE	0.32	0.65	0.32	0.65
AVX	0.19	0.37	0.19	0.37

Table X. Modular Operations in CPU Clock Cycles

Operation	sum		product				
Type	`float`	`double`	`float`		`double`		
m	21	50	11	21	25	26	50
Scalar	2.1	2.1	4.6	7.1	4.7	5.9	7.2
SSE & FMA	0.60	1.2	0.94	1.5	1.9	2.8	3.0
AVX & FMA	0.31	0.78	0.48	0.75	0.95	1.4	1.5

FUNCTION 3.10.
Input: Packed doubles $\vec{x}$ and $\vec{y}$ modulo $\vec{p}$, assuming $m \leqslant \ell - 2$.
Output: $(\vec{x}\vec{y}) \operatorname{rem} \vec{p}$.

```
__m128d mul_mod_50 (__m128d x, __m128d y) {
(1)  __m128d h = _mm_mul_pd (x, y);
(2)  __m128d l = _mm_fmsub_pd (x, y, h);
(3)  __m128d b = _mm_mul_pd (h, u);
(4)  __m128d c = _mm_floor (b);
(5)  __m128d d = _mm_fnmadd_pd (c, p, h);
(6)  __m128d g = _mm_add_pd (d, l);
(7)  __m128d t = _mm_sub_pd (g, p);
(8)  g = _mm_blendv_pd (t, g, t);
(9)  t = _mm_add_pd (g, p);
(10) return _mm_blendv_pd (g, t, g); }
```

Our AVX based implementation is the same, *mutatis mutandis*.

3.5. Timings

Table IX is obtained under similar conditions as Tables I and IV (i.e., using in-place operations on chunks of 2,048 bytes) but for types `float` and `double`. The row "Scalar" corresponds to the unvectorized implementation with loops unrolled eight by eight and preventing the compiler from using auto-vectorization. The next rows concern timings using SSE 4.1 and AVX instructions with loops unrolled four by four on aligned data. For instance, with AVX instructions, the product, applied simultaneously on four double precision numbers, perform two loads, one multiplication, and one save instruction, which amounts to a total throughput of 2. The observed average of 0.55 cycle per entry is close to this value, thanks to a suitable reordering of the instructions performed by the compiler in order to minimize costs due to latencies.

Table X shows timings for the modular sum and product functions from this section, in the most favorable case that benefits from rounding towards infinity. The row "Scalar" excludes auto-vectorization but does use the processor built-in FMA unit.

Compared to Tables II, V, and VI, using numerical types is the most efficient solution in general. This is especially so when operating on small amounts of data (that fit into the L3 cache, say) and for precisions of at least 31 bits. For lower precisions, the 30-bit Barrett's product and 31-bit Montgomery's product are nevertheless more efficient since they consume half of the memory. It may therefore be relevant to compare the number of cycles divided by m; for instance, performing several products by a fixed multiplicand costs $1.1/31 = 0.035$ per bit with Barrett's algorithm using AVX 2 (see Table VII), while we reach $1.5/50 = 0.03$ over `double` with FMA enabled. Since the cost per bit of modular sums is essentially halved when using 31-bit integers instead of 50-bits of `double`, the implementation of the FFT on 31-bit integers is expected to be a bit faster (per bits) than the one over 50 bits; this will we confirmed by our implementations in Section 5.

Remark 3.11. The total of the throughputs of the AVX and FMA instructions needed for the modular product with $m = 50$ in Function 3.10 is 6 (counting load and save instructions). Here the rounding mode is set towards infinity, so an average of 1.5 cycles is expected whenever all latencies vanish; this is well reflected by our timings.

Remark 3.12. As previously described in Remark 2.27, timings measured for the scalar algorithms are obtained with the `-fno-tree-vectorize` compiler option, which prevents the compiler from auto-vectorizing. In the scalar model, and for the typical case of Table X with $m = 50$, removing the latter compiler option and allowing AVX 2 with FMA instruction set leads to 6.7 cycles per modular product, which means that the modular product is not well vectorized automatically.

Remark 3.13. We also notice that efficient hardware quadruple precision arithmetic would allow us to consider moduli with larger bit sizes. An alternative to hardware quadruple precision arithmetic would be to provide an efficient "three-sum" $a + b + c$ instruction with correct IEEE-754 rounding. This would actually allow for the efficient implementation of more general medium precision floating point arithmetic.

4. IMPLEMENTATION DESIGN IN C++

The C++ libraries of MATHEMAGIX are not only oriented towards mathematical interfaces but also towards reusability of generic low-level implementations. Advanced users cannot only build on existing algorithms but also replace them *a posteriori* by more efficient implementations for certain special cases. We heavily rely on C++ template programming. Our techniques turn out to be especially useful for implementing the scalar and SIMD algorithms in a unified framework.

In this section, we review some of the basic design principles behind our implementation of modular arithmetic and several other basic objects such as vectors, polynomials, and matrices. We included it for making the algorithms in the next section more precise and easier to understand, as well as for those readers who wish to use the MATHEMAGIX libraries. The present design is original work that results from our experience and may turn out to be useful for developers. It is also important to stress that the sole design of efficient macros for modular arithmetic is not sufficient to take advantage of SIMD instructions in high-level algorithms for polynomials and matrices: One also needs to take care of memory alignment, loop unrolling, thresholds for short vectors, and so on.

4.1. Data Structures and Top Level Functions

Consider a typical template class in MATHEMAGIX, such as `modulus`, `vector`, `matrix`, `polynomial`, or `series`. Besides the usual template parameters, such as a coefficient type `C`, such classes generally come with a special additional parameter `V`, called the *implementation variant*. The parameter `V` plays a similar role as *traits* classes in usual

C++ terminology [Abrahams and Gurtovoy 2004, Chapter 2]. The variant `V` does *not* impact the internal representation of instances of the class, but it does control the way in which basic functions manipulate such instances.

For example, the class `vector<C,V>` (defined in `basix/vector.hpp`) corresponds to dense vectors with entries in `C`, stored in a contiguous segment of memory. A vector of this type consists of a pointer of type `C*` to its first entry, of its size n, and of the size l of the allocated memory. For the sake of simplicity we omit that our vectors are endowed with reference counters. The type `nat` is a `typedef` to a native unsigned integer type that can be set at configuration time and which is uniformly used to represent sizes of allocated memory. At the top level user interface, for instance, the sum of two vectors is defined as follows:

```
template<typename C, typename V>
vector<C,V> operator + (const vector<C,V>& v, const vector<C,V>& w) {
  typedef implementation<vector_linear, V> Vec;
  nat n= N(v); nat l= aligned_size<C,V> (n);
  C* t= mmx_new<C> (l);
  Vec::add (t, seg (v), seg (w), n);
  return vector<C,V> (t, n, l); }
```

In this piece of code, `N(v)` represents the size of `v`, and `aligned_size<C,V> (n)` computes the length to be allocated in order to store vectors of size `n` over `C` in memory. According to the values of `C` and `V`, we can force the allocated memory segment to start at an address multiple of a given value, such as 16 bytes, when using SSE instructions. The function `mmx_new<C>` is a reimplementation in MATHEMAGIX of the usual `new` function, which allows faster allocation of objects of small sizes. Finally, the data `t`, `n`, `l` are stored into an instance of `vector<C,V>`. The allocated memory is released once the vector is not used anymore, that is, when its reference counter becomes zero. The core of the computations is implemented in the static member function `add` of the class `implementation<vector_linear,V>` explained in the next paragraphs. At the implementation level, operations are usually performed efficiently on the object representations. The expression `seg (v)` returns the first address of type `C*` of the memory segment occupied by `v`.

4.2. Algorithms and Implementations

Classes containing implementations are specializations of the following:

```
template<typename F, typename V, typename W=V> struct implementation;
```

The first template argument `F` is usually an empty class that names a set of *functionalities*. In the latter example, we introduced `vector_linear`, which stands for the usual entrywise vector operations, including addition, subtraction, product, and so on. The value of the argument `W` for *naive implementations* of vector operations is `vector_naive`. The role of the second argument `V` will be explained later. The naive implementation of the addition of two vectors is then declared as a static member of `implementation<vector_linear, V, vector_naive>` as follows:

```
template<typename V>
struct implementation<vector_linear, V, vector_naive> {
  static inline void
  add (C* dest, const C* s1, const C* s2, nat n) {
    for (nat i= 0; i < n; i++)
      dest[i]= s1[i] + s2[i]; }                                    ../..
```

Four by four *loop unrolling* can, for instance, be implemented within another variant, say, `vector_unrolled_4`, as follows:

```
template<typename V>
struct implementation<vector_linear, V, vector_unrolled_4> {
  static inline void
  add (C* dest, const C* s1, const C* s2, nat n) {
    nat i= 0;
    for (; i + 4 < n; i += 4) {
      dest[i  ]= s1[i  ] + s2[i  ]; dest[i+1]= s1[i+1] + s2[i+1];
      dest[i+2]= s1[i+2] + s2[i+2]; dest[i+3]= s1[i+3] + s2[i+3]; }
    for (; i < n; i++)
      dest[i]= s1[i] + s2[i]; }                                    ../..
```

When defining a new variant, we hardly ever want to redefine the whole set of functionalities of other variants. Instead, we wish to introduce new algorithms for a subset of functionalities and to have the remaining implementations inherit from other variants. We say that a variant `V` *inherits* from `W` when the following partial specialization is active:

```
template<typename F, typename U>
struct implementation<F,U,V>: implementation<F,U,W> {};
```

For instance, if the variant `V` inherits from `vector_naive`, then the `add` function from `implementation<vector_linear,V>` is inherited from `implementation<vector_linear, vector_naive>`, unless the following partial template specialization is implemented: `template<typename U> implementation<vector_linear,U,V>`.

It remains to explain the role of the three parameters of `implementation<F,U,V>`. Actual implementations of a variant usually involve a lot inheritance. The parameter `U` keeps track of the original top level variant, whereas `V` corresponds to the actual variant that we are implementing. Therefore, in a static member of `implementation<F,U,V>`, when one needs to call a function related to another set of functionalities `G`, then it is fetched in `implementation<G,U>`.

For vectors, several variants are available in MATHEMAGIX: naive loop-based implementations (`basix/vector_naive.hpp`), improved support for plain old data types (to avoid constructors and destructors for each entry), vectors of fixed size (`algebramix/vector_fixed.hpp`), unrolled loops (`algebramix/vector_unrolled.hpp`), memory alignment contraints (`algebramix/vector_aligned.hpp`), and automatic use of SIMD instructions (`algebramix/vector_simd.hpp`).

Let us illustrate the interest of our approach on the hand of polynomials. It is a very standard situation in computer algebra to design and analyze algorithms for complex operations such as division, gcd, multi-point evaluation, and so on, in terms of a function that computes the product (this is typically done so in the book [von zur Gathen and Gerhard 2003]). This point of view is also useful in practice, especially when implementing asymptotically fast algorithms.

Now our class `polynomial<C,V>` (defined in `algebramix/polynomial.hpp`) represents polynomials with coefficients in `C` using implementation variant `V`. Each instance of a polynomial is represented by a vector, that is, a pointer with a reference counter to a structure containing a pointer to the array of coefficients of type `C*` with its allocated size, and an integer for the length of the considered polynomial (defined as the degree plus one). The set of functionalities includes linear operations, mainly inherited from those of the vectors (since the internal representations are the same),

multiplication, division, composition, Euclidean sequences, subresultants, evaluations, Chinese remaindering, and so on. All these operations are implemented for the variant `polynomial_naive` (in the file `algebramix/polynomial_naive.hpp`) with the most general but naive algorithms (with quadratic cost for multiplication and division).

The variant `polynomial_dicho<W>` inherits from the parameter variant `W` and contains implementations of classical divide-and-conquer algorithms: Sieveking's polynomial division, half-gcd, and divide-and-conquer evaluation and interpolations [von zur Gathen and Gerhard 2003, Chapters 8–11]. Polynomial product functions belong to the set of functionalities `polynomial_multiply` and division functions to the set `polynomial_divide`. The division functions of `polynomial_dicho<W>` are implemented as static members of

```
template<typename U, typename W>
struct implementation<polynomial_divide,U,polynomial_dicho<W> >
```

They make use of the product functions of `implementation<polynomial_multiply,U>`.

Let us now consider polynomials with modular integer coefficients and let us describe how the Kronecker substitution product is plugged in. In a nutshell, the coefficients of the polynomials are first lifted into integers. The lifted integer polynomials are next evaluated at a sufficiently large power of two, and we compute the product of the resulting integers. The polynomial product is retrieved by splitting the integer product into chunks and reducing them by the modulus. For details, we refer the reader, for instance, to the book of von zur Gathen and Gerhard [2003, Chapter 8]. As to our implementation, we first create the new variant `polynomial_kronecker<W>` on top of another variant `W` (see file `algebramix/polynomial_kronecker.hpp`), which inherits from `W` but which only redefines the implementation of the product in

```
template<typename U, typename W>
struct implementation<polynomial_multiply, U, polynomial_kronecker<W> >
```

When using the variant `K` defined by

```
typedef polynomial_dicho<polynomial_kronecker<polynomial_naive> > > K;
```

the product functions in `implementation<polynomial_multiply, K>` correspond to the Kronecker substitution. The variant `polynomial_dicho` does not redefine the product but contains divide-and-conquer implementations for other operations such as the division, gcd, and so on. The functions in `implementation<polynomial_divide, K>` are finally inherited from

```
implementation<polynomial_divide, K,
               polynomial_dicho<polynomial_naive> >
```

and thus use Sieveking's division algorithm. The divisions rely in turn on the multiplication from `implementation<polynomial_multiply, K>`, which benefits from Kronecker substitution. In this way, it is very easy for a user to redefine a new variant and override given functions *a posteriori*, without modifying existing code. In particular, we derive many variants for concrete coefficient types in this way (`polynomial_int`, `polynomial_integer`, `polynomial_polynomial`, etc.). This approach also turns out to be well suited to numerical coefficient types, which rely on exact types, as explained by Lecerf [2010].

Finally, for a given mathematical template type, we define a *default variant* as a function of the remaining template parameters. For instance, the default variant of the parameter `V` in `vector<C,V>` is `typename vector_variant_helper<C>::VV` which is intended to provide users with reasonable performance. This default value is set to

vector_naive, but can be overwritten for special coefficients such as modular integers. The default variant is also the default value of the variant parameter in the declaration of vector. Users can thus simply write vector<C>. The same mechanism applies to polynomials, series, matrices, and so on.

4.3. Modular Integers

In MATHEMAGIX, moduli are stored in the dedicated class modulus<C,V> (from numerix/modulus.hpp). This abstraction allows us to attach extra information to the modulus, such as a pre-inverse. Modular numbers are instances of modular<M,W> (from numerix/modular.hpp), where M is a modulus type and W is a variant specifying the way the modulus is stored (e.g., in a global variable or as an additional field for each modular number). We also provide a variant for static constant moduli whose pre-inverses are computed at compilation time.

Scalar arithmetic over integer (respectively, numeric) types are implemented in numerix/modular_int.hpp (respectively, numerix/modular_double.hpp) and include variants for the aforementioned algorithms (modulus_int_preinverse, modulus_int_montgomery, etc.). In all cases, the bit-size bound m for the modulus is a template parameter. For convenience, packed integer and numeric types are wrapped into C++ classes so modular operations can easily be implemented for them (see numerix/sse.hpp and numerix/avx.hpp and files where modular arithmetic is derived). The variant mechanism is also useful to implement arithmetic dedicated to specific moduli, such as $2^r - 1$ or $2^{r-1} + 1$.

Intensive tests are useful to make sure that all cases are correctly implemented and compiled for all bit sizes, with random primes, with all powers of two, with $2^r - 1$, and also with the special case when $p = 0$. All these tests are implemented in our numerix/test directory for all the possible variants. Of course, modular arithmetic is also used in our higher level C++ libraries (such as MULTIMIX, GEOMSOLVEX, and FACTORIX), which further increases the reliability of our implementations of most common variants.

Let us mention that vectors of modular numbers have a specific implementation variant allowing us to share the same modulus when performing entrywise operations (algebramix/vector_modular.hpp). In this way, we avoid fetching the modulus for each arithmetic operation. Operations on vectors over integer and numeric types are implemented in a hierarchy of variants. One major variant controls the way loops are unrolled. Loops must be unrolled enough in order to let the compiler reorganize sufficiently large blocks of SIMD instructions to decrease the total latency. Unrolling must not be too high because the compiler may then skip the optimization. In fact, the number of unrolled steps is often limited by the number of SIMD registers. Meta-template programing is intensively used to make the implementation easily extensible; we plan to support AVX-512 in the near future.

5. FAST FOURIER TRANSFORM AND APPLICATIONS

In order to benefit from vectorized modular arithmetic within the fast Fourier transform, we implemented a vectorized variant of the classical in-place algorithm. In this section, we describe this variant and its applications to polynomial and matrix multiplication.

5.1. Vectorized Truncated Fourier Transform

Let K be a commutative field, let $n = 2^k$ with $k \in \mathbb{N}$, and let $\omega \in K$ be a primitive n-th root of unity, which means that $\omega^n = 1$, and $\omega^j \neq 1$ for all $j \in \{1, \ldots, n-1\}$. Let $\mathcal{A}$ be a K-vector space. The fast Fourier transform (with respect to ω) of an n-tuple

ALGORITHM 1: In-place Vectorized Truncated Fourier Transform

Input: $n = 2^k$, ω a n-th primitive root of unity, $l \leqslant n$, $(a_0, \ldots, a_{l-1}) \in K^l$, and an integer $n_1 = 2^{k_1} \leqslant l$. Let $n_2 = 2^{k_2} = n/n_1$.
Output: $A(\omega^{[i]_k})$ for all $i \in \{0, \ldots, (\lambda-1)n_1 - 1\}$, where $\lambda := \lceil l/n_1 \rceil$, and $A(X) = a_0 + a_1 X + \cdots + a_{l-1}X^{l-1}$.
For convenience we consider that $a_i = 0$ for all $i \in \{l, \ldots, n\}$.

(1) Let $b_{j_2} := (a_{j_2 n_1}, \ldots, a_{(j_2+1)n_1-1})$ for all $j_2 \in \{0, \ldots, \lambda - 1\}$.
Compute $\left(\hat{b}_{[0]_{k_2}}, \ldots, \hat{b}_{[\lambda-1]_{k_2}}\right) := \mathrm{TFT}_{\omega^{n_1}}(b_0, \ldots, b_{\lambda-1})$.
(2) Let $c_{j_1} := \left(\hat{b}_{[0]_{k_2},j_1}\omega^{j_1[0]_{k_2}}, \hat{b}_{[1]_{k_2},j_1}\omega^{j_1[1]_{k_2}}, \ldots, \hat{b}_{[\lambda-1]_{k_2},j_1}\omega^{j_1[\lambda-1]_{k_2}}\right)$ for all $j_1 \in \{0, \ldots, n_1 - 1\}$.
(3) Compute $\left(\hat{c}_{[0]_{k_1}}, \ldots, \hat{c}_{[n_1-1]_{k_1}}\right) := \mathrm{TFT}_{\omega^{n_2}}(c_0, \ldots, c_{n_1-1})$.
(4) Return $\left(\hat{c}_{[0]_{k_1},0}, \ldots, \hat{c}_{[n_1-1]_{k_1},0}, \hat{c}_{[0]_{k_1},1}, \ldots, \hat{c}_{[n_1-1]_{k_1},1}, \ldots, \hat{c}_{[0]_{k_1},\lambda-1}, \ldots, \hat{c}_{[n_1-1]_{k_1},\lambda-1}\right)$.

$(a_0, \ldots, a_{n-1}) \in \mathcal{A}^n$ is the n-tuple $(\hat{a}_0, \ldots, \hat{a}_{n-1}) =: \mathrm{FFT}_\omega(a) \in \mathcal{A}^n$ with

$$\hat{a}_i = \sum_{j=0}^{n-1} \omega^{ij} a_j.$$

In other words, $\hat{a}_i = A(\omega^i)$, where $A \in \mathcal{A}\otimes K[X]$ denotes the element $A(X) := \sum_{i=0}^{n-1} a_i \otimes X^i$. If $i \in \{0, \ldots, n-1\}$ has binary expansion $i_0 + i_1 2 + i_2 2^2 + \cdots + i_{k-1}2^{k-1}$, then we write $[i]_k = i_{k-1} + i_{k-2}2 + i_{k-3}2^2 + \cdots + i_0 2^{k-1}$ for the *bitwise mirror* of i in length k. Following the terminology of van der Hoeven [2004], the *truncated Fourier transform* (TFT) of an l-tuple $(a_0, \ldots, a_{l-1}) \in \mathcal{A}^l$ (with respect to ω) is

$$\mathrm{TFT}_\omega(a) := \left(\hat{a}_{[0]_k}, \ldots, \hat{a}_{[l-1]_k}\right) = \left(A\left(\omega^{[0]_k}\right), \ldots, A\left(\omega^{[l-1]_k}\right)\right).$$

Usually we have $n/2 < l \leqslant n$. Roughly speaking, the TFT computes only the "relevant part" of an FFT, exploiting the fact that the $n - l$ last values $A(\omega^{[l]_k}), \ldots, A(\omega^{[n-1]_k})$ are not needed. In terms of branchings the TFT is more complicated, which makes it difficult to optimize for small sizes with entries of a few bytes. The original FFT and TFT algorithms (see articles by van der Hoeven [2004] and Harvey and Roche [2010], for instance) do not directly exploit low-level vectorization. Specific adaptations are necessary and have already been designed for high-performance software such as FFTW3 and SPIRAL [Frigo and Johnson 2005; McFarlin et al. 2011; Meng and Johnson 2014].

Our strategy is quite rough but fine for large sizes and easy to program. Let $n_1 = 2^{k_1} < l$ be a divisor of n. Algorithm 1 reduces one TFT of size l over K into one TFT over K^{n_1} of size $\lambda := \lceil l/n_1 \rceil$ and into one TFT of size n_1 over K^λ.

PROPOSITION 5.1. *Algorithm 1 is correct and takes at most $\frac{3}{2}\lambda n_1 k + O(n)$ operations in K, assuming given all the powers of ω.*

PROOF. Let $B_i(X) = b_{0,i} + b_{1,i}X + \cdots + b_{\lambda-1,i}X^{\lambda-1}$ for $i \in \{0, \ldots, n_1 - 1\}$, so we have $A(X) = B_0(X^{n_1}) + B_1(X^{n_1})X + \cdots + B_{n_1-1}(X^{n_1})X^{n_1-1}$ and $\hat{b}_{[j_2]_{k_2}} = (B_0(\omega^{n_1[j_2]_{k_2}}), \ldots, B_{n_1-1}(\omega^{n_1[j_2]_{k_2}}))$ for all $j_2 \in \{0, \ldots, \lambda-1\}$. Let $j_1 \in \{0, \ldots, n_1-1\}$ and $j_2 \in \{0, \ldots, \lambda - 1\}$. A straightforward calculation leads to $A(\omega^{[j_2 n_1 + j_1]_k}) = A(\omega^{[j_1]_{k_1} n_2 + [j_2]_{k_2}}) =$

$$\begin{aligned}
&\hat{b}_{[j_2]_{k_2},0} + \hat{b}_{[j_2]_{k_2},1}\omega^{[j_1]_{k_1}n_2 + [j_2]_{k_2}} + \cdots + \hat{b}_{[j_2]_{k_2},n_1-1}\omega^{([j_1]_{k_1}n_2 + [j_2]_{k_2})(n_1-1)} \\
&= \hat{b}_{[j_2]_{k_2},0} + \left(\hat{b}_{[j_2]_{k_2},1}\omega^{[j_2]_{k_2}}\right)\omega^{[j_1]_{k_1}n_2} + \cdots + \left(\hat{b}_{[j_2]_{k_2},n_1-1}\omega^{[j_2]_{k_2}(n_1-1)}\right)\omega^{[j_1]_{k_1}n_2(n_1-1)} \\
&= c_{0,j_2} + c_{1,j_2}\omega^{[j_1]_{k_1}n_2} + \cdots + c_{n_1-1,j_2}\omega^{[j_1]_{k_1}n_2(n_1-1)} = \hat{c}_{[j_1]_{k_1},j_2}.
\end{aligned}$$

Table XI. FFT of Size n Over $\mathbb{Z}/469762049\mathbb{Z}$; User Time in Microseconds

n	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Scalar	2.8	6.3	14	31	69	150	320	670	1400	3200	6700	14000	30000
SSE	1.0	2.2	4.8	11	22	48	110	220	470	1100	2300	4900	11000
AVX	0.64	1.4	2.9	6.2	13	29	65	140	290	640	1400	2900	6800

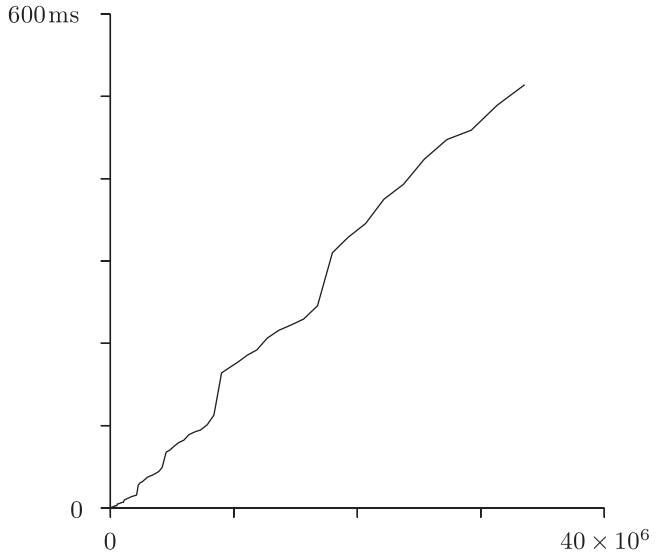

Fig. 1. TFT of size n over $\mathbb{Z}/469762049\mathbb{Z}$ using AVX 2; user time in milliseconds.

By van der Hoeven [2004, Theorem 1], step 1 can be done with $\frac{3}{2}n_1(\lambda k_2 + n_2) + n_1/2$ operations in K. Step 2 involves λn_1 operations, and step 3 takes $\frac{3}{2}\lambda(n_1 k_1 + n_1) + n_2/2$ more operations.

Inverting Algorithm 1 is straightforward: It suffices to invert steps from 4 to 1 and use the inverse of the TFT. If $l = n$, then Algorithm 1 can be used to compute the FFT. If n_1 is taken to be the size corresponding to a machine vector, then most of the TFT can be performed using SIMD operations. For sizes of order at most a few kilobytes, we actually use this strategy. In larger sizes, it is preferable to take n_1 much larger, for instance, of order $\sqrt{n}$, so the TFT runs on rather large vectors. With n_1 of order $\sqrt{n}$, this algorithm is very close to the cache-friendly version of the TFT designed by Harvey [2009].

A critical operation for large sizes is matrix transposition, necessary to reorganize data in steps 1 and 4 of Algorithm 1. We designed *ad hoc* cache-friendly SIMD versions for it. Table XI reports on timings obtained in this way for $K = \mathbb{Z}/p\mathbb{Z}$ with $p = 469762049 = 7 \times 2^{26} + 1$ in MATHEMAGIX, with Barrett's approach (see `algebramix/bench/fft_bench.cpp`, compiled successively with options `-fno-tree-vectorized`, then with `-msse4.2` and `-mavx2`). In particular, the row "Scalar" has been obtained by preventing the compiler from auto-vectorizing. All the necessary primitive roots and fixed multiplicands (including the *twiddle factors*) are pre-computed once and cached in memory. We observe significant speedups for the SSE and AVX versions. In Figure 1, we draw the graph of the timings of our vectorized TFT: It shows how the gaps due to FFT padding are softened (see `algebramix/bench/tft_bench.cpp`).

Table XII. FFT of Size n Over $\mathbb{Z}/1108307720798209\mathbb{Z}$; User Time in Microseconds

n	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
NTL	2.2	4.5	9.7	21	51	100	230	480	1000	2200	4900	11000	25000
`int64_t`	3.2	7.0	16	36	77	160	360	770	1600	3500	7400	16000	34000
`double`	5.1	11	26	57	120	270	590	1300	2700	5800	12000	26000	55000
AVX/FMA	1.2	2.5	5.6	13	27	59	130	270	590	1200	2600	5600	14000

Table XIII. FFT for Complex of `double` of Size n; User Time in Microseconds

n	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
FFTW3	0.42	0.94	2.3	5.6	14	35	84	190	400	880	2500	8000	19000
Scalar	1.9	4.4	10	23	52	110	250	560	1100	2500	5800	14000	31000
AVX/FMA	0.50	1.1	2.3	6.0	14	33	79	200	410	920	2100	6000	15000

Table XII concerns FFTs for $K = \mathbb{Z}/p\mathbb{Z}$ and $p = 1108307720798209 = 63 \times 2^{44} + 1$. We compare NTL 9.1.0 (in the first row) with our FFT implementations. The row "`int64_t`" refers to the unvectorized FFT on 64-bit modular integers using Barrett's product. Although timings are about 50% slower than NTL, the timing ratio is not very far from the bit-size ratio limit 50/63. The row "`double`" corresponds to using our unvectorized modular product of Function 3.6, while the last row is vectorized with AVX and FMA instructions. In all cases, we measure average running time, so the cost for computing root tables is discarded. NTL performs additional "bit-reverse copies," whereas MATHEMAGIX does not. Nevertheless, NTL benefits from the "improved lazy multiplication" strategy of Harvey [2014]. We have not implemented this strategy yet but expect it to yield a significant speedup in our implementations. Unfortunately, since NTL modular product uses 64-bit low products, it cannot be easily vectorized with SSE or AVX technologies. Intensive tests for all variants of FFT and TFT are implemented in our `algebramix/test` directory.

For the sake of comparison, we also report on the performance of the FFT over complex numbers in double precision. Table XIII compares timings provided by the command `test/bench` bundled with FFTW version 3.3.4 [Frigo and Johnson 2005] (configured with the `--enable-avx` option) to the MATHEMAGIX implementation in `algebramix/fft_split_radix.hpp` (see also `algebramix/bench/fft_complex_bench.cpp`). In small sizes, the vectorization ratio can be observed. In large sizes, the ratio changes because the cost of memory management relatively increases, although we use the six-step algorithm. We wish to mention that vectorized implementations of the FFT in higher medium precisions are also available in MATHEMAGIX [van der Hoeven and Lecerf 2015].

5.2. Polynomial Matrix Product

One major application of the TFT is the computation of polynomial products. In Table XIV, we provide timings for multiplying two polynomials of degrees $< d$ over $\mathbb{Z}/469762049\mathbb{Z}$ (see `algebramix/bench/polynomial_bench.cpp`). The last two rows concern the scalar and AVX 2 versions of our modular TFT. Here an important fact must be noticed. Previous timings of TFT were obtained as average on several runs on the same input. An overhead thus applies when performing a polynomial product where two direct TFT and one inverse TFT have to be performed. Taking also into account zero padding and the entrywise vector product in between, the total time sensibly exceeds 3 times the one displayed in Table XI.

Table XIV. Polynomial Product for Degrees $< d$ Over $\mathbb{Z}/469762049\mathbb{Z}$; User Time in Milliseconds

d	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
FLINT	0.017	0.043	0.11	0.29	0.71	1.8	4.6	11	22	48	110	290	610
NTL	0.017	0.035	0.076	0.18	0.36	0.81	1.7	3.6	7.9	17	38	89	200
Kronecker	0.032	0.078	0.19	0.47	1.2	2.6	5.9	12	30	60	140	310	680
Scalar	0.023	0.049	0.11	0.22	0.49	1.0	2.3	4.9	10	21	46	97	200
AVX	0.0062	0.012	0.024	0.050	0.11	0.23	0.47	1.1	2.2	4.8	10	24	55

Table XV. Polynomial Matrix Product over $\mathbb{Z}/469762049\mathbb{Z}$ for Degrees $d < 2^{15}$; User Time in Seconds

n	1	2	4	8	16	32
FLINT	0.011	0.086	0.70	5.6	45	360
`matrix_naive`, AVX	0.0011	0.0096	0.078	0.62	5.0	37
`matrix_tft`, scalar	0.005	0.023	0.10	0.48	2.5	15
`matrix_tft`, AVX	0.0011	0.0059	0.026	0.12	0.58	3.2

For information, we also added timings obtained with our Kronecker substitution implementation [von zur Gathen and Gerhard 2003, Section 8.4] (that reduces the product to one integer product, for which we appeal to GMP version 6.0.0a [Granlund et al. 1991]), with FLINT 2.4.4 [Hart and the FLINT Team 2008], and with NTL. For a fair comparison against NTL, we use the function `UserFFTInit` to ensure that the product directly uses the FFT. Since FLINT makes use of the Kronecker substitution for large sizes, it does not benefit from FFT primes. Therefore, the comparison is not quite fair, but it illustrates the interest of using FFT primes whenever possible. In fact, in some applications over rational numbers such as sparse interpolation, polynomial factorization, or polynomial system solving, we often can choose prime moduli. FFT primes thus turn out to be very worthy.

If K is a field with sufficiently many primitive roots of unity of order a power of two, then two $n \times n$ matrices A and B with coefficients in $K[x]$ of degrees $<d$ can be multiplied by performing TFT transforms of length $2d$ on each of the coefficients of A and B, by multiplying $2d$ matrices over K, and, finally, by recovering the matrix product through coefficient-wise inverse transforms. This requires $O(dn^\omega + n^2 d \log d)$ operations in K, where $\omega \leqslant 3$ is the exponent of $n \times n$ matrix multiplication over K. In general, most of the time is spent in the matrix products over K. Nevertheless, if n remains sufficiently small with respect to d, then most of the time is spent in the fast Fourier transforms, and this method becomes most efficient.

Table XV compares this approach (see the rows "`matrix_tft`" for both vectorized and unvectorized versions and the source code in `algebramix/matrix_tft.hpp` and `algebramix/bench/matrix_tft_bench.cpp`) to naive multiplication (see the row "`matrix_naive`"). The FFT polynomial products and sums are performed following the naive matrix product formulas. We completed the table with timings obtained with the function `nmod_poly_mat_mul` of FLINT, which automatically chooses the best-available algorithm.

5.3. Integer Matrix Product

Another important application of fast Fourier transforms is integer multiplication. The implemented method is based on *Kronecker segmentation* and *three-prime FFT* (see Pollard [1971] and von zur Gathen and Gerhard [2003, Section 8.3]). Let p_1, p_2, and p_3

Table XVI. Integer Product in Bit-Size $32 \times n$; User Time in Milliseconds

n	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
GMP	0.0060	0.017	0.045	0.12	0.32	0.81	1.9	4.2	9.6	21	49	110	240
MATHEMAGIX	0.031	0.057	0.11	0.23	0.48	1.0	2.1	4.9	10	20	45	91	190

Table XVII. Integer Matrix Product in Bit-size 32×2^{15}; User Time in Seconds

n	1	2	4	8	16	32
LINBOX	0.0042	0.033	0.27	2.2	17	140
FLINT	0.0042	0.036	0.29	2.3	18	150
`matrix_naive`	0.0042	0.033	0.27	2.2	17	140
`matrix_integer`	0.0047	0.019	0.080	0.36	1.8	10

be three prime numbers. The two integers a and b of at most N bits to be multiplied are split into chunks of a suitable bit-size h and converted into polynomials A and B of $\mathbb{Z}[X]$ of degrees $<d$, with $A(2^h) = a$, $B(2^h) = b$ and $d = \lceil N/h \rceil$. The maximum bit-size of the coefficients of the product $C(X) = A(X)B(X)$ is at most $H = 2h + \lceil \log d / \log 2 \rceil$. The parameter h is taken such that d is minimal under the constraint that $2^H < p_1 p_2 p_3$. The polynomial C can then be recovered from its values computed modulo p_1, p_2, and p_3 using TFT multiplications. The row "MATHEMAGIX" of Table XVI shows the performance obtained by this approach for $p_1 = 998244353$, $p_2 = 985661441$, and $p_3 = 943718401$. Table XVI also provides timings for GMP. Notice that GMP seems to make good use of SIMD instructions, even if the nature of its internal algorithms makes these tasks difficult. Our implementations can be found in `algebramix/fft_int.hpp` and `algebramix/bench/fft_int_bench.cpp`. Of course, without the AVX technology, our timings are obviously not competitive, and we discard them here.

Similarly to polynomial matrices, small matrices over large integers can be multiplied efficiently using modular TFTs. In Table XVII, the row "`matrix_integer`" shows timings for this approach (see `algebramix/matrix_integer.hpp`). The row "`matrix_naive`" corresponds to the classical product calling GMP functions directly. We also compare to FLINT and LINBOX version 1.3.2 [Dumas et al. 2010] (using NTL and FFLAS-FFPACK 1.6.0 [Dumas et al. 2008, 2004]).

6. CONCLUSION

Nowadays, SIMD technology has widely proved to be extremely efficient for numerical computations. A major conclusion of the present work is that modular integers, and therefore exact and symbolic computations, can also greatly benefit from this technology via integer types and even in small sizes. In particular, 30-bit modular integers may be used in very competitive applications such as large integer multiplication thanks to the AVX 2 technology. We believe that the dissemination of the recent AVX-512 and other forthcoming extensions with wider vectors will be very profitable.

Of course, the tradeoff between integer and floating point operations is highly dependent on specificities of the processor. In MATHEMAGIX, we have chosen to implement the best currently available algorithms for both integer and floating point types. We hope that this will also allow other researchers to make fair comparisons between both approaches.

The use of C++ as a programming language allowed us to develop generic template libraries inside MATHEMAGIX, which are both efficient and uniform. Nevertheless, ensuring good performance for the implementations presented here was not an easy task. Depending on the presence or absence of specific SIMD instructions, we had to

implement a large number of variants for the same algorithms. Template programming turns out to be essential to control loop unrolling and memory alignment. Furthermore, the efficiency of several elementary routines for small fixed sizes could only be ensured through hand optimization of very specific pieces of code. For instance, several FFTs in small fixed sizes, say, the size of SIMD registers, cannot be easily vectorized optimally. Here hardware transposition routines would be helpful, for example for 8×8 matrices of 32-bit integers using AVX 2 registers. In the longer run, we expect that part of the work that we did by hand could be done by an optimizing compiler. One major challenge for compilers and the design of programming languages is the automatic generation of FFTW3-style codelets. This would also be useful for tuning arithmetic operations in small and medium finite fields, in the same vein as the MPFQ library [Gaudry and Thomé 2007].

As a final comment, we would like to emphasize that SIMD technology can be seen as a low-level way to parallelize computations. We believe that bringing easy access to this technology in MATHEMAGIX will be fruitful to nonspecialist computer algebra developers. Currently, such developers are faced with monolithic software such as MAPLE [Geddes et al. 1980] or MAGMA [Bosma et al. 1997], which very well cover low and high user-level mathematical functions but lack general lower-level programming facilities. Some recent C++ and PYTHON projects [Rossum and Boer 1991] aim at providing frameworks for large-scale scientific programming, such as those by Stein et al. [2004] and the Boost team [1999], but all gaps have not been filled yet. One example of an interesting practical challenge is the multiplication of dense matrices with large integer coefficients. Indeed, this was one of the programming contests of the PASCO 2010 conference. Our benchmarks in Section 5.3 show that there was indeed room for practical progress. For recent theoretical advances on this topic, we refer the interested reader to the article of Harvey and van der Hoeven [2014].

ACKNOWLEDGMENTS

We thank David Harvey and Victor Shoup for their useful comments. We are also grateful to the referees for their valuable corrections and suggestions.

REFERENCES

D. Abrahams and A. Gurtovoy. 2004. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison Wesley.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

R. Alverson. 1991. Integer division using reciprocals. In *Proceedings of the Tenth Symposium on Computer Arithmetic*. IEEE Computer Society Press, 186–190.

H. G. Baker. 1992. Computing A*B (mod N) efficiently in ANSI C. *SIGPLAN Not.* 27, 1 (1992), 95–98.

B. Bank, M. Giusti, J. Heintz, G. Lecerf, G. Matera, and P. Solernó. 2015. Degeneracy loci and polynomial equation solving. *Found. Comput. Math.* 15, 1 (2015), 159–184.

N. Bardis, A. Drigas, A. Markovskyy, and J. Vrettaros. 2010. Accelerated modular multiplication algorithm of large word length numbers with a fixed module. In *Organizational, Business, and Technological Aspects of the Knowledge Society*, M. D. Lytras, P. Ordonez de Pablos, A. Ziderman, A. Roulstone, H. Maurer, and J. B. Imber (Eds.). Communications in Computer and Information Science, Vol. 112. Springer, Berlin, 497–505.

P. Barrett. 1987. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology – CRYPTO 86*, A. Odlyzko (Ed.). Lect. Notes Comput. Sci., Vol. 263. Springer, Berlin, 311–323.

D. J. Bernstein, Hsueh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Cing Lin, and Bo-Yin Yang. 2009a. The billion-mulmod-per-second PC. In *SHARCS09 Special-purpose Hardware for Attacking Cryptographic Systems: 131*. 131–144. http://cr.yp.to/djb.html.

D. J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. 2009b. ECM on graphics cards. In *Advances in Cryptology - EUROCRYPT 2009*, A. Joux (Ed.). Lect. Notes Comput. Sci., Vol. 5479. Springer, Berlin, 483–501.

J. Berthomieu, G. Lecerf, and G. Quintin. 2013. Polynomial root finding over local rings and application to error correcting codes. *Appl. Alg. Eng. Comm. Comp.* 24, 6 (2013), 413–443.

J. Berthomieu, J. van der Hoeven, and G. Lecerf. 2011. Relaxed algorithms for p-adic numbers. *J. Théor. Nombr. Bord.* 23, 3 (2011), 541–577.

D. Bini and V. Y. Pan. 2012. *Polynomial and Matrix Computations: Fundamental Algorithms*. Birkhauser Verlag GmbH.

Boost team. From 1999. Boost (C++ libraries). Software available at http://www.boost.org. (From 1999).

W. Bosma, J. Cannon, and C. Playoust. 1997. The Magma algebra system. I. The user language. *J. Symbol. Comput.* 24, 3–4 (1997), 235–265.

A. Bosselaers, R. Govaerts, and J. Vandewalle. 1994. Comparison of three modular reduction functions. In *Advances in Cryptology CRYPTO 93*, D. R. Stinson (Ed.). Lect. Notes Comput. Sci., Vol. 773. Springer, Berlin, 175–186.

British Standards Institution. 2003. *The C Standard: Incorporating Technical Corrigendum 1: BS ISO/IEC 9899/1999*. John Wiley.

CLANG From 2007. CLANG, a C language family frontend for LLVM. Software available at http://clang.llvm.org. (From 2007).

J.-G. Dumas, T. Gautier, C. Pernet, and B. D. Saunders. 2010. LinBox founding scope allocation, parallel building blocks, and separate compilation. In *Mathematical Software ICMS 2010*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama (Eds.). Lect. Notes Comput. Sci., Vol. 6327. Springer, Berlin, 77–83.

J.-G. Dumas, P. Giorgi, and C. Pernet. 2004. FFPACK: Finite field linear algebra package. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation (ISSAC'04)*, J. Schicho (Ed.). ACM, 119–126.

J.-G. Dumas, P. Giorgi, and C. Pernet. 2008. Dense linear algebra over word-size prime fields: The FFLAS and FFPACK packages. *ACM Trans. Math. Softw.* 35, 3 (2008), 19:1–19:42.

A. Fog. 2012a. *Instruction Tables. Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs*. http://www.agner.org/optimize, Copenhagen University College of Engineering.

A. Fog. 2012b. *Optimizing Software in C++. An Optimization Guide for Windows, Linux and Mac Platforms*. http://www.agner.org/optimize, Copenhagen University College of Engineering.

A. Fog. 2012c. *Optimizing Subroutines in Assembly Language. An Optimization Guide for x86 Platforms*. http://www.agner.org/optimize, Copenhagen University College of Engineering.

L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2 (2007), Article No. 13. Software available at http://www.mpfr.org.

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

J. von zur Gathen and J. Gerhard. 2003. *Modern Computer Algebra* (2nd ed.). Cambridge University Press.

P. Gaudry and E. Thomé. 2007. The mpFq library and implementing curve-based key exchanges. In *SPEED: Software Performance Enhancement for Encryption and Decryption*. ECRYPT Network of Excellence in Cryptology, Amsterdam, Netherlands, 49–64.

GCC From 1987. GCC, the GNU Compiler Collection. Software available at http://gcc.gnu.org. (From 1987).

K. Geddes, G. Gonnet, and Maplesoft. From 1980. Maple (TM). http://www.maplesoft.com/products/maple. (From 1980).

P. Giorgi, Th. Izard, and A. Tisserand. 2010. Comparison of modular arithmetic algorithms on GPUs. In *Parallel Computing: From Multicores and GPU's to Petascale*, B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. Peters, and Th. Priol (Eds.). Advances in Parallel Computing, Vol. 19. IOS Press, 315–322.

P. Giorgi and R. Lebreton. 2014. Online order basis algorithm and its impact on block wiedemann algorithm. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC'14)*, K. Nabeshima (Ed.). ACM, 202–209.

T. Granlund and others. From 1991. GMP, the GNU multiple precision arithmetic library. (From 1991). Software available at http://gmplib.org.

T. Granlund and P. L. Montgomery. 1994. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI'94)*. ACM, 61–72.

S. Anisul Haque and M. Moreno Maza. 2012. Plain polynomial arithmetic on GPU. *J. Phys.: Conf. Ser.* 385, 1 (2012), 012014.

W. Hart and the FLINT Team. From 2008. FLINT: Fast Library for Number Theory. (From 2008). Software available at http://www.flintlib.org.

W. Hart and the MPIR Team. From 2010. MPIR, Multiple Precision Integers and Rationals. (From 2010). Software available at http://www.mpir.org.

D. Harvey. 2009. A cache-friendly truncated FFT. *Theoret. Comput. Sci.* 410, 27–29 (2009), 2649–2658.

D. Harvey. 2014. Faster arithmetic for number-theoretic transforms. *J. Symbol. Comput.* 60 (2014), 113–119.

D. Harvey and J. van der Hoeven. 2014. On the complexity of integer matrix multiplication. (2014). Preprint available at https://hal.archives-ouvertes.fr/hal-01071191.

D. Harvey and D. S. Roche. 2010. An In-place truncated Fourier transform and applications to polynomial multiplication. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation (ISSAC'10)*, S. M. Watt (Ed.). ACM, 325–329.

D. Harvey and A. V. Sutherland. 2014. Computing Hasse–Witt matrices of hyperelliptic curves in average polynomial time. *LMS J. Comput. Math.* 17 (2014), 257–273. Special Issue A (Algorithmic Number Theory Symposium XI).

W. Hasenplaugh, G. Gaubatz, and V. Gopal. 2007. Fast modular reduction. In *18th IEEE Symposium on Computer Arithmetic, ARITH '07*, P. Kornerup and J.-M. Muller (Eds.). IEEE Computer Society, 225–229.

J. van der Hoeven. 2004. The truncated Fourier transform and applications. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation (ISSAC'04)*, J. Schicho (Ed.). ACM, 290–296.

J. van der Hoeven and G. Lecerf. 2013a. Interfacing Mathemagix with C++. In *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation (ISSAC'13)*, M. Monagan, G. Cooperman, and M. Giesbrecht (Eds.). ACM, 363–370.

J. van der Hoeven and G. Lecerf. 2013b. *Mathemagix User Guide*. CNRS & École polytechnique. http://hal.archives-ouvertes.fr/hal-00785549.

J. van der Hoeven and G. Lecerf. 2013c. On the bit-complexity of sparse polynomial and series multiplication. *J. Symbol. Comput.* 50 (2013), 227–254.

J. van der Hoeven and G. Lecerf. 2015. Faster FFTs in medium precision. In *IEEE 22nd Symposium on Computer Arithmetic*, A. Tisserand and J. Villalba (Eds.). IEEE, 75–82.

J. van der Hoeven, G. Lecerf, B. Mourain, Ph. Trébuchet, J. Berthomieu, D. Diatta, and A. Mantzaflaris. 2011. Mathemagix, the quest of modularity and efficiency for symbolic and certified numeric computation. *ACM SIGSAM Communications in Computer Algebra* 177, 3 (2011). In Section "ISSAC 2011 Software Demonstrations", edited by M. Stillman, p. 166–188.

J. van der Hoeven, G. Lecerf, B. Mourrain, and others. From 2002. Mathemagix. (From 2002). Software available at http://www.mathemagix.org.

Intel Corporation. 2013a. Intel (R) Architecture Instruction Set Extensions Programming Reference. (2013). Ref. 319433-015. 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA. http://software.intel.com/en-us/intel-isa-extensions.

Intel Corporation. 2013b. Intel (R) Intrinsics Guide. (2013). Version 3.0.1, released 7/23/2013. http://software.intel.com/en-us/articles/intel-intrinsics-guide.

Ç. Kaya Koç. 1994. Montgomery reduction with even modulus. *IEE Proc. Comput. Dig. Techn.* 141, 5 (1994), 314–316.

Ç. Kaya Koç, T. Acar, and Jr. Kaliski, B. S. 1996. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro* 16, 3 (1996), 26–33.

D. E. Knuth. 1997. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (3rd ed.). Pearson Education.

G. Lecerf. 2010. Mathemagix: Towards large scale programming for symbolic and certified numeric computations. In *Mathematical Software, ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010 (Lect. Notes Comput. Sci.)*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama (Eds.), Vol. 6327. Springer, 329–332.

D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel. 2011. Automatic SIMD vectorization of fast Fourier transforms for the Larrabee and AVX instruction sets. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, 265–274.

L. Meng and J. Johnson. 2014. High performance implementation of the TFT. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC'14)*, K. Nabeshima (Ed.). ACM, 328–334.

L. Meng, Y. Voronenko, J. R. Johnson, M. Moreno Maza, F. Franchetti, and Y. Xie. 2010. Spiral-generated modular FFT algorithms. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation (PASCO'10)*. ACM, 169–170.

N. Möller and T. Granlund. 2011. Improved division by invariant integers. *IEEE Trans. Comput.* 60, 2 (2011), 165–175.

P. L. Montgomery. 1985. Modular multiplication without trial division. *Math. Comp.* 44, 170 (1985), 519–521.

M. Moreno Maza and Y. Xie. 2010. FFT-based dense polynomial arithmetic on multi-cores. In *High Performance Computing Systems and Applications*, D. J. K. Mewhort, N. M. Cann, G. W. Slater, and T. J. Naughton (Eds.). Lect. Notes Comput. Sci., Vol. 5976. Springer, Berlin, 378–399.

N. Nedjah and L. de Macedo Mourelle. 2006. A review of modular multiplication methods and respective hardware implementations. *Informatica* 30, 1 (2006), 111–129.

T. Ogita, S. M. Rump, and S. Oishi. 2005. Accurate sum and dot product. *SIAM J. Sci. Comput.* 26, 6 (2005), 1955–1988.

J. M. Pollard. 1971. The fast Fourier transform in a finite field. *Math. Comp.* 25, 114 (1971), 365–374.

G. van Rossum and J. de Boer. 1991. Interactively testing remote servers using the Python programming language. *CWI Quart,* 4, 4 (1991), 283–303. Software available at http://www.python.org.

M. J. Schulte, J. Omar, and E. E. Jr. Swartzlander. 1994. Optimal initial approximations for the Newton-Raphson division algorithm. *Computing* 53, 3–4 (1994), 233–242.

V. Shoup. 2015. *NTL: A Library for Doing Number Theory*. Software, version 9.1.0. http://www.shoup.net/ntl.

W. A. Stein and others. From 2004. *Sage Mathematics Software*. The Sage Development Team. Software available at http://www.sagemath.org.

E. Thomé. 2012. Théorie algorithmique des nombres et applications à la cryptanalyse de primitives cryptographiques. http://www.loria.fr/∼thome/files/hdr.pdf. (2012). Mémoire d'habilitation à diriger des recherches de l'Université de Lorraine, France.

Received July 2014; revised June 2015; accepted January 2016