High performance SIMD modular arithmetic for polynomial evaluation

Pierre Fortin, Ambroise Fleury, François Lemaire, Michael Monagan

High performance SIMD modular arithmetic for polynomial evaluation

Pierre Fortin[ab], Ambroise Fleury[b], François Lemaire[b], and Michael Monagan[c]

[a]Sorbonne Université, CNRS, LIP6, Paris, France
[b]Université de Lille, CNRS, Centrale Lille, CRIStAL, Lille, France
[c]Department of Mathematics, Simon Fraser University,
Burnaby, B.C., V5A 1S6, Canada
Emails: pierre.fortin@sorbonne-universite.fr, ambroise.fleury.etu@univ-lille.fr,
francois.lemaire@univ-lille.fr, mmonagan@cecm.sfu.ca

April 23, 2020

Abstract

Two essential problems in Computer Algebra, namely polynomial factorization and polynomial greatest common divisor computation, can be efficiently solved thanks to multiple polynomial evaluations in two variables using modular arithmetic. In this article, we focus on the efficient computation of such polynomial evaluations on one single CPU core. We first show how to leverage SIMD computing for modular arithmetic on AVX2 and AVX-512 units, using both intrinsics and OpenMP compiler directives. Then we manage to increase the operational intensity and to exploit instruction-level parallelism in order to increase the compute efficiency of these polynomial evaluations. All this results in the end to performance gains up to about 5x on AVX2 and 10x on AVX-512.

Keywords: modular arithmetic; SIMD; polynomial evaluation; operational intensity; instruction-level parallelism

1 Introduction

Computer Algebra, also called symbolic computation, consists of developing algorithms and data structures for manipulating mathematical objects in an *exact* way. Multivariate polynomials with rational coefficients are essential objects in Computer Algebra, and they naturally arise in many applications (Mechanics, Biology, ...), especially in non-linear problems. Among the classical operations on multivariate polynomials (sum, product, quotient, ...), two non trivial operations are essential: polynomial factorization and polynomial gcd (greatest common divisor) computation[1, 2, 3]. Those operations are necessary for solving polynomial systems and simplifying quotients of multivariate polynomials.

Modern algorithms[4, 5] for factorization and gcd computation rely on many polynomial evaluations, which dominate the overall computation cost. These polynomial evaluations have two main features. First, these are partial evaluations in the sense that not all variables are evaluated: given a polynomial with n variables, we evaluate $n-2$ variables which results in a polynomial with 2 variables. Second, the variables are evaluated at integers modulo a prime p thus all integer arithmetic operations are performed modulo p. Computing modulo a 64

bit prime p makes it possible to use machine integers and native CPU operations, instead of arbitrary-precision integers. Since these partial modular polynomial evaluations are currently a performance bottleneck for polynomial factorizations and gcd computations, we aim in this article to speed-up their computation on modern CPUs.

We focus here on one compute server since most symbolic computations are usually performed on personal workstations. We distinguish three main techniques to obtain performance gain on current CPU compute servers[6]:

- increasing the compute efficiency. This can be achieved by increasing the operational intensity, i.e. the number of operations per byte of memory (DRAM) traffic, which helps exploit modern HPC architectures whose off-chip memory bandwidth is often the most performance constraining resource [7]. The compute efficiency can also be increased by better filling the pipelined floating-point units with instruction-level parallelism;

- exploiting data-level parallelism on vector (or SIMD - single instruction, multiple data) units. Such parallelism is increasingly important in the overall CPU performance since the SIMD vector width has been constantly increasing from 64 bits (MMX[8] and 3DNow![9]) to 128 bits (SSE[10], AltiVec[11]), then to 256 bits (AVX and AVX2[12]), and recently to 512 bits (AVX-512[13]). For 64-bit integers, such AVX-512 SIMD units can now offer a 8x speedup with respect to a scalar computation. But efficiently exploiting such parallelism requires "regular" algorithms where the memory accesses and the computations are similar among the lanes of the SIMD vector unit. Moreover, multiple programming paradigms can be used for SIMD programming: intrinsics, compiler directives, automatic vectorization, to name a few. For key applications, it is important to determine which programming paradigm can offer the highest programming level (ease of programming and portability) without sacrificing (too much) performance.

- exploiting thread-level parallelism on the multiple cores and on the multiple processors available in a compute server. This can be achieved thanks to multi-process, multi-thread or multi-task programming.

Thread-level parallelism has already been introduced for the partial modular polynomial evaluation (see e.g. Hu and Monagan[5]). We will therefore focus here on single-core optimizations, namely increasing the compute efficiency and SIMD parallelism, and we present in this article the following contributions.

- Multiple algorithms have been designed to efficiently compute modular multiplications in a scalar mode. We first justify why the floating-point (FP) based algorithm with FMA (fused multiply-add) of van der Hoeven et al.[14] is best suited for current HPC architectures, especially regarding SIMD computing. We show that the optimized AVX version implementation of van der Hoeven et al.[14] can safely be used in our polynomial evaluation, and we then propose the first (to our knowledge) implementation of such modular multiplication algorithm on AVX-512, as well as the corresponding FP-based modular addition. With respect to the reference polynomial evaluation implementation of Monagan and coworkers[4, 5], which relies on scalar integer-based algorithms for modular operations, we detail the performance gains of our SIMD FP-based modular arithmetic for modular operation microbenchmarks and for the polynomial evaluation.

- We carefully compare intrinsics and OpenMP compiler directives for programming SIMD modular arithmetic and for their integration within the polynomial evaluation. We detail the relevance, the advantages, the issues and the performance results of both programming paradigms.

- We show how to significantly improve the performance of the modular polynomial evaluation by increasing the operational intensity via data reuse, and by filling the pipelines of the floating-point units. This is achieved thanks to the introduction of multiple "dependent" and "independent" evaluations and loop unrolling. We also show that close-to-optimal performance can be obtained without extra memory requirements.

In the rest of this article, we first introduce in Sect. 2 our partial polynomial evaluation. Then we detail in Sect. 3 our integration of SIMD computing first in modular arithmetic, then in the polynomial evaluation. Finally, we show how we can increase the compute efficiency of our polynomial evaluation in Sect. 4.

2 Modular polynomial evaluation in two variables

2.1 Presentation

We are given a multivariate polynomial $f \in \mathbb{Z}[x_1, \ldots, x_n]$ and we want to evaluate the variables $x_3, \ldots, x_n$ in f at integers modulo a prime p. Let $\mathbb{F}_p$ denotes the finite field of integers modulo a prime p. The prime p is chosen so that all integer arithmetic in $\mathbb{F}_p$ can be done with the hardware arithmetic logic units of the underlying machine. For example, with a machine which supports a 64 bit by 64 bit integer multiply, the application would use $p < 2^{64}$. We will pick non-zero integers $\beta_3, \beta_4, \ldots, \beta_n$ uniformly at random from $\mathbb{F}_p$ and compute a sequence of T evaluations

$$b_t(x_1, x_2) = f(x_1, x_2, \beta_3^t, \beta_4^t, \ldots, \beta_n^t) \quad \text{for} \quad 1 \leq t \leq T.$$

The values b_t are polynomials in $\mathbb{F}_p[x_1, x_2]$. We call them bivariate images of f. For convenience we will use the notation

$$\beta = (\beta_3, \beta_4, \ldots, \beta_n) \in \mathbb{F}_p^{n-2}, \quad \text{and} \quad \beta^t = (\beta_3^t, \beta_4^t, \ldots, \beta_n^t) \in \mathbb{F}_p^{n-2},$$

so that we may write $b_t(x_1, x_2) = f(x_1, x_2, \beta^t)$.

Before presenting two application examples of such computation, we first emphasize that we evaluate here at powers of β, not at T different random points in $\mathbb{F}_p^{n-2}$, since this enables one to save computations. Indeed, when Zippel[15] introduced the first sparse polynomial interpolation algorithm and used his algorithm to compute a gcd of two polynomials f and h in $\mathbb{Z}[x_1, \ldots, x_n]$, he used random evaluation points. To interpolate a polynomial with s terms his method solves one or more $s \times s$ linear systems modulo p. Using Gaussian elimination this does $O(s^3)$ arithmetic operations in $\mathbb{F}_p$ and requires space for $O(s^2)$ elements of $\mathbb{F}_p$. Then Zippel[16] showed that if powers β^t are used for the evaluation points, the linear systems are transposed Vandermonde systems which can be solved using only $O(s^2)$ operations in $\mathbb{F}_p$ and $O(s)$ space. In Sect. 2.3, we will see that using powers of β also reduces the polynomial evaluation cost.

Second, we also emphasize that we evaluate at β^t for $1 \leq t \leq T$, and not for $0 \leq t < T$, since the evaluation point $\beta^0 = (1, 1, \ldots, 1)$ may not be usable. For example, consider the following gcd problem in $\mathbb{Z}[x_1, x_2, x_3, x_4]$. Let

$$f = gc \ \text{ and } h = gc + gd \quad \text{where} \quad c = x_1x_2 + x_3x_4, d = x_3 - x_4 \ \text{ and } g \in \mathbb{Z}[x_1, x_2, x_3, x_4].$$

Since $\gcd(c, d) = 1$ we have

$$\gcd(f, h) = \gcd(gc, gc + gd) = \gcd(gc, gd) = g \gcd(c, d) = g.$$

But suppose we use $\beta^0 = (1, 1)$. Then since $d(x_1, x_2, 1, 1) = 0$ we have

$$\gcd(f(x_1, x_2, \beta^0), h(x_1, x_2, \beta^0)) = g(x_1, x_2, 1, 1)c(x_1, x_2, 1, 1) = g(x_1, x_2, 1, 1)(x_1x_2 + 1).$$

We cannot interpolate g using this image. We say $(1, 1)$ is an unlucky evaluation point. Such unlucky evaluation points are avoided with high probability by picking β at random from $\mathbb{F}_p^{n-2}$ and evaluating at β^t for t starting at 1.

2.2 Application examples

Such bivariate images of f are needed in modern algorithms of Computer Algebra for factoring polynomials with integer coefficients and for computing gcd of polynomials with integer coefficients. Two examples are presented below.

2.2.1 Polynomial factorization

Regarding polynomial factorization[1, 2], Monagan and Tuncer[17, 4] reduce factorization of a multivariate polynomial f in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$ to (i) evaluating $f(x_1, x_2, \beta^t)$ for $1 \le t \le T$, (ii) doing a computation with the bivariate images, and (iii) recovering the factors of f using sparse interpolation techniques[15, 16, 18]. See Roche[19] for a recent discussion on sparse polynomial interpolation methods and an extensive bibliography. If $f = \prod_{i=1}^{r} f_i$ is the factorization of f over $\mathbb{Z}$ then usually the factors f_i have a lot fewer terms than their product f. Furthermore, because the method interpolates the coefficients of the factors f_i from bivariate images in x_1 and x_2, the largest coefficient is likely to have a lot fewer terms than the factor f_i. Because of this, the evaluation step dominates the cost. The interpolation step, though more complicated, is cheaper because the coefficients of the factors f_i being interpolated have far fewer terms than f which is being evaluated.

We wish to give an example to illustrate the numbers involved. We consider the factorization of determinants of symmetric Toeplitz matrices[4]. The m'th symmetric Toeplitz matrix T_m is an $m \times m$ matrix in m variables $x_0, x_1, \ldots, x_{m-1}$ where $T_{ij} = x_{|i-j|}$. For example

$$T_4 = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ x_1 & x_0 & x_1 & x_2 \\ x_2 & x_1 & x_0 & x_1 \\ x_3 & x_2 & x_1 & x_0 \end{bmatrix}.$$

The problem is to factor the polynomial $\det T_m$. For $m = 14$, $\det T_m$ has $s = 5,165,957$ terms. It factors into two factors each with $34,937$ terms. The largest coefficient has $u = 9,705$ terms. Thus u, the size of the interpolation problem, is 532 times smaller than s, the size of the evaluation problem.

2.2.2 Polynomial gcd

Given two polynomials $f, h \in \mathbb{Z}[x_1, x_2, \ldots, x_n]$, to compute $g = \gcd(f, h)$, the parallel algorithm of Hu and Monagan[5, 20] works by computing bivariate images of g modulo a prime p, namely,

$$g_t(x_1, x_2) = \gcd(f(x_1, x_2, \beta^t), h(x_1, x_2, \beta^t)) \quad \text{for} \quad 1 \le t \le T.$$

It then uses sparse interpolation techniques to interpolate g from the images $g_t(x_1, x_2)$. Since g is a factor of f and h, the number of terms in g is typically much fewer than the number in f and h. Let us use the notation $\#f$ for the number of terms of a polynomial f. So for the gcd problem, typically $\#g \ll \max(\#f, \#h)$. Hu and Monagan[20] present a "benchmark" problem

where $\#f = 10^6$, $\#h = 10^6$, and $\#g = 10^4$. If one interpolates g from univariate images then the largest coefficient of g in x_1 has 1108 terms. If instead, as the authors recommend, one interpolates g from bivariate images, then the largest coefficient of g in x_1 and x_2 has only $u = 122$ terms. So for this problem, u is almost 10,000 times smaller than $\max(\#f, \#h)$, the size of the evaluation problem. Again, because of this, the authors found that the evaluations of the input polynomials f and h completely dominate the cost of polynomial gcd computations.

Thus two very central problems in Computer Algebra, namely, polynomial factorization and polynomial gcd computation are usually dominated by evaluations when there are many variables.

2.3 The matrix method

Let p be a prime and let $f \in \mathbb{F}_p[x_1, x_2, \ldots, x_n]$. We may write f as

$$f = \sum_{i=1}^{s} a_i x_1^{d_i} x_2^{e_i} M_i(x_3, \ldots, x_n) \tag{1}$$

where $a_i \in \mathbb{F}_p$ are non-zero, d_i and e_i are non-negative integers and M_i is a monomial in $x_3, \ldots, x_n$. For $\beta = (\beta_3, \beta_4, \ldots, \beta_n) \in \mathbb{F}_p^{n-2}$, we want to compute T partial evaluations

$$b_t(x_1, x_2) = f(x_1, x_2, \beta^t) = \sum_{i=1}^{s} a_i x_1^{d_i} x_2^{e_i} M_i(\beta^t) \quad \text{for } 1 \le t \le T.$$

If we let $m_i = M_i(\beta_3, \ldots, \beta_n) \in \mathbb{F}_p$ and $M_i(x_3, \ldots, x_n) = \prod_{k=3}^{n} x_k^{d_{ik}}$ then we observe that

$$M_i(\beta^t) = \prod_{k=3}^{n} (\beta_k^t)^{d_{ik}} = \prod_{k=3} (\beta_k^{d_{ik}})^t = M_i(\beta)^t = m_i^t.$$

Thus

$$b_t(x_1, x_2) = f(x_1, x_2, \beta^t) = \sum_{i=1}^{s} a_i x_1^{d_i} x_2^{e_i} m_i^t.$$

Now we can present the "matrix method"[5] which relies on the powers of β to efficiently compute the T bivariate images. First we compute the monomial evaluation m_i by evaluating $M_i(\beta_3, \ldots, \beta_n)$. To do this, let $d_k = \deg(f, x_k)$ and let $d = \max_{k=3}^{n} d_k$. We pre-compute tables of powers

$$\left[\beta_k^i \text{ for } 0 \le i \le d_k\right] \quad \text{for } 3 \le k \le n.$$

This takes at most $(n-2)(d-1)$ multiplications. Then, for $1 \le i \le s$ we compute $m_i = M_i(\beta)$, using $n-3$ multiplications for each m_i thanks to the tables of powers, and thus using $(n-3)s$ multiplications in total. Therefore we can compute the m_i with $O(nd + ns)$ multiplications. Computing

$$b_1(x_1, x_2) = f(x_1, x_2, \beta) = \sum_{i=1}^{s} a_i m_i x_1^{d_i} x_2^{e_i}$$

needs 1 multiplication for each $a_i m_i \in \mathbb{F}_p$, hence s multiplications in total. We can compute the next evaluation

$$b_2(x_1, x_2) = f(x_1, x_2, \beta^2) = \sum_{i=1}^{s} a_i m_i^2 x_1^{d_i} x_2^{e_i}$$

Algorithm 1 Compute kernel of the matrix method for T bivariate images of a polynomial $f \in \mathbb{F}_p[x_1, x_2, \ldots, x_n]$, using notations of Eq. (1). Inputs are the vector $m = [M_1(\beta), \ldots, M_s(\beta)] \in \mathbb{F}_p^s$ of monomial evaluations and the coefficient vector $a = [a_1, \ldots, a_s] \in \mathbb{F}_p^s$.

for each evaluation $1 \leq t \leq T$ **do**
  $i \leftarrow 1$; $b_t \leftarrow 0$
  while $i \leq s$ **do**
    $c \leftarrow 0$
    $J \leftarrow$ #monomials with same (d_i, e_i)
    for $i \leq j < i + J$ **do**
      $a[j] \leftarrow a[j] \otimes_p m[j]$ ▷ Hadamard product
      $c \leftarrow c \oplus_p a[j]$ ▷ coefficient reduction
    end for
    if $c \neq 0$ **then** add $c x_1^{d_i} x_2^{e_i}$ to bivariate image b_t
    $i \leftarrow i + J$
  end while
end for

using another s multiplications if we save $a_i m_i \in \mathbb{F}_p$ for $1 \leq i \leq s$ and multiply them by m_i. This leads to an algorithm that computes the T evaluations using $O(nd+ns)$ multiplications to compute the m_i, plus a further sT multiplications to compute the $a_i m_i^t$ for $1 \leq t \leq T$, $1 \leq i \leq s$, hence $O(nd + ns + sT)$ multiplications in total. With random points instead of powers of β, the T evaluations would have required a larger operation count of $O(nT(d+s))$ multiplications in total[5].

One way to see the matrix method is to think of evaluating f at β^t as the following $t \times s$ matrix-vector multiplication.

$$\begin{bmatrix} m_1 & m_2 & \ldots & m_s \\ m_1^2 & m_2^2 & \ldots & m_s^2 \\ \vdots & \vdots & \vdots & \vdots \\ m_1^T & m_2^T & \ldots & m_s^T \end{bmatrix} \begin{bmatrix} a_1 x_1^{d_1} x_2^{e_1} \\ a_2 x_1^{d_2} x_2^{e_2} \\ \vdots \\ a_s x_1^{d_s} x_2^{e_s} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_T \end{bmatrix}$$

In practice, the complete matrix is not explicitly built and we take advantage of the connection between successive rows of the matrix. Let $a = [a_1, a_2, \ldots, a_s]$, $m = [m_1, m_2, \ldots, m_s]$ and $X = [x_1^{d_1} x_2^{e_1}, x_1^{d_2} x_2^{e_2}, \ldots, x_1^{d_s} x_2^{e_s}]$. Let $u \circ v$ denote the Hadamard product of two vectors $u, v \in \mathbb{F}_p^s$, that is $u \circ v = [u_1 v_1, u_2 v_2, \ldots, u_s v_s] \in \mathbb{F}_p^s$. Then, viewing $b_1(x_1, x_2)$ and $b_2(x_1, x_2)$ as vectors of terms, we have

$$b_1(x_1, x_2) = (a \circ m) \circ X \qquad \text{and} \qquad b_2(x_1, x_2) = ((a \circ m) \circ m) \circ X.$$

Finally, for a given t we will have to compute the sum $c_{i,t}$ of $a_i m_i^t$ for all $a_i m_i^t x_1^{d_i} x_2^{e_i}$ sharing the same d_i and e_i values. This sum $c_{i,t}$ is indeed the coefficient of $x_1^{d_i} x_2^{e_i}$ in $b_t(x_1, x_2)$. These "coefficient reductions" are required since in Eq. (1), multiple $M_i(x_3, \ldots, x_n)$ can potentially share the same d_i and e_i values. If the monomials $x_1^{d_i} x_2^{e_i} M_i(x_3, \ldots, x_n)$ in the input polynomial f are sorted in lexicographical order with $x_1 > x_2 > x_3 > \cdots > x_n$ then the monomials $x_1^{d_i} x_2^{e_i}$ will be sorted in X which makes adding up $c_{i,t}$ coefficients of like monomials in $b_t(x_1, x_2)$ straightforward (with $O(s)$ additions for each evaluation). Doing so, we compute $f(x_1, x_2, \beta^t)$ for $1 \leq t \leq T$ with $O(nd + ns + sT)$ multiplications and $O(sT)$ additions.

The resulting algorithm for the compute kernel of the matrix method is detailed in Algorithm 1, where we save the successive $a_i m_i^t$ values in the a vector, and where $\oplus_p$ and $\otimes_p$ denote the

arithmetic operators modulo p ($c = a \otimes_p b$ denoting $c \equiv a \times b \pmod{p}$, and $d = a \oplus_p b$ denoting $d \equiv a + b \pmod{p}$ with $(a, b, c, d) \in \mathbb{F}_p^4$).

In this article, we will use by default the following parameters when measuring the time or performance of our partial modular polynomial evaluation with 64-bit integers: $s = 5 \times 10^5$ terms; $n = 6$ variables, hence 4 evaluated variables; a maximum degree of $d = 10$ in each variable; and the number of evaluations T chosen here as 10000 to have a measurable computation time, but T can be much lower in actual use. These parameters have been chosen to be realistic and to lead to stable and reproducible performance results.

2.3.1 Multi-core parallel evaluation

Hu and Monagan[5] parallelized the matrix method for partial modular polynomial evaluations on a multi-core architecture with N cores by doing N evaluations at a time. They first compute $\Gamma = [m_1^N, m_2^N, \ldots, m_s^N]$ using exponentiations by squaring, requiring $O(s \log_2 N)$ multiplications. Then they compute $\Lambda_k = a \circ [m_1^k, m_2^k, \ldots, m_s^k]$ for $1 \leq k \leq N$ using Ns multiplications. Then, in parallel, the k'th core successively computes $f(x_1, x_2, \beta^{k+N}) = \Lambda_k \circ \Gamma \circ X$; $f(x_1, x_2, \beta^{k+2N}) = \Lambda_k \circ \Gamma \circ \Gamma \circ X$; $f(x_1, x_2, \beta^{k+3N}) = \Lambda_k \circ \Gamma \circ \Gamma \circ \Gamma \circ X$; $\ldots$ using Algorithm 1 (with each Λ_k as the a vector, and Γ as the m vector). This was implemented with multi-task programming in Cilk[5]. This method significantly increases the space needed as N vectors $C_1, C_2, \ldots, C_N$ of length s are required where s can be very large. Monagan and Tuncer[17] introduced an alternative parallelization strategy by using a 1D block decomposition of the a and m vectors for each evaluation.

Finally, we mention the asymptotically fast method[21] for computing $f(x_1, x_2, \beta^t)$. If p is chosen of the form $p - 1 = 2^k q$ with $2^k > T$ so that an FFT of order 2^k can be done in the finite field $\mathbb{F}_p$, after computing the monomial evaluations $m = [m_1, m_2, \ldots, m_s]$, this method computes $f(x_1, x_2, \beta^t)$ for $1 \leq t \leq T$ in $O(s \log^2 T)$ multiplications. Monagan and Wong[22] found that a serial implementation of this fast method first beat the matrix method at $T = 504$ but that it was much more difficult to parallelize than the matrix method – the fast method required $s \gg 10^6$ to deliver good parallel speedups. Moreover, the simplicity and data locality of the matrix method makes it very suitable for vectorization and other single-core optimizations targeted in this article.

3 SIMD modular arithmetic for partial polynomial evaluation

3.1 Selection of the modular arithmetic algorithm

Given a fixed[1] integer $p > 1$, we focus on the efficient computation of $c = a \otimes_p b$ and $d = a \oplus_p b$. We target the algorithm that will offer the best performance: such an algorithm must thus be efficient in scalar mode (i.e. non-SIMD), while being also suitable for vectorization. While $\oplus_p$ can be implemented with a compare instruction, $\otimes_p$ requires integer divisions which are expensive operations on current processors[23], and for which no SIMD integer division instruction is available in SSE, AVX or AVX-512. Hence, various alternate algorithms have been designed in order to efficiently compute $\otimes_p$. We briefly recall the most important ones.

In order to compute $c = a \otimes_p b$, one can first rely on floating-point arithmetic to compute $q = \lfloor \frac{a \times b}{p} \rfloor$ and then deduce $c = a \times b - q \times p$ (see for example Alverson[24], Baker[25]). This requires conversions to/from floating-point numbers, and the number of bits of the floating-point number mantissa has to be twice as large as the number of bits of p (to hold the product).

[1]The value of p is fixed in this article, but is not a constant (from the programming point of view) in our implementation. Our implementation will indeed be used for multiple p values, which are unknown at compile time. The compiler cannot therefore optimize the code for a specific p value.

Algorithm 2 Modular multiplication of 64-bit integers x and y with a 50-bit prime p. x and y are considered to be already reduced modulo p, and converted to `double` along with p prior to the beginning of the algorithm. u stores: `1/(double) p`

1: double $h \leftarrow x * y$;
2: double $\ell \leftarrow \text{fma}(x,\ y,\ -h)$;
3: double $b \leftarrow h * u$;
4: double $c \leftarrow \text{floor}(b)$; ▷ c is the quotient ± 1
5: double $d \leftarrow \text{fma}(-c,\ p,\ h)$;
6: double $g \leftarrow d + \ell$; ▷ g is the remainder $\pm p$
7: **if** $g \geq p$ **then** return $g - p$;
8: **if** $g < 0.0$ **then** return $g + p$;
9: return g ;

Algorithm 3 Modular addition of 64-bit integers x and y with a 50-bit prime p. x and y are considered to be already reduced modulo p, and converted to `double` along with p prior to the beginning of the algorithm.

1: double $s \leftarrow x + y$;
2: return $s \geq p\ ?\ s - p\ :\ s$;

In order to avoid the conversions between floating point numbers and integers, the floating-point reciprocal p^{-1} can be rescaled and truncated into an integer. The quotient q is hence approximated, and some adjustments enable one to obtain the remainder c. This integer-based method is known as the *Barrett's product*[26, 27]. Another integer-based approach relies on Montgomery's reduction[28]: a comparison between the two methods has been done for example by van der Hoeven et al.[14]. An improved version of Barrett's product with integer only operations has been proposed by Möller and Granlund[29]: Monagan and coworkers[5, 17] use an implementation (written by Roman Pearce) of this latter[29] method (with p^{-1} precomputed) in their original code for polynomial evaluation with 64-bit integers. This offers a 11x performance gain[30] for $\otimes_p$ with respect to one integer division. However, this implementation relies on 128-bit intermediate results: for SIMD processing on 64-bit elements, this implies that only half of the SIMD lanes will be used, hence leading to twice lower SIMD speedups. One can replace these 128-bit variables with two 64-bit variables (hence using only one 64-bit lane per operation), but this requires extra arithmetic and bit shifting. Moreover, to our knowledge there is no SSE/AVX2/AVX-512 intrinsic which performs multiplications on 64-bit integers and provides either the 128-bit results (similarly to the `_mm{,256,512}_mul_epu32` intrinsics on 32-bit integers) or their upper and lower 64-bit parts. This greatly complicates the SIMD programming of the Möller and Granlund[29] algorithm for 64-bit integers.

We therefore focus in this article on the use of floating-point (FP) FMA (fused multiply-add) instructions for floating-point based modular arithmetic. Since the FMA instruction performs two operations ($a * b + c$) with one single final rounding, it can indeed be used to design a fast *error-free transformation* of the product of two floating-point numbers[31]. Such error-free transformation computes the accurate floating-point result of the product. As described and proved by van der Hoeven et al.[14], this makes it possible to design a modular multiplication with double-precision floating-point numbers, provided that p has at most 50 bits: see Algorithm 2. Intuitively, an error-free transformation (Lines 1 and 2 in Algorithm 2) enables one to compute in twice working precision[31], and hence to precisely handle the multiplication result before reduction modulo p. More precisely, ℓ stores the rounding error of the product $x * y$ (i.e. $h + \ell$ exactly equals $x * y$). The approximate real quotient $(x * y)/p$ is then computed in b

using the pre-computed $u = 1/(double)p$, and rounded to an (approximate) integer quotient c. A first approximate remainder d is computed using $c * p$, and added to ℓ in g in order to take into account the initial rounding error. g is finally corrected so that we exactly have: $g \equiv x \times y \pmod{p}$. We emphasize that all this is achieved with 64-bit floating-point numbers only: no larger variables are required and we can thus benefit from the full SIMD speedup (up to 8x on AVX-512). The corresponding FP-based modular addition algorithm is presented in Algorithm 3.

We also emphasize that the limit on the size of p (at most 50 bits) is not problematic regarding our targeted applications (presented in Sect. 2.2). Indeed, let $f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{s} a_i M_i(x_1, x_2, \ldots, x_n)$ be a polynomial we wish to interpolate. Ben-Or/Tiwari [18] and Zippel [16] both pick $\beta \in \mathbb{F}_p^n$ at random and interpolate f from $f(\beta), f(\beta^2), f(\beta^3), \ldots$. Both methods require the monomial evaluations to be distinct, that is, $M_i(\beta) \neq M_j(\beta)$ for $1 \leq i < j \leq s$. For this to hold with reasonable probability we require $p > 100s^2$. For a large value of s, say $10^4 < s < 10^6$, the requirement $p > 100s^2$ means 32-bit primes are too small but 50-bit primes are sufficient.

Finally, we stress that relying on FMAs is relevant regarding HPC architectures. Current high-end HPC-oriented Intel CPUs with AVX2 or AVX-512 indeed offer two FMA SIMD units. HPC-oriented GPUs from NVIDIA or AMD (not studied in this article), whose performance strongly depends on SIMD computing, also fully support FMA instructions.

3.2 SIMD programming paradigms

We plan to integrate the SIMD implementations of the FMA-based $\oplus_p$ and $\otimes_p$ modular operations in the polynomial evaluation algorithm (see Algorithm 1) on AVX2 or AVX-512 CPUs. For this purpose, there are multiple programming paradigms regarding SIMD computing.

A first possibility is to rely on intrinsics programming. Such low-level programming enables the programmer to reach high performance, but at a non-negligible development cost. This will be our primary programming paradigm, and we will detail the corresponding implementations in Sect. 3.3.

A second possibility is to rely on the compiler to benefit from a higher programming level. Compilers can detect parallel and vectorizable loops and automatically vectorize these loops. But, as further detailed in Sect. 3.5, such automatic vectorization will fail in our polynomial evaluation. Therefore, we consider here a third programming paradigm: compiler directives for SIMD programming. In C/C++ programming, these are pragmas which enable the programmer to indicate (and ensure) that a given loop is parallel: no dependency analysis is then required by the compiler. Such compiler directives are available in the Intel C/C++ Compiler ICC (`#pragma simd`), and have been standardized in the last versions of OpenMP[2] (starting from OpenMP 4.0). We will rely here on OpenMP due to its sustainability, its wide usage in HPC, and its availability in both ICC and GCC (the GNU Compiler Collection). Such high level programming with OpenMP will enable us to avoid writing intrinsics, to have one scalar C code for both AVX2 and AVX512, and to avoid array padding or loop splitting when the iteration number is not a multiple of the SIMD vector size. OpenMP directives will also enable us to overcome the limits of the automatic vectorization for our polynomial evaluation. However, the SIMD code generated by the compiler may differ from the intrinsic code and hence lead to lower performance.

In the rest of Sect. 3, we will thus investigate these two SIMD programming paradigms: SIMD intrinsics and OpenMP SIMD directives. Their performance results will also be detailed and compared.

[2] `https://www.openmp.org/`

3.3 SIMD intrinsics and the AVX-512 version

Using AVX intrinsics: Van der Hoeven et al.[14] have presented a SSE/AVX version of Algorithm 2 to implement $\otimes_p$. They use two SSE/AVX `blendv_pd` intrinsics to efficiently implement the two final tests (Lines 7-8 in Algorithm 2), hence removing divergence in the SIMD computation. This `blendv_pd` intrinsic blend `double` elements from two vectors depending on the most significant bit of elements from a third vector. For floating-point elements this most significant bit corresponds to the sign bit, which enables one to implement in SIMD without branching the two final tests using comparisons to 0.0. We will rely on this AVX version on AVX2 CPUs. However, we recall that IEEE standard 754 for floating-point arithmetic[32] includes signed zeros which may lead to incorrect results regarding the use of `blendv_pd`. Indeed if for example g equals -0.0 at Line 8 in Algorithm 2, using the `blendv_pd` instruction directly on g would return p, which is incorrect for modulo p arithmetic. We show below that -0.0 cannot appear in our specific context: we can thus safely use this implementation with AVX intrinsics.

Van der Hoeven et al. also present an AVX $\oplus_p$ implementation (see function 3.9[14]) of the scalar FP-based $\oplus_p$ algorithm (presented in Algorithm 3). Similarly, as shown below, we can safely ignore signed zeros for the the `blendv_pd` instruction used in this AVX version.

Regarding the issue with signed zeros and the AVX `blendv_pd` intrinsic: For $\otimes_p$, we consider the tests at Lines 7 ($g \geq p$ here rewritten as $g - p \geq 0$) and 8 ($g > 0$) in Algorithm 2. We aim at showing here that no -0.0 value will occur in these two tests, so that one can safely use the `blendv_pd` intrinsics to implement in AVX the conditional affectation resulting from these tests. For completeness, we recall beforehand the paragraphs 3 and 4 of §6.3, *The sign bit*, from the IEEE 754 standard[32].

Paragraph 3: *When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be +0 in all rounding-direction attributes except roundTowardNegative; under that attribute, the sign of an exact zero sum (or difference) shall be* -0. *However,* $x + x = x - (-x)$ *retains the same sign as* x *even when* x *is zero.*

Paragraph 4: *When (a * b) + c is exactly zero, the sign of fusedMultiplyAdd(a, b, c) shall be determined by the rules above for a sum of operands. When the exact result of (a * b) + c is non-zero yet the result of fusedMultiplyAdd is zero because of rounding, the zero result takes the sign of the exact result.*

Now, let us first consider the case where x or y is zero (possibly both). Then h computed at Line 1 is either $+0.0$ or -0.0. According to the paragraph 4 quoted just above, the ℓ computed at Line 2 is obtained from the computation $h - h$ which cannot result in -0.0 thanks to paragraph 3 and since we rely on the default rounding mode (Round to nearest). Thus $\ell = +0.0$. Using again paragraph 3, the g computed at Line 6 by $g = d + \ell$ cannot be equal to -0.0 since $\ell = +0.0$. Therefore g at Line 6 equals $+0.0$ (since the expected result of the modular product is zero here): thus no -0.0 is evaluated in the two tests at Lines 7 and 8.

Second, consider the case were both x and y are nonzero. Since p is a prime number, and since $x < p$ and $y < p$, the product $x \times y$ cannot be zero modulo p: this is due to the uniqueness of the prime factorization of $x \times y$. Thus the function returns a nonzero value that is not a multiple of p. Consequently g cannot hold -0.0, and neither can $g - p$ evaluate to -0.0.

For $\oplus_p$, the `blendv_pd` intrinsic evaluates the result of a subtraction by p since the test $s \geq p$ at Line 2 in Algorithm 3 is rewritten as $s - p \geq 0$ (see function 3.9[14]). According to paragraph 3, this subtraction cannot lead to -0.0 since $p \neq -0.0$.

Algorithm 4 AVX-512 modular multiplication of AVX-512 vectors $\bar{x}$ and $\bar{y}$ with a 50-bit prime p replicated in the AVX-512 vector $\bar{p}$. Elements of $\bar{x}$ and $\bar{y}$ are considered to be already reduced modulo p, and converted (like $\bar{p}$) to `double` elements ($\bar{x}$, $\bar{y}$, $\bar{p}$ and $\bar{u}$ being `__m512d` vectors) prior to the beginning of the algorithm. $\bar{u}$ stores replicates of: `1/(double) p`

1: __m512d $\bar{h} \leftarrow$ _mm512_mul_pd($\bar{x}$, $\bar{y}$) ;
2: __m512d $\bar{\ell} \leftarrow$ _mm512_fmsub_pd($\bar{x}$, $\bar{y}$, $\bar{h}$) ;
3: __m512d $\bar{b} \leftarrow$ _mm512_mul_pd($\bar{h}$, $\bar{u}$) ;
4: __m512d $\bar{c} \leftarrow$ _mm512_floor_pd($\bar{b}$) ;
5: __m512d $\bar{d} \leftarrow$ _mm512_fnmadd_pd($\bar{c}$, $\bar{p}$, $\bar{h}$) ;
6: __m512d $\bar{g} \leftarrow$ _mm512_add_pd($\bar{d}$, $\bar{\ell}$) ;
7: __mmask8 $m \leftarrow$ _mm512_cmplt_pd_mask($\bar{g}$, _mm512_setzero_pd()) ;
8: __mmask8 $mm \leftarrow$ _mm512_cmple_pd_mask($\bar{p}$, $\bar{g}$) ;
9: $\bar{g} \leftarrow$ _mm512_mask_add_pd($\bar{g}$, m, $\bar{g}$, $\bar{p}$) ;
10: $\bar{g} \leftarrow$ _mm512_mask_sub_pd($\bar{g}$, mm, $\bar{g}$, $\bar{p}$) ;
11: return $\bar{g}$;

Algorithm 5 AVX-512 modular addition of AVX-512 vectors $\bar{x}$ and $\bar{y}$ with a 50-bit prime p replicated in the AVX-512 vector $\bar{p}$. Elements of $\bar{x}$ and $\bar{y}$ are considered to be already reduced modulo p, and converted (like $\bar{p}$) to `double` elements ($\bar{x}$, $\bar{y}$ and $\bar{p}$ being `__m512d` vectors) prior to the beginning of the algorithm.

1: __m512d $\bar{s} \leftarrow$ _mm512_add_pd($\bar{x}$, $\bar{y}$) ;
2: __mmask8 $m \leftarrow$ _mm512_cmple_pd_mask($\bar{p}$, $\bar{s}$) ;
3: return _mm512_mask_sub_pd($\bar{s}$, m, $\bar{s}$, $\bar{p}$) ;

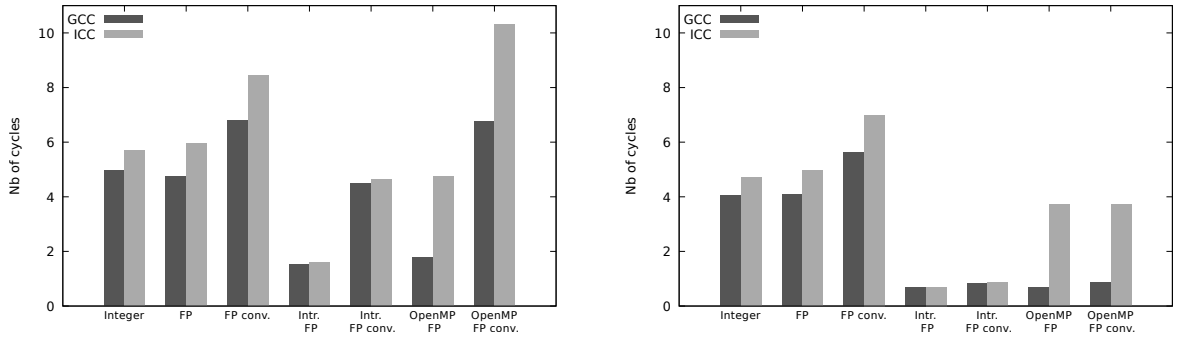
Using AVX-512 intrinsics: Regarding AVX-512, the `blendv_pd` intrinsic is not available: we therefore explicitly build 8-bit masks to conditionally perform (without branching) the addition and the subtraction at the end of the algorithm, as presented in Algorithm 4. Hence the SIMD divergence is efficiently handled within the AVX-512 arithmetic instructions. To our knowledge this is the first AVX-512 floating-point based modular arithmetic. Orisaka et al.[33] have also accelerated modular arithmetic with AVX-512 but using Montgomery reduction and targeting very large primes for cryptography.

Regarding $\oplus_p$, we also adapt the SSE/AVX implementation for floating-point numbers presented by Van der Hoeven et al.[14] to AVX-512. As detailed in Algorithm 5, we use 1 addition, 1 comparison and 1 masked subtraction in AVX-512 instead of 1 addition, 1 subtraction and 1 `blendv_pd` in AVX.

3.4 Microbenchmarks

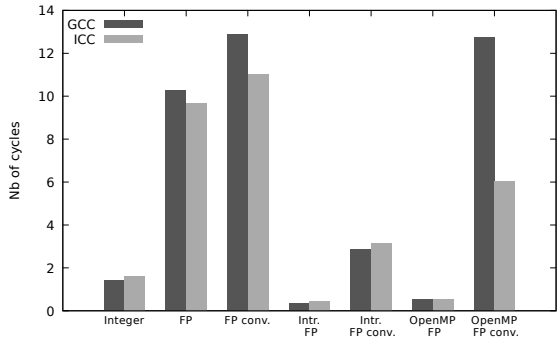
We start with microbenchmarks, presented in Fig. 1, of the two modular arithmetic operations $\otimes_p$ and $\oplus_p$. Like all following performance tests, these microbenchmarks have been performed on the two compute servers *AVX2 server* and *AVX-512 server* presented in Table 1. We first notice that, regarding the original implementation of the modular multiplication (integer based, by R. Pearce), our microbenchmark results are consistent with the 6.016 cycles obtained with GCC by Monagan and coworkers[30] on older CPUs.

Regarding $\otimes_p$ (see Figs. 1(a) and 1(b)), and with respect to the original integer based implementation, the scalar floating-point (FP) based implementation offers lower performance when including back and forth conversions between integers and floating-points numbers, and similar performance when not considering these conversions. The SIMD FP based implementa-

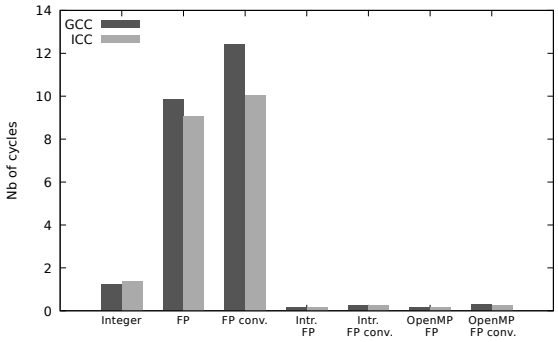


(a) $\otimes_p$ on *AVX2 server*.

(b) $\otimes_p$ on *AVX-512 server*.



(c) $\oplus_p$ on *AVX2 server*.



(d) $\oplus_p$ on *AVX-512 server*.

Figure 1: Performance comparison of various implementations for $\otimes_p$ and $\oplus_p$. `Integer` denotes the original integer-based implementation as used by Monagan and coworkers for $\otimes_p$[5, 17] and $\oplus_p$[34]. `FP` denotes the scalar floating-point based implementation without considering the conversions between integers and floating-points numbers, whereas `FP conv.` includes these conversions. `Intr. FP` (respectively `OpenMP FP`) denotes the SIMD version of the floating-point based implementation using intrinsics (resp. using OpenMP). These performance results have been obtained on element-wise operations over vectors of 2048 elements.

tion offer (with intrinsics, and without considering the conversions) a 3.2x (resp. 3.7x) speedup with GCC (resp. ICC) over the scalar FP-based implementation on *AVX2 server*. On *AVX-512 server*, the speedup is 5.9x (resp. 7.2x) speedup with GCC (resp. ICC). This shows that the performance gain of our new AVX-512 $\otimes_p$ implementations is indeed twice greater than the AVX2 one.

Regarding $\oplus_p$ (see Figs. 1(c) and 1(d)), the scalar FP-based implementation leads to much greater cycle numbers than the integer-based one: this is due to the branching of the compare instruction required in the FP implementation (see Algorithm 3). The comparison in the integer-based $\oplus_p$ implementation can indeed be replaced by shifting[34, 14], which avoids the branching performance impact on the pipeline filling. Thanks to the use of the AVX2 `blendv_pd` intrinsic and of AVX-512 masks, there is also no branching in the SIMD FP-based implementations (with intrinsics) which implies a strong performance gain (around one order of magnitude), in addition to the SIMD speedup. With respect to the scalar integer-based one, the SIMD speedups of the FP-based implementations with intrinsics (without considering the conversions) are 4.0x (resp. 3.6x) with GCC (resp. ICC) on *AVX2 server*, and 8.7x (resp. 8.5x) with GCC (resp. ICC) on *AVX-512 server*. This shows that our AVX-512 $\oplus_p$ implementation is twice faster than the AVX2 one of Van der Hoeven et al.[14].

Table 1: Test platforms.

Servers	Name:	Hardware features:
	AVX2 server	2× Intel Xeon CPU E5-2695 v4 CPUs: 2×18 2-way SMT cores - 2.10 GHz (base) / 3.30 GHz (turbo) - AVX2
	AVX-512 server	2× Intel Xeon Gold 6152 CPUs: 2×22 2-way SMT cores - 2.10 GHz (base) / 3.70 GHz (turbo) - AVX512
Compilers	Name:	Performance-related options:
	GCC 8.2.0	`-O3 -mfma -fno-trapping-math -march=native -mtune=native`
	ICC 19.0.3.199	`-O3 -fma -xhost`

When considering the conversions, the overhead of these conversions can annihilate the SIMD performance gain on *AVX2 server*. This is due to the lack of AVX2 conversion instruction between 64-bit integers and 64-bit floating-point numbers: the conversions are thus performed in scalar mode which has a strong performance impact. In comparison, such a SIMD instruction is available in AVX-512[3], where conversions can be performed in SIMD mode. Their overheard is therefore much lower on *AVX-512 server*.

Finally, we also consider using OpenMP to vectorize the code. The first issue lies in having the compiler generate SIMD FMA instructions from the `fma()` function call in the C+OpenMP code for $\otimes_p$. This is effective with GCC thanks to the `-fno-trapping-math` option which allows us to assume that floating-point operations cannot generate traps, such as division by zero, overflow, underflow, inexact result and invalid operation. Unfortunately, using all possible floating-point model variations (`-fp-model` options) did not enable us to generate SIMD FMA instructions with ICC. The ICC OpenMP code hence relies on scalar FMA instructions, which explains its important performance overhead over the GCC OpenMP code for $\otimes_p$. Secondly, on *AVX-512 server* we had to force the AVX-512 vectorization using `-qopt-zmm-usage=high` with ICC and `-mprefer-vector-width=512` with GCC, otherwise only AVX2 instructions are generated.

As far as $\otimes_p$ is concerned, the OpenMP code (with GCC) has the same performance than the SIMD code written in intrinsics on *AVX-512 server*, but is slower by 18% on *AVX2 server*. This is because of one additional compare instruction added by the compiler before each `blendv_pd` instruction. This comparison to zero (either $g > 0$, or $g - p \geq 0$) is here to prevent any issue with IEEE 754 signed zeros and the `blendv_pd` instruction. The compiler is indeed unaware of our specific context which enables us not to consider -0.0, as shown in Sect. 3.3.

For $\oplus_p$, the compiler similarly adds one unnecessary compare instruction which results in a 45% performance penalty on *AVX2 server* for the OpenMP code (with GCC). However no branching instruction is generated in the SIMD code for $\oplus_p$ with OpenMP, which makes this OpenMP still rather efficient with respect to the scalar integer-based implementation: 2.8x faster on *AVX2 server*, and 7.4x on *AVX-512 server* (with GCC).

3.5 Integration in polynomial evaluation

We can now consider the integration of SIMD modular arithmetic in our partial polynomial evaluation. Due to the cost of the conversions between integers and floating-points numbers (see Sect. 3.4), we choose to perform the first conversion (from integers to floating-point numbers) for each value of the a and m vectors once before the evaluation (i.e. just before Line 1 in Algorithm

[3]More precisely, the 64-bit conversions belong to the AVX-512DQ instruction set which is available on our Intel Xeon Gold 6152 CPUs, but not on the prior Intel Knights Landing (Xeon Phi) processors.

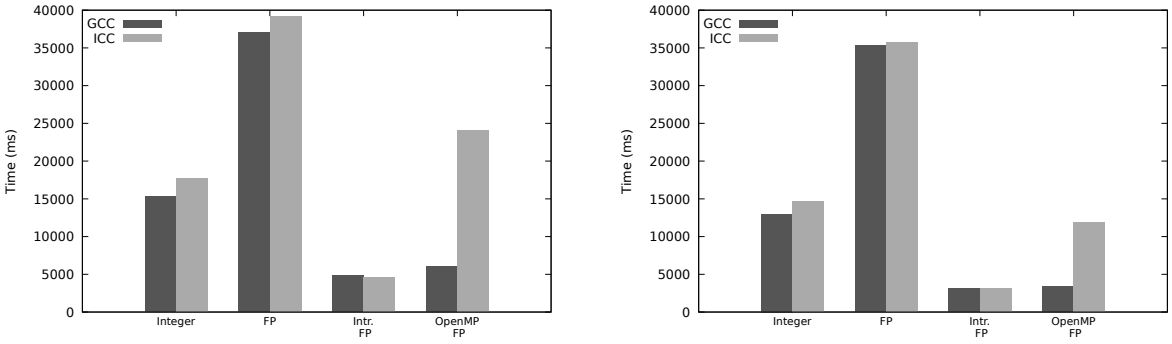
Algorithm 6 SIMD compute kernel of the matrix method (see Algorithm 1 for notations and inputs). $\bar{x}$ denotes the SIMD vector corresponding to variable x. $\mathcal{V}$ is the size of the SIMD vector.

1: In-place conversions for vectors a and m (64-bit integers $\rightarrow$ doubles)
2: **for** each evaluation $1 \leq t \leq T$ **do**
3: $\quad i \leftarrow 1$; $b_t \leftarrow 0$
4: $\quad$ **while** $i \leq s$ **do**
5: $\quad\quad \bar{c} \leftarrow 0.0$
6: $\quad\quad J \leftarrow$ #monomials with same (d_i, e_i)
7: $\quad\quad$ **for** $i \leq j < i + J$ with step $\mathcal{V}$ **do**
8: $\quad\quad\quad \bar{a} \leftarrow a[j \,..\, j + \mathcal{V} - 1]$ ▷ SIMD load
9: $\quad\quad\quad \bar{m} \leftarrow m[j \,..\, j + \mathcal{V} - 1]$ ▷ SIMD load
10: $\quad\quad\quad \bar{a} \leftarrow \bar{a} \otimes_p \bar{m}$ ▷ SIMD Hadamard product
11: $\quad\quad\quad \bar{c} \leftarrow \bar{c} \oplus_p \bar{a}$ ▷ SIMD coefficient reduction
12: $\quad\quad\quad a[j \,..\, j + \mathcal{V} - 1] \leftarrow \bar{a}$ ▷ SIMD store
13: $\quad\quad$ **end for**
14: $\quad\quad c \leftarrow reduce(\bar{c},\ \oplus_p)$ ▷ $\bar{c}$ final reduction
15: $\quad\quad$ **if** $c \neq 0.0$ **then** convert c to 64-bit integer and add $cx_1^{d_i}x_2^{e_i}$ to bivariate image b_t
16: $\quad\quad i \leftarrow i + J$
17: $\quad$ **end while**
18: **end for**

1). These conversions are performed in-place to save memory. The reverse conversion (from floating-point numbers to integers) is only performed once for each reduction result (i.e. the c value at Line 10 in Algorithm 1).

Figure 2 presents performance results for our polynomial evaluation using various modular arithmetic implementations. One can see that the scalar floating-point based modular arithmetic makes our polynomial evaluation about 2.5 times slower than the original implementation by Monagan and coworkers[5] (using integer-based modular arithmetic). This is due to the slow FP-based $\oplus_p$ implementation (because of its branch instruction: see Sect. 3.4).

We now consider SIMD intrinsics to integrate SIMD FP-based modular arithmetic in our originally scalar polynomial evaluation (see Algorithm 1). The resulting SIMD algorithm is written in Algorithm 6. First, a reduction has to be computed within the SIMD vector at the end of the inner loop (Line 14 in Algorithm 6), in order to obtain the final scalar c value. Instead of performing a sequential reduction with the scalar $\oplus_p$ and its branch instruction, we use SIMD shuffle instructions to write a parallel tree-shaped reduction using only SIMD $\oplus_p$ operations. Second, we also have to consider memory alignement which can be important for efficient vector loads and stores. However, the j indices used in the inner loop (Line 7 in Algorithm 6) do not lead to aligned memory accesses since the successive J values are not necessarily multiples of the SIMD width. One could choose to make a copy of the vectors a and m with relevant zero padding to ensure aligned memory accesses, but this would require twice the memory. In order to obtain good SIMD speedups without padding, we explicitly decompose this inner loop into three successive loops $L1$, $L2$ and $L3$ (not shown in Algorithm 6). $L1$ and $L3$ have an iteration count lower than the SIMD width and are vectorized thanks to explicit masks. These two loops ensure that the vectorized $L2$ loop perform aligned memory accesses (i.e. its first j index is a multiple of the SIMD width). We noticed that the use of such a vectorized reduction and of such vectorized $L1$ and $L3$ loops offer additionnal performance gains when processing a lower number of terms: e.g. up to 29% for $s = 5 \times 10^4$ terms. With respect to the original implementation



(a) On *AVX2 server*.

(b) On *AVX-512 server*.

Figure 2: Performance comparison of scalar and SIMD polynomial evaluations. `Integer` denotes the original polynomial evaluation by Monagan and coworkers[5] with scalar integer-based modular arithmetic. `FP` denotes the use of scalar floating-point modular arithmetic for the polynomial evaluation. `Intr. FP` (respectively `OpenMP FP`) denotes the SIMD version of the floating-point based modular arithmetic using intrinsics (resp. using OpenMP).

using integer-based arithmetic, the resulting polynomial evaluation with SIMD intrinsics offers performance gains ranging from 3.13x (resp. 3.89x) with GCC (resp. ICC) on *AVX2 server* to 4.18x (resp. 4.74x) with GCC (resp. ICC) on *AVX-512 server* (see Fig. 2).

Finally, we also consider using OpenMP vectorization for the SIMD FP-based modular arithmetic in our polynomial evaluation. We rely on the new `declare reduction` directive (available since OpenMP 4.0) to instruct the compiler that the final reduction (Line 14 in Algorithm 6) has to be performed using our modular arithmetic. We emphasize that the vectorization is achieved here thanks to this OpenMP directive (along with the directive which instructs the compiler to vectorize the loop with a reduction). Without such directives given by the programmer, the GCC and ICC compilers both manage to vectorize the microbenchmarks presented in Sect. 3.4 (with the same performance as the OpenMP version), but both fail to vectorize the polynomial evaluation code (because of the required specific reductions). As shown in Fig. 2, the ICC OpenMP vectorization is inefficient due to the FMA issue (see Sect. 3.4), whereas the GCC OpenMP vectorization offers computation times somewhat slower than the intrinsic vectorization: 22% slower on *AVX2 server*, and 8% on *AVX-512 server*. It can also be noticed that GCC currently fails to generate align memory accesses (despite the use of the `aligned` OpenMP clause) and SIMD $\oplus_p$ reductions, as we do with intrinsics.

In conclusion, while the use of scalar FP-based modular arithmetic lowers the performance of the polynomial evaluation, the SIMD FP-based modular arithmetic clearly improves its performance (up to 4.74x). In the rest of the article, we will rely on the SIMD implementation with intrinsics, and not on the OpenMP one. This is due to the OpenMP performance issue with ICC and to the somewhat lower performance of the SIMD code generated with OpenMP, especially on AVX2 which still equips the vast majority of available CPUs at the time of writing. We however emphasize that the performance results of OpenMP with GCC on AVX-512 are very promising for the future and show the relevance of this approach.

As a last remark, we recall that regarding the microbenchmarks presented in Sect. 3.3 the AVX-512 $\otimes_p$ and $\oplus_p$ implementations are twice as fast than the AVX2 ones. Here however for the polynomial evaluation, the AVX-512 performance is only 1.57x (resp. 1.47x) faster than the AVX2 one with GCC (resp. ICC). We believe that this is due to the difference in operational intensities. Indeed, the microbenchmarks performed in Sect. 3.3 have been intentionally designed

to be compute-bound in order to measure the number of cycles of the arithmetic operations, and not of the memory accesses. But the operational intensity of our polynomial evaluation is much lower: the Hadamard product and the coefficient reduction correspond to a dot product which is a memory-bound operation in classic floating-point arithmetic. More precisely, the floating-point based modular arithmetic requires 9 flop (floating-point operation) for $\otimes_p$ (see Algorithm 2) and 2 flop for $\oplus_p$ (see Algorithm 3), versus 3 memory accesses for each (2 loads and 1 store, without considering u and p). This makes our compute kernel (i.e. our polynomial evaluation) less memory-bound than a floating-point dot-product, but the operational intensity of our kernel is not high enough to make it compute-bound: memory accesses are still important in the kernel performance. These memory accesses also tend to lower the performance gain due to the increased compute power of the AVX-512 SIMD units, with respect to the AVX2 units, since there is more stress on memory bandwidth with AVX-512 instructions than with AVX2 ones. We will show how to increase this operational intensity and the polynomial evaluation performance in the next section.

4 Increasing the compute efficiency

We now focus on the compute efficiency of our SIMD polynomial evaluation. More precisely, we aim to fill at best the pipelined floating-point units and to minimize the time lost in memory accesses.

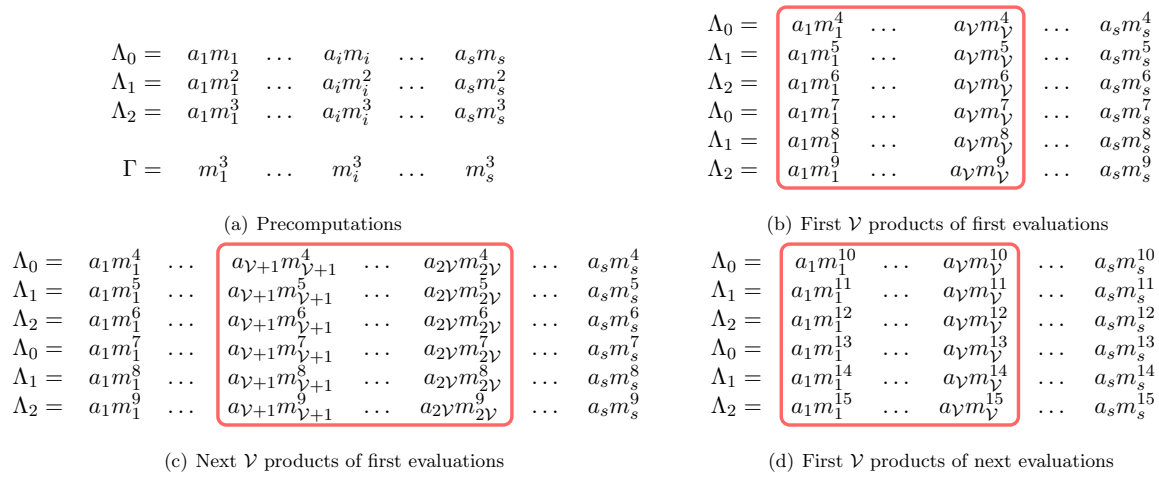
4.1 Multiple dependent evaluations

We first rely on the consecutive powers of β used for the successive evaluations in the matrix method (see Sect. 2.3). Hence in Algorithm 6, if we consider two consecutive polynomial evaluations t and $t+1$, the values computed in the a vector for the evaluation t are re-used as input for evaluation $t+1$. But for large s values (s denoting the number of terms, see Sect. 2.3), the a elements may have been moved out of the CPU caches. We hence consider computing multiple evaluations at a time, and we denote by T_d the number of such (dependent) evaluations. T_d is an algorithmic constant, known at compile time. We can then explicitly avoid storing and reloading data from the vector a to/from memory between these T_d evaluations. We can also load only once m data from memory for these T_d evaluations. Such data reuse increases the operational intensity of our kernel by reducing the number of memory accesses.

However each evaluation depends on the output of the previous one. Even if some operations can be performed concurrently (such as the $\oplus_p$ operation of the t evaluation and the $\otimes_p$ of the $t+1$ evaluation), this dependency limits the instruction-level parallelism, and hence prevents us from filling the pipelines of the floating-point units.

4.2 Multiple independent evaluations

Therefore, we rewrite the loop over the T evaluations (Line 1 in Algorithm 1) in order to have fully *independent* polynomial evaluations. For this purpose, we adapt the algorithm used by Hu and Monagan[5] to introduce thread-level parallelism on multi-core CPUs (see Sect. 2.3.1) in order to introduce here instruction-level parallelism in our compute kernel. Namely, denoting by T_i the desired number of independent evaluations (like T_d, T_i is an algorithmic constant, known at compile time), we first precompute $\Gamma = [m_1^{T_i}, m_2^{T_i}, \ldots, m_s^{T_i}]$ using $O(s \log_2 T_i)$ SIMD multiplications. We also precompute $\Lambda_k = a \circ [m_1^{k+1}, m_2^{k+1}, \ldots, m_s^{k+1}]$ for $0 \leq k < T_i$ using $O(sT_i)$ SIMD multiplications. Then, for the computation of the T evaluations we will first perform the coefficient reductions for the first T_i evaluations (i.e. on $(\Lambda_k)_{0 \leq k < T_i}$), then the



(a) Precomputations

(b) First $\mathcal{V}$ products of first evaluations

(c) Next $\mathcal{V}$ products of first evaluations

(d) First $\mathcal{V}$ products of next evaluations

Figure 3: Illustration of the execution of Algorithm 7 with $T_i = 3$, $T_d = 2$ and using notations of Sect. 2.3. $\mathcal{V}$ is the size of the SIMD vector. For the ease of reading we only represent here the Hadamard products, but the SIMD coefficient reductions are performed alongside, and the $\bar{c}_{k_i,k_d}$ final reductions when required. First (Fig. 3(a)), we precompute Γ and $(\Lambda_k)_{0\leq k<T_i=3}$. Then (Fig. 3(b)), using Γ we compute in SIMD the first $\mathcal{V}$ products of the first $T_i \times T_d = 3 \times 2$ evaluations in $(\Lambda_k)_{0\leq k<3}$. The next $\mathcal{V}$ products are performed in SIMD for the same evaluations (Fig. 3(c)), and so on for the remainings of vectors $(\Lambda_k)_{0\leq k<3}$ and Γ. Once these first evaluations have been fully computed, we start again with the next 3×2 evaluations (Fig. 3(d)), until all evaluations have been fully processed.

Hadamard product $\Lambda_k \leftarrow \Lambda_k \circ \Gamma$ (with $0 \leq k < T_i$) for the second chunk of T_i evaluations. This will be repeated (coefficient reductions on the previous T_i evaluations, then Hadamard product for the next T_i evaluations) until all T evaluations have been processed.

This instruction-level parallelism helps fill the instruction pipelines. Moreover the $\oplus_p$ and $\otimes_p$ operations are now inverted. Contrary to Algorithm 1 (Lines 7-8) where the second operation depends on the output of the first one, the second operation now only depends on the input of the first one. The two operations can thus be more overlapped, hence easing the pipeline filling.

The main drawback of using T_i independent evaluations is the extra memory requirements. For each independent evaluation k we have to store an extra copy Λ_k of the complete a vector. Moreover, for each independent evaluation k we have to load the Λ_k vector from memory and store its update in memory. The operational intensity is thus only improved for the Γ memory accesses. Therefore, introducing an extra independent evaluation increases less the operational intensity than introducing an extra dependent evaluation.

There is hence a trade-off between pipeline filling and operational intensity regarding the numbers of dependent (T_d) and independent (T_i) evaluations. We will thus consider an algorithm where we introduce T_d dependent evaluations for each of the T_i evaluations, hence computing together $T_i \times T_d$ evaluations at a time. The loop over T_d is chosen as the outer one, and the loop over T_i as the inner one: this results in better performance than the opposite loop ordering, which indicates that pipeline filling is here more important than increasing the operational intensity. The optimal values for T_d and T_i depend on the compiler and on the CPU hardware features, and these will have to be determined in practice using parameter testing and tuning. Since the total number of evaluations T is not necessarily a multiple of $T_i \times T_d$, the remaining evaluations are processed first by blocks of $T_i \times N'_d$ evaluations (with $1 \leq N'_d < T_d$) and then with N''_d dependent evaluations (with $N''_d < T_i$), where $T \, mod \, (T_i \times T_d) = T_i \times N'_d + N''_d$.

The final version is presented in Algorithm 7, along with the SIMD programming. Once

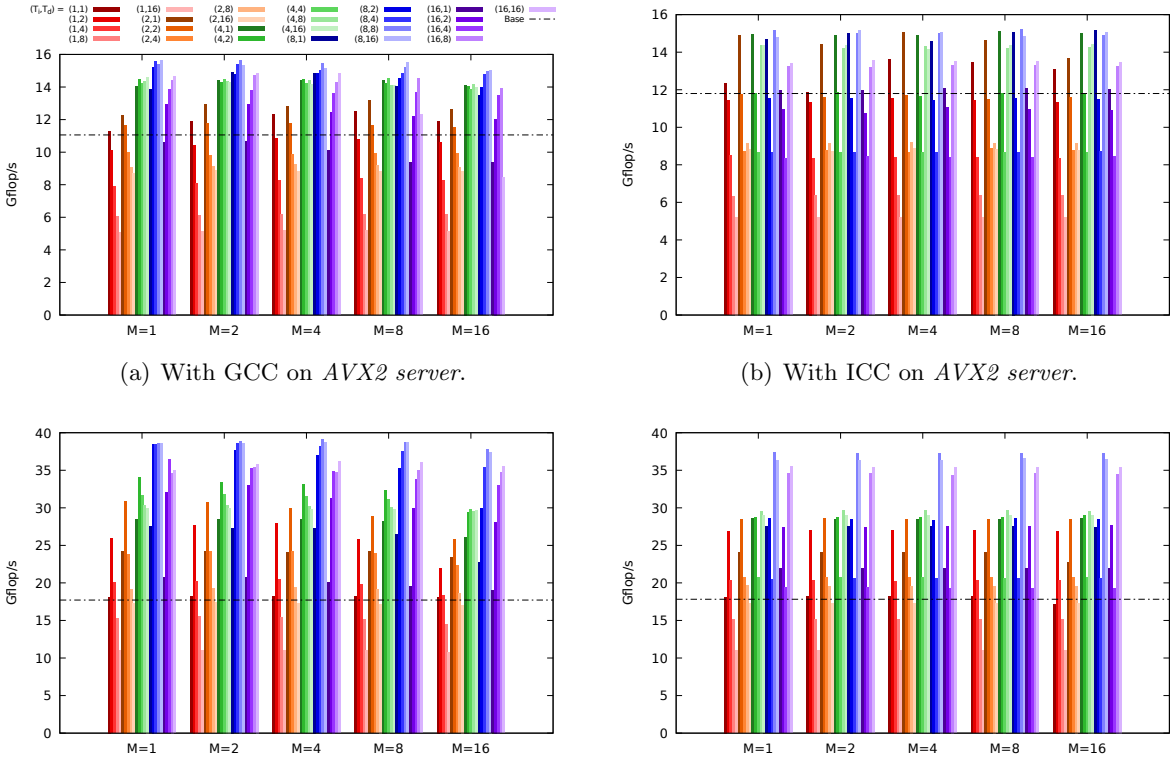
Algorithm 7 SIMD compute kernel of the matrix method with independent and dependent evaluations (see Algorithm 1 for notations and inputs). $\bar{x}$ denotes the SIMD vector corresponding to variable x, and $\bar{x}_i$ the ith SIMD vector. $\mathcal{V}$ is the size of the SIMD vector.

Pre-compute $\Gamma = [m_1^{T_i}, \ldots, m_s^{T_i}]$ and $\Lambda_{k_i} = a \circ [m_1^{k_i+1}, \ldots, m_s^{k_i+1}]$ for $0 \leq k_i < T_i$
(with in-place conversions: 64-bit integers $\rightarrow$ doubles)
for each evaluation $1 \leq t \leq T$ with step $T_i \times T_d$ **do**
  $i \leftarrow 1$; $b_{t+k_i+k_dT_i} \leftarrow 0$, for $(0 \leq k_i < T_i \ ; \ 0 \leq k_d < T_d)$
  while $i \leq s$ **do**
    $\bar{c}_{k_i,k_d} \leftarrow 0.0$, for $(0 \leq k_i < T_i \ ; \ 0 \leq k_d < T_d)$
    $J \leftarrow$ #monomials with same (d_i, e_i)
    for $i \leq j < i + J$ with step $\mathcal{V}$ **do**
      $\bar{\Lambda}_{k_i} \leftarrow \Lambda_{k_i}[j \ .. \ j + \mathcal{V} - 1]$, for $0 \leq k_i < T_i$ ▷ SIMD loads
      $\bar{\Gamma} \leftarrow \Gamma[j \ .. \ j + \mathcal{V} - 1]$ ▷ SIMD load
      for $0 \leq k_d < T_d$ **do**
        for $0 \leq k_i < T_i$ **do**
          $\bar{c}_{k_i,k_d} \leftarrow \bar{c}_{k_i,k_d} \oplus_p \bar{\Lambda}_{k_i}$ ▷ SIMD coefficient reduction
          $\bar{\Lambda}_{k_i} \leftarrow \bar{\Lambda}_{k_i} \otimes_p \bar{\Gamma}$ ▷ SIMD Hadamard product
        end for
      end for
      $\Lambda_{k_i}[j \ .. \ j + \mathcal{V} - 1] \leftarrow \bar{\Lambda}_{k_i}$, for $0 \leq k_i < T_i$ ▷ SIMD stores
    end for
    for $0 \leq k_d < T_d$ **do**
      for $0 \leq k_i < T_i$ **do**
        $c_{k_i,k_d} \leftarrow reduce(\bar{c}_{k_i,k_d}, \oplus_p)$ ▷ $\bar{c}_{k_i,k_d}$ final reduction
        if $c_{k_i,k_d} \neq 0.0$ **then** convert c_{k_i,k_d} to 64-bit integer and
          add $c_{k_i,k_d} x_1^{d_i} x_2^{e_i}$ to bivariate image $b_{t+k_i+k_dT_i}$
      end for
    end for
    $i \leftarrow i + J$
  end while
end for

$T_i \times T_d$ evaluations have been computed together for some monomials, we could choose to iterate over the next $T_i \times T_d$ evaluations or to iterate over the next monomials with same (d_i, e_i) values. If we had iterated over the next $T_i \times T_d$ evaluations, we would have had to store $T_i \times T_d$ SIMD vectors (all $(\bar{c}_{k_i,k_d})_{0 \leq k_i < T_i, \ 0 \leq k_d < T_d}$) for the coefficient reductions. By iterating on the next monomials (as done in Algorithm 7), $T_i + 1$ SIMD loads (all $(\bar{\Lambda}_{k_i})_{0 \leq k_i < T_i}$ and for $\bar{\Gamma}$) and T_i SIMD stores (all $(\bar{\Lambda}_{k_i})_{0 \leq k_i < T_i}$) are required. As $T_d > 2$ in practice (as confirmed for the optimal configurations in Sect. 4.4), it is indeed preferable to iterate over the next monomials in order to minimize the number of memory accesses and hence increase the operational intensity. This results in the end in an algorithm where we browse all the monomials with same (d_i, e_i) values to compute $T_i \times T_d$ evaluations at a time. An illustration of the execution of Algorithm 7 is given in Fig. 3.

4.3 Loop unrolling

Loop unrolling[6] enables us to remove the exit test at the end of the loop body and to interleave instructions from successive loop iterations in order to better fill the pipelines. The two nested



(a) With GCC on *AVX2 server*.

(b) With ICC on *AVX2 server*.

(c) With GCC on *AVX-512 server*.

(d) With ICC on *AVX-512 server*.

Figure 4: Performance results for all possible (T_i, T_d, M) configurations.

loops over the T_d and T_i evaluations (Lines 10 and 11 in Algorithm 7) are hence completely unrolled thanks to the "unroll(F)" pragma of ICC and to the "GCC unroll F" pragma of GCC (recently introduced in GCC 8) to impose an unroll factor of F (F being respectively equal to T_d and T_i). Similarly, we unroll the third loop over the monomials with same (d_i, e_i) values (Line 7 in Algorithm 7) by a factor M. We could also have let the compiler choose which loops to unroll (or not) and determine the best unroll factors. This leads to similar performance with GCC, but to lower performance with ICC (up to 8.5% performance loss). We thus impose our unrollings on the three loops with the corresponding pragmas.

Once the three nested loops have been unrolled, we rely on the compiler and on the out-of-order execution of the processor to schedule at best the instructions to fill the pipelines and to overlap the memory accesses. Other loops over T_d and/or T_i evaluations (Lines 8, 16, 18, 19, in Algorithm 7) are also similarly unrolled.

4.4 Performance results

Using the test platforms (server and compiler) described in Table 1, we present in Figure 4 the performance results for all possible configurations for (T_i, T_d, M), each value ranging in 1,2,4,8,16. We also indicate the performance of the `Base` SIMD code corresponding to the version obtained with the SIMD intrinsics only (as presented in Sect. 3). The performance varies significantly depending on the (T_i, T_d, M) values, especially on T_i and T_d, which shows the relevance of these parameters. The performance impact of M is lower but can still reach 11% for some (T_i, T_d) configurations. The best configurations are the following.

- ($T_i = 8$, $T_d = 16$, $M = 1$) with GCC on *AVX2 server*: 15.65 Gflop/s, and 42% of

Table 2: Performance comparison between our best version and the reference implementation using integer-based arithmetic.

Server	Compiler	Reference scalar integer-based version (time in ms)	SIMD FP-based version with improved compute efficiency (time in ms)	Gain
AVX2 server	GCC	15262	3482	4.4x
AVX2 server	ICC	17704	3671	4.8x
AVX-512 server	GCC	12984	1411	9.2x
AVX-512 server	ICC	14638	1476	9.9x

performance gain over the `Base` SIMD code.

- ($T_i = 8, T_d = 8, M = 8$) with ICC on *AVX2 server*: 15.21 Gflop/s, and 29% of performance gain over the `Base` SIMD code.
- ($T_i = 8$, $T_d = 8$, $M = 4$) with GCC on *AVX-512 server*: 39.11 Gflop/s, and 121% of performance gain over the `Base` SIMD code.
- ($T_i = 8$, $T_d = 8$, $M = 1$) with ICC on *AVX-512 server*: 37.31 Gflop/s, and 109% of performance gain over the `Base` SIMD code.

As determining the theoretical peak performance of modern CPUs becomes more and more complicated[35], we use the BLAS DGEMM routine of the Intel MKL[4] to estimate the single-core double-precision peak performance at 45 Gflop/s on *AVX2 server* and 101 Gflop/s on *AVX-512 server*. Moreover, we can only reach 61% of the peak performance since there are only 2 FMAs out of the 9 floating-point instructions required for $\otimes_p$ and $\oplus_p$. We manage hence to reach 57% and 63% of the attainable single-core peak performance, respectively on *AVX2 server* and on *AVX-512 server*.

In the end, as shown in Table 2, we manage to reach speedups of almost 5x and 10x (respectively on *AVX2 server* and on *AVX-512 server*) on one CPU core over the reference original polynomial evaluation with scalar integer-based modular arithmetic. It can be noticed that Monagan and coworkers already used to process two (dependent) evaluations at a time to increase the operational intensity for some variants of the polynomial evaluation. However for the variant studied in this article (see Algorithm 1), which is the fastest one, no performance gain is obtained by processing two evaluations at a time with the original scalar code. Such divergence with respect to the gains obtained in Fig. 4 can be explained by the lack of other optimizations (multiple independent evaluations, loop unrollings) as well as by the differences in the modular arithmetic implementation between the original integer-based version by Monagan and coworkers and the floating-point based version of this article.

4.5 Without extra memory requirements

One drawback of using multiple independent evaluations is the significant memory overhead: the complete a vector has to be duplicated for each extra independent evaluation. We therefore investigate here the best attainable performance without any extra independent evalution.

We first implement a code without multiple independent evaluation, and tune the T_d and M parameters for this code via extensive benchmarks (as in Sect. 4.4). Figure 5 shows for each

[4]See: `https://software.intel.com/en-us/mkl`

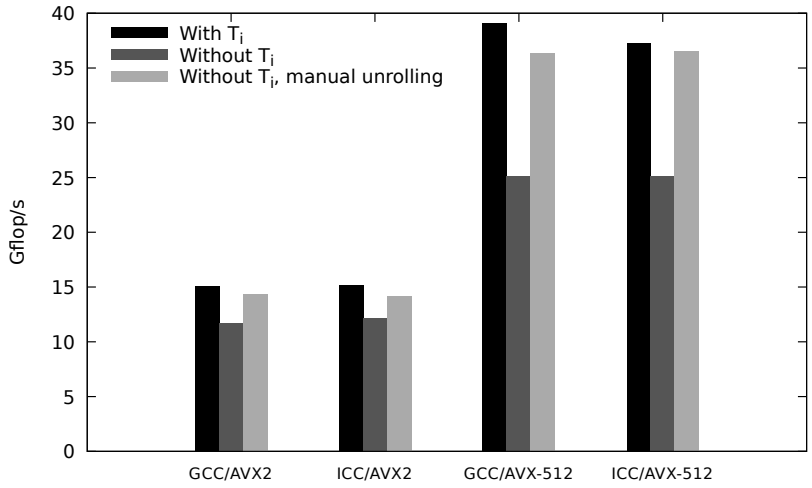


Figure 5: Performance comparison between best configuration with extra memory for multiple independent evaluations (`with` T_i) and best configurations without extra memory for multiple independent evaluations (`without` T_i), with and without manual loop unrolling.

test platform the performance drop obtained for this code (referred to as `Without` T_i) with respect to the best version obtained in Sect. 4.4 (referred to as `With` T_i). The performance drop is important here (up to 36%), due to the lower number of independent instructions to fill the pipelines.

We then introduced manual loop unrolling, using preprocessor macros to ease and automate the tedious code writing. We also manually group all arithmetic instructions. This way, we provide all arithmetic instructions for the computation of T_d dependent evaluations and $M \times \mathcal{V}$ monomials to the compiler and to the out-of-order execution of the processor, so that these can be scheduled at best to fill the pipelines. One can see that this new version (referred to as: `Without` T_i`, manual unrolling`; and after tuning of its T_d and M parameters) greatly reduces the performance drop with respect to the best version obtained in Sect. 4.4 (`With` T_i). This way, we can reach 95% (GCC) and 94% (ICC) on *AVX2 server* and 93% (GCC) and 98% (ICC) on *AVX-512 server* of the best attainable performance (`With` T_i). At the price of non-negligeable development efforts, we can thus obtain, without introducing extra memory, almost the same performance of our best versions with multiple independent evaluations.

It can also be noticed that the performance impact of M is here much more important than in Sect. 4.4 (detailed tests not shown). Such manual loop unrolling and instruction grouping have also been tried on the best version obtained in Sect. 4.4 (with multiple independent evaluations): this however only offers up to 5.1% performance gain for such code. In our opinion, this does not justify the manual unrolling development effort when using multiple independent evaluations.

5 Conclusion

In this article, we have first justified the choice of a modular multiplication algorithm relevant for HPC and SIMD computing. We have ensured the correct use of an optimized AVX2 implementation (regarding a potential issue with signed zeros and the `blendv_pd` intrinsic) and we have presented its AVX-512 version. This floating-point (FP) based algorithm with FMAs (fused multiply-adds) enables us to obtain SIMD speedups of up to 3.7x on AVX2, and up to 7.2x on AVX-512, which validates its efficiency. With respect to a reference (scalar) integer-

based modular arithmetic, the performance gains are similar for our SIMD FP-based modular multiplication and for the corresponding SIMD FP-based modular addition. As all current desktop and HPC processors have SIMD units, we believe that such SIMD FP-based modular arithmetic should from now on be used instead of the scalar ones. Using OpenMP for their SIMD programming turned out to be a very promising approach on the new AVX-512 units (with GCC), due to its very relevant performance-programmability trade-off. Currently, we still rely on intrinsics programming for best performance and performance portability among compilers.

In a second part, we have focused on the partial polynomial evaluation which is a key computation in Computer Algebra. We have rewritten this algorithm in order to introduce multiple independent and dependent evaluations. These enable us, along with loop unrolling, to fill at best the pipelined floating-point units of the CPU and to minimize the time lost in memory accesses. Combined with SIMD computing, we achieve speedups up to almost 5x on AVX2 and up to almost 10x on AVX-512 with respect to the reference implementation of the polynomial evaluation. Moreover, using manual loop unrolling we manage to closely reach such performance gains without extra memory requirements.

In the future, we plan to integrate our efficient polynomial evaluation on one CPU core in the multi-core parallel implementation of Monagan and coworkers[5, 17], and to study the performance impact on polynomial factorizations and polynomial greatest common divisor computations. We also believe that GPUs may be well suited to further accelerate our polynomial evaluation thanks to their higher compute power and memory bandwidth. We emphasize that our FP-based modular arithmetic will be very relevant for the GPU FMA SIMD units, and will offer a direct and efficient implementation of modular arithmetic on GPUs. We may also investigate using a few less bits for our prime p in order to decrease the number of reductions as done for example with error-free transformations in linear algebra[36].

Acknowledgments

The authors would like to thank the master in computer science at Sorbonne Université, especially N. Picot and P. Cadinot, for administering and providing access to the compute servers. They also thank Professor S. Graillat (Sorbonne Université) for helpful discussions on error-free transformations.

References

[1] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley, Boston, MA, USA, 1997.

[2] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Springer, 1992.

[3] Joachim Von Zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, USA, 2 edition, 2003.

[4] Michael Monagan and Baris Tuncer. The complexity of sparse Hensel lifting and sparse polynomial factorization. *Journal of Symbolic Computation*, 99:189 – 230, 2020.

[5] Jiaxiong Hu and Michael Monagan. A Fast Parallel Sparse Polynomial GCD Algorithm. Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, pages 271–278, New York, NY, USA, 2016. ACM.

[6] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, Sixth Edition*. The Morgan Kaufmann Series in Computer Architecture and Design, 2017.

[7] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[8] Intel Developer Services. MMX Technology Technical Overview, 1996.

[9] AMD. 3DNow! Technology Manual, 2000.

[10] Intel. Intel SSE4 Programming Reference, Reference number: D91561-003, 2007.

[11] K. Diefendorf. Altivec extension to Power PC accelerates media processing, 2001.

[12] Intel. Intel Architecture Instruction Set Extensions Programming Reference, Number: 319433-012A, 2012.

[13] Intel. Intel Architecture Instruction Set Extensions Programming Reference, Number: 319433-024, 2016.

[14] Joris Van Der Hoeven, Grégoire Lecerf, and Guillaume Quintin. Modular SIMD arithmetic in Mathemagix. *ACM Trans. Math. Softw.*, 43(1):5:1–5:37, August 2016.

[15] Richard E. Zippel. Probabilistic algorithms for sparse polynomials. Symbolic and Algebraic Computation, EUROSAM '79, pages 72:216–226, Berlin, Heidelberg, 1979. Springer.

[16] Richard E. Zippel. Interpolating polynomials from their values. *Journal of Symbolic Computation*, 9(3):375 – 403, 1990.

[17] Michael Monagan and Baris Tuncer. Sparse multivariate polynomial factorization: a high-performance design and implementation. Mathematical Software – ICMS 2018, pages 359–368, Cham, 2018. Springer International Publishing.

[18] Michael Ben-Or and Prasoon Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 301–309, New York, NY, USA, 1988. Association for Computing Machinery.

[19] Daniel S. Roche. What can (and can't) we do with sparse polynomials? In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '18, pages 25–30, New York, NY, USA, 2018. ACM.

[20] Jiaxiong Hu and Michael Monagan. A fast parallel sparse polynomial GCD algorithm. *Journal of Symbolic Computation*, 2019 (submitted).

[21] Joris Van Der Hoeven and GréGoire Lecerf. On the Bit-complexity of Sparse Polynomial and Series Multiplication. *Journal of Symbolic Computation*, 50:227–254, March 2013.

[22] Michael Monagan and Alan Wong. Fast Parallel Multi-point Evaluation of Sparse Polynomials. Proceedings of the International Workshop on Parallel Symbolic Computation, pages 4:1–4:7, New York, NY, USA, July 2017. ACM.

[23] Agner Fog. Optimizing software in c++. an optimization guide for windows, linux and mac platforms. Technical report, Technical University of Denmark, 2018. `https://www.agner.org/optimize/`.

[24] R. Alverson. Integer division using reciprocals. Proceedings 10th IEEE Symposium on Computer Arithmetic, pages 186–190, June 1991.

[25] Henry G. Baker. Computing a*b (mod n) efficiently in ansi c. *SIGPLAN Not.*, 27(1):95–98, January 1992.

[26] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. Advances in Cryptology — CRYPTO' 86, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[27] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pages 61–72, New York, NY, USA, 1994. ACM.

[28] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[29] N. Moller and T. Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, Feb 2011.

[30] Matthew Gibson and Michael Monagan. Optimizing and Parallelizing the Modular GCD Algorithm. In *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*, PASCO '15, pages 44–52, New York, NY, USA, 2015. ACM.

[31] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, June 2005.

[32] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[33] Gabriell Orisaka, Julio López, and Diego F. Aranha. Finite field arithmetic using avx-512 for isogeny-based cryptography. XVIII Simpósio Brasileiro de Segurança da Informação e Sistemas Computacionais (SBSeg 2018), pages 49–56, 2018.

[34] Marshall Law and Michael Monagan. A parallel implementation for polynomial multiplication modulo a prime. Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, pages 78–86, New York, NY, USA, 2015. ACM.

[35] Romain Dolbeau. Theoretical peak FLOPS per instruction set: a tutorial. *The Journal of Supercomputing*, 74(3):1341–1377, Mar 2018.

[36] J. Jean and S. Graillat. A parallel algorithm for dot product over word-size finite field using floating-point arithmetic. 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pages 80–87, 2010.