Multiword matrix multiplication over large finite fields in floating-point arithmetic

Jérémy Berthomieu, Stef Graillat, Dimitri Lesnoff, Theo Mary

MULTIWORD MATRIX MULTIPLICATION OVER LARGE FINITE FIELDS IN FLOATING-POINT ARITHMETIC*

JÉRÉMY BERTHOMIEU†, STEF GRAILLAT†, DIMITRI LESNOFF†, AND THEO MARY†

Abstract. This article is concerned with the efficient computation of modular matrix multiplication $C = AB \bmod p$, a key kernel in computer algebra. We focus on floating-point arithmetic, which allows for using efficient matrix multiplication libraries. However, the existing approach is limited to primes p with bitsize at most half the mantissa size (e.g., 26 bits with double precision arithmetic), and becomes quite inefficient when p approaches this limit. We present a new approach that overcomes this limitation and can efficiently handle primes with larger bitsizes. The key idea is to use multiword decompositions $A = \sum_{i=0}^{u-1} \alpha^i A_i$ and $B = \sum_{j=0}^{v-1} \beta^j B_j$, which represent A and B as the scaled sum of u and v matrices (words) A_i and B_j with smaller coefficients. The product C can then be reconstructed by computing uv modular products $A_iB_j \bmod p$. We provide a rigorous analysis that proves the correctness of this approach for suitably chosen scaling parameters α and β. Our analysis determines the maximum bitsize of p that can be handled for a given (u, v) decomposition; in particular, we show that using a $(2, 2)$ decomposition suffices to handle bitsizes almost equal to the full mantissa size (e.g., the 26 bits limit is raised to 51 bits in double precision arithmetic). Moreover, we show that $(1, v)$ decompositions with $v > 1$ are also of interest to handle intermediate bitsizes. We perform an extensive experimental analysis for various matrix shapes and prime bitsizes. Our performance benchmarks on both CPU and GPU architectures confirm the efficiency of the proposed approach, which can outperform the existing single word approach for bitsizes as low as 23, and can handle bitsizes as high as 52 while retaining high performance.

Key words. matrix multiplication, multiword decomposition, modular arithmetic, finite fields, floating-point arithmetic, CPU, GPU, high-performance computing, rounding error

AMS subject classifications. 65Y05, 65Y20, 65F99, 65G50

1. Introduction.

We are interested in efficiently computing the modular matrix product

$$C = AB \bmod p, \tag{1.1}$$

where $p \in \mathbb{N}$ is prime, which is a key kernel in computer algebra problems. Indeed, solving computer algebra problems requires efficient yet exact linear algebra operations on rational numbers, such as matrix inversion, linear system solving, PLUQ factorization, echelon form, characteristic or minimal polynomial. A direct computation with rationals is infeasible due to the growth of intermediate coefficients. To circumvent this issue, computations are done over a finite field of modular integers $\mathbb{Z}/p\mathbb{Z}$, and the exact solution is reconstructed using, for example, the Chinese remainder theorem. Moreover, this reconstruction has a chance of not being valid for some values of p, so it is desirable to handle values as large as possible to minimize this chance. Therefore, in this article, we aim to efficiently compute (1.1) for large values of p.

To this purpose, most computer algebra systems implement elementary arithmetic operations and linear algebra subroutines over finite fields, see for example FLINT [9], NTL [13] and FFLAS/Linbox [8]. These libraries use either integer or floating-point arithmetic to represent finite field elements. FLINT adopts a GMP-based integer approach [7], while NTL implements its own arbitrary-precision integer arithmetic. In contrast, FFLAS uses floating-point arithmetic, which generally provides better

*Version of January 28, 2025.

†Sorbonne Université, CNRS, LIP6, F-75005, Paris, France
(jeremy.berthomieu@lip6.fr, stef.graillat@lip6.fr, dimitri.lesnoff@lip6.fr, theo.mary@lip6.fr)

performance due to the availability of SIMD (Single Instruction, Multiple Data) instructions, such as SSE, AVX, and FMA, and can take advantage of the BLAS (Basic Linear Algebra Subprograms) libraries, which are highly optimized on modern CPUs and GPUs. However, current floating-point approaches are limited by the restriction to finite fields with prime moduli smaller than 2^{26}, which corresponds to half the mantissa bitsize in double-precision arithmetic. For primes larger than 2^{26}, these libraries switch to (possibly arbitrary-precision) integer arithmetic, which is slower and lacks the same level of hardware acceleration available to floating-point operations.

In this article, we propose new matrix multiplication algorithms that are able to overcome this limitation and therefore enable more efficient modular matrix operations, fully leveraging the performance potential of multicore CPUs and GPUs. Our new algorithms are able to handle larger primes, from half all the way to the full mantissa bitsize. The key idea to achieve this is to use the matrix multiword decompositions

$$A = \sum_{i=0}^{u-1} \alpha^i A_i, \qquad B = \sum_{j=0}^{v-1} \beta^j B_j, \tag{1.2}$$

for which (1.1) becomes

$$C = \sum_{i=0}^{u-1} \sum_{j=0}^{v-1} \alpha^i \beta^j A_i B_j \bmod p. \tag{1.3}$$

With a suitable choice of the scaling parameters α and β, the coefficients of matrices A_i and B_j can be made sufficiently small so that the products $A_i B_j \bmod p$ can be efficiently computed with classical floating-point modular matrix multiplication algorithms. We describe how to compute the decompositions (1.2) and the product (1.3) in floating-point arithmetic, and we carry out a rigorous analysis to determine how to choose α and β and to prove the correctness of the algorithms. In particular, we determine the maximum size of p that can be handled depending on the number of words u and v. This allows for adaptively selecting u and v based on the size of p, and thus to optimize the cost of the algorithm which is proportional to uv. We also present a concatenated variant of the algorithm that stacks together the B_j (respectively A_i) matrices to increase the arithmetic intensity of the product, and is particularly efficient when B (respectively A) is a tall-and-skinny (respectively short-and-wide) matrix. We implement the proposed algorithms on both multicore CPU and GPU architectures, and perform numerical experiments that confirm their ability to handle primes as large as 2^{52} while retaining high performance.

The rest of this article is organized as follows. We first describe in Section 2 the existing single word algorithm and its limitations. We then propose the new multiword algorithms in Section 3. We report our numerical experiments in Section 4. Finally, we provide some concluding remarks in Section 5.

2. Existing single word algorithm and its limitations. Throughout this article, we consider computations on integers using a floating-point arithmetic with t bits of significand; for IEEE double precision, $t = 53$. We define $\mathbb{F}$ the set of floating-point numbers that are nonnegative integers: this set certainly includes all integers x such that $0 \leq x \leq 2^t$. We also define $\mathbb{F}_p$ the set of floating-point numbers that are integers modulo $p \in \mathbb{N}$, that is, less than p. For the entirety of the article, we assume that $t \geq 3$ and $p \geq 5$.

We denote by $\mathrm{fl}(\cdot)$ the result of a floating-point computation, where all operations inside parentheses are done in floating-point working precision. We recall that floating-point operations in the IEEE 754 standard satisfy, in absence of underflow or overflow,

$$\mathrm{fl}(a \operatorname{op} b) = (a \circ b)(1+\eta), \quad |\eta| \leq \epsilon/(1+\epsilon), \quad \operatorname{op} \in \{+,-,\times,/\}, \tag{2.1}$$

where $\epsilon = 2^{-t}$ is the unit roundoff.

2.1. Modular reductions in floating-point arithmetic. Computing exactly with finite fields elements using floating-point arithmetic requires defining an efficient modulo operator similar to the predefined operator for integer types. Each element can be reduced using the `FMOD` instruction where $\texttt{FMOD}(x,y) = \mathrm{fl}\left(x - \left\lfloor \frac{x}{y} \right\rfloor y\right)$. In a finite field, we always reduce by the same modulus $y = p$ and so we may precompute its floating-point inverse $q = \mathrm{fl}(1/p)$. This yields Algorithm 2.1.

Algorithm 2.1: Floating-point reduction

Input : $x \in \mathbb{F} < 2^t$, a prime number $p \in \mathbb{F} < 2^{t-1}$, and $q = \mathrm{fl}(1/p)$.
Output: $d \in \mathbb{F}_p$ such that $d = x \bmod p$.

$b = xq$
$c = \lfloor b \rfloor$
$d = \texttt{fma}(-c, p, x)$ `//` $x - cp$
if $d \geq p$ **then**
  $d = d - p$
if $d < 0$ **then**
  $d = d + p$

PROPOSITION 2.1. *Algorithm 2.1 is correct for any integer input $x \in \mathbb{F}$ and a modulus p such that $4 \leq p < 2^{t-1}$ and $x < 2^t$.*

Proof. As $q = \mathrm{fl}(1/p)$ it follows from (2.1) that $q = (1/p)(1+\eta_1)$ with $|\eta_1| \leq \epsilon/(1+\epsilon)$. Similarly as $b = \mathrm{fl}(xq)$, we have that $b = (xq)(1+\eta_2) = (x/p)(1+\eta_1)(1+\eta_2)$ with $|\eta_2| \leq \epsilon/(1+\epsilon)$. Approximation terms can be merged into one since $b = (x/p)(1+\eta)$ with $\eta = \eta_1 + \eta_2 + 2\eta_1\eta_2$ and $|\eta| \leq 2\epsilon$. As by hypothesis, $x \leq 2^{t-2}p$, we have: $b \leq (x/p)(1+2\epsilon) < (1+2\epsilon)2^{t-2} < 2^t$. As $b < 2^t$, its integer part can be stored as a floating-point number. As a consequence, $c = \lfloor b \rfloor$. By definition of the Euclidean division of x by p, there exist some integers q and r such that $x = qp + r$ with $0 \leq r < p$. It follows that $b = q(1+\eta) + (r/p)(1+\eta)$ which can be written as:

$$b = q + \underbrace{q\eta + (r/p)(1+\eta)}_{\gamma}. \tag{2.2}$$

We can deduce that $\gamma \leq 2\epsilon q + (r/p)(1+2\epsilon)$. As $r < p$ and $q \leq x/p \leq 2^{t-2}$ then $2\epsilon q \leq 1/2$ and so $\gamma \leq 3/2 + 2\epsilon < 2$ as long as $t \geq 3$. Moreover $\gamma \geq -2\epsilon q + (r/p)(1-2\epsilon)$ and so similarly $\gamma \geq -1/2 - 2\epsilon > -1$. We can conclude that b belongs to the interval $]q-1, q+2[$ and so $c = \lfloor b \rfloor \in \{q-1, q, q+1\}$. Let us now verify that $x - cp < 2^t$ and so is exactly representable by a floating-point number. If $c = q$ then $x - cp = r < p < 2^t$. If $c = q-1$ then $x - cp = p + r \leq 2p - 1 < 2^t$. Finally if $c = q+1$ then $x - cp = r - p$ so $|x - cp| \leq p < 2^t$. ∎

In some cases we need to reduce the product of two integers whose result would overflow before reduction, that is, be larger than 2^t and thus not necessarily in $\mathbb{F}$.

Algorithm 2.2: Modular product reduction [14, Function 3.6].

Input : $x \in \mathbb{F}$ and $y \in \mathbb{F}$ satisfying $xy \leq 2^{t-2}p$, a prime number $p \in \mathbb{F} \leq 2^{t-1}$ and $q = \mathrm{fl}(1/p)$
Output: $d \in \mathbb{F}$ such that $d = xy \bmod p$.

$h = \mathrm{fl}(xy)$
$l = \texttt{fma}(x, y, -h)$ // $xy - h$
$b = \mathrm{fl}(h/p)$
$c = \lfloor b \rfloor$
$d = \texttt{fma}(-c, p, h)$ // $h - cp$
if $d \geq p$ **then**
  $d = d - p$
if $d < 0$ **then**
  $d = d + p$

Assuming a fused multiply-add `fma` instruction is available, these cases can be handled with Algorithm 2.2 [14, Function 3.6]. The next result is a slightly more general version of [14, Proposition 3.7].

PROPOSITION 2.2. *Algorithm 2.2 is correct for integer input x and y in $\mathbb{F}$ such that their product satisfies $xy \leq 2^{t-2}p$ and for input $p \leq 2^{t-1}$.*

Proof. Using error-free transformation and `fma`, it is shown in [11, 12] that $h+l = xy$ with $|l| \leq \epsilon|xy|$. As $xy \leq 2^{t-1}p$, it follows that $|l| \leq p/2$. By definition of h and b, we have $h = xy(1+\eta_1)$ and $b = (h/p)(1+\eta_2)$ with $|\eta_1|, |\eta_2| \leq \epsilon/(1+\epsilon)$ so that $h \leq (1+\epsilon/(1+\epsilon))xy$ and $b \leq (1+\epsilon/(1+\epsilon))(h/p)$. As a consequence, $b \leq (1+2\epsilon)xy/p < 2^t$ so that b is representable with a floating-point number and finally $c = \lfloor b \rfloor$.

Let us now write down the Euclidean division of xy by p. By definition there exist some integers q and r such that $xy = qp + r$ with $0 \leq r < p$. It follows that $(1+\eta_1)xy = (1+\eta_1)qp + (1+\eta_1)r$ and so $h = (1+\eta_1)qp + (1+\eta_1)r$. This can be written as $h/p = (1+\eta_1)q + (1+\eta_1)r/p$ and so $h/p(1+\eta_2)$. We then have that $b = (1+\beta)q + (1+\beta)r/p$ with $|\beta| \leq 2\epsilon$ which can be written as

$$b = q + \underbrace{q\beta + (r/p)(1+\beta)}_{\alpha}.$$

We can deduce that $\alpha \leq 2\epsilon q + (r/p)(1+2\epsilon)$. As $r < p$ and $q \leq xy/p \leq 2^{t-2}$ then $2\epsilon q \leq 1/2$ and so $\alpha \leq 3/2 + 2\epsilon < 2$ since $t \geq 3$ by assumption on p. Moreover $\alpha \geq -2\epsilon q + (r/p)(1-2\epsilon)$ and so similarly $\alpha \geq -1/2 - 2\epsilon > -1$.

We can conclude that b belongs to the interval $]q-1, q+2[$ and so $c = \lfloor b \rfloor \in \{q-1, q, q+1\}$. ∎

2.2. Block matrix product. Once we have defined a modulo operator using floating-point arithmetic, modular matrix multiplication can be naively implemented by simply performing a reduction after each floating-point operation to ensure the size of the integers remain bounded: given $A \in \mathbb{F}^{m\times k}$ and $B \in \mathbb{F}^{k\times n}$, $C = AB \in \mathbb{F}^{m\times n}$ can be computed as

$$C \leftarrow C + (a_j b_j^T \bmod p) \bmod p, \quad j = 1\colon k, \tag{2.3}$$

where a_j is the jth column of A and b_j^T is the jth row of B.

This approach is however extremely inefficient since it requires as many reductions as floating-point operations. The number of reductions can be reduced by computing instead

$$C \leftarrow C + (A_j B_j \bmod p) \bmod p, \quad j = 1 \colon \lceil k/\lambda \rceil, \tag{2.4}$$

where $A_j \in \mathbb{F}^{m \times \lambda}$ and $B_j \in \mathbb{F}^{\lambda \times n}$ are block-columns of A and block-rows of B, respectively, and where λ is a block size that controls how often the reductions are performed. When choosing the value of λ we must ensure that the intermediate computations do not reach the range at which integers are approximated when written as a floating-point (numbers x with exponent e strictly greater than t such that $x \not\equiv 0 \bmod 2^{e-t}$). Assuming that the coefficients of A and B are in $\mathbb{F}_p$ (that is, they are already reduced modulo p), then the coefficients of $A_j B_j$ are bounded by $\lambda(p-1)^2$ and so it suffices to take $\lambda = \lfloor 2^t/(p-1)^2 \rfloor$. This approach is for example implemented in the FFLAS library [4].

To perform the inner reduction in (2.4), the result of $A_j B_j$ must be stored in a temporary workspace. To avoid this additional workspace, one can remove this inner reduction provided that the coefficients of $C + A_j B_j$ remain representable at all steps j of the computation. Then (2.4) becomes

$$C \leftarrow C + A_j B_j \bmod p, \quad j = 1 \colon \lceil k/\lambda \rceil, \tag{2.5}$$

Algorithm 2.3 implements this latter approach.

Algorithm 2.3: Block matrix product over $\mathbb{F}_p$

Input : $A \in \mathbb{F}^{m \times k}$, $B \in \mathbb{F}^{k \times n}$, $C \in \mathbb{F}_p^{m \times n}$, and a block size λ satisfying Proposition 2.3.
Output: $C = C + AB \bmod p \in \mathbb{F}_p^{m \times n}$.

```
for j = 1 to ⌈k/λ⌉ do
    C = C + A_j B_j    // A_j, B_j submatrices of size m × λ and λ × n
    C = C mod p        // Using Algorithm 2.1
```

Computationally, Algorithm 2.3 is attractive because it mainly relies on the efficient matrix products $A_j B_j$. Indeed, it performs $2mkn$ floating-point operations (flops) for the matrix products and only $mn\lceil k/\lambda \rceil$ reductions, whose cost is thus negligible for a sufficiently large block size λ. It is therefore crucial to determine the largest possible λ such that the algorithm remains correct.

PROPOSITION 2.3. *Algorithm 2.3 is correct for input matrices A, B, a prime number $p < 2^{t-1}$, and a block size λ such that*

$$\lambda \max(A) \max(B) + p - 1 \leq 2^t, \tag{2.6}$$

where the operator $\max(\cdot)$ *returns the maximum coefficient of a matrix.*

Proof. At each iteration of the `for` loop, each coefficient of $A_j B_j$ is computed as the dot product of vectors of size at most λ and is thus bounded by $\lambda \max(A) \max(B)$. Then, it is added to a coefficient of C, which is bounded by $p - 1$ since C is reduced modulo p at each iteration. The result is thus exact as long as the coefficients of $C + A_j B_j$ and p match the conditions of Algorithm 2.1 on x and p, that is, as long as (2.6) holds and $p < 2^{t-1}$. ∎

Algorithm 2.3 is classically used with $C = 0 \in \mathbb{F}_p^{m\times n}$ and with A and B with coefficients in $\mathbb{F}_p$ [4]. In this case, since $\max(A)$ and $\max(B)$ are both bounded by $p-1$, (2.6) rewrites as $\lambda(p-1)^2 + p - 1 \leq 2^t$, which holds for

$$\lambda = \left\lfloor \frac{2^t - p + 1}{(p-1)^2} \right\rfloor. \tag{2.7}$$

This provides a sufficient condition on the maximum size of p.

COROLLARY 2.4. *Calling Algorithm 2.3 on $A \in \mathbb{F}_p^{m\times k}$, $B \in \mathbb{F}_p^{k\times n}$, $C = 0 \in \mathbb{F}_p^{m\times n}$ and block-size λ satisfying (2.7) correctly returns $AB \in \mathbb{F}_p^{m\times n}$ if*

$$p \leq 2^{t/2}. \tag{2.8}$$

Proof. The result is correct if $\lambda \geq 1$, that is, if $(p-1)^2 + p - 1 \leq 2^t$. Since $(p-1)^2 + p - 1 = p(p-1)$, (2.8) is certainly sufficient. ∎

With double precision arithmetic ($t = 53$), Algorithm 2.3 can thus only handle prime numbers with at most 26 bits. Moreover, for prime numbers approaching this limit, the algorithm becomes quite inefficient since it must use a small block size λ.

In the next section we propose a new approach based on multiword arithmetic that can handle much larger primes.

3. New multiword algorithms. To overcome the limitations of the existing block matrix product algorithm, we propose instead to rely on multiword arithmetic, which consists in splitting the numbers into smaller parts, called *words*, which can be stored with a smaller precision (with fewer bits). Multiword matrix multiplication algorithms are well studied in inexact floating-point arithmetic, and have generated a renewed interest due to their ability to emulate high precision arithmetic while exploiting efficient mixed precision GPU hardware [5]. However, to the best of our knowledge, using multiword arithmetic for modular integer computations (though still based on floating-point arithmetic) is a new idea, which we develop in the rest of this section.

3.1. Multiword matrix decomposition. Given $M \in \mathbb{F}_p^{m\times n}$ with coefficients bounded by p, we seek to decompose it as the unevaluated sum of u words M_i:

$$M = \sum_{i=0}^{u-1} \alpha^i M_i,$$

where to balance the coefficients of M_i and make them as small as possible, we should take $\alpha \approx p^{1/u}$. If s bits are required to store the coefficients of M, about s/u bits should be sufficient to store those of M_i. Algorithm 3.1 describes a method to obtain such a decomposition using only floating-point arithmetic.

PROPOSITION 3.1. *Assuming $1 < p < 2^t$ and $u \leq t$, Algorithm 3.1 computes exactly the decomposition*

$$M = \sum_{i=0}^{u-1} \alpha^i M_i \tag{3.1}$$

where each matrix M_i has nonnegative coefficients bounded by

$$c = (\alpha + 1)(1 + \epsilon)^{u-1} \leq 2^t,$$

where $\alpha = \lceil p^{1/u} \rceil$ and $\epsilon = 2^{-t}$.

Algorithm 3.1: Multiword matrix decomposition

Input : $M \in \mathbb{F}_p^{m\times n}$ and the number of words u.

Output: $\alpha \in \mathbb{N}$ and $M_0, \dots, M_{u-1} \in \mathbb{F}_p^{m\times n}$ such that $M = \sum_{i=0}^{u-1} \alpha^i M_i$.

$\alpha = \lceil p^{1/u} \rceil$
$T = M$
for $i = 0$ **to** $u-2$ **do**
  $R = \lfloor \frac{T}{\alpha} \rfloor$ `// Quotient`
  $M_i = T - \alpha R$ `// Remainder`
  $T = R$
$M_{u-1} = T$

Proof. In addition to M_i, we denote as T_i and R_i the values that T and R take at the end of iteration i of the `for` loop, with the notation $T_{-1} = M$. Our goal is to bound the coefficients of these matrices and check that no overflow occurs during any step of the computation. Note first that since $p > 1$ we have $\alpha \geq 2$. Let us first bound $R_i = \left\lfloor \mathrm{fl}(\frac{T_{i-1}}{\alpha}) \right\rfloor$ from above. Using (2.1), we have

$$R_i = \left\lfloor \mathrm{fl}(\frac{T_{i-1}}{\alpha}) \right\rfloor = \left\lfloor \frac{T_{i-1}}{\alpha}(1+\eta) \right\rfloor, \quad |\eta| \leq \epsilon, \tag{3.2}$$

where $\epsilon = 2^{-t}$ is the unit roundoff of the floating-point arithmetic. We therefore have the bound

$$R_i \leq \left\lfloor (1+\epsilon)\frac{T_{i-1}}{\alpha} \right\rfloor \leq (1+\epsilon)\frac{T_{i-1}}{\alpha}. \tag{3.3}$$

Since $\alpha \geq 2 \geq (1+\epsilon)$ we have $R_i \leq R_{i-1}$, which means that the coefficients of R decrease throughout the iterations. Thus for any i we have

$$\alpha R_i \leq \alpha R_0 \leq (1+\epsilon)T_{-1} \leq (1+\epsilon)(p-1) = p - 1 + p\epsilon \leq p$$

since $p\epsilon < 1$. We have thus shown that the product αR_i does not overflow and hence is exact.

Let us now also bound $M_i = T_{i-1} - \alpha R_i$ from above. We first require a lower bound on R_i. By (3.2), we have

$$R_i \geq \left\lfloor \frac{T_{i-1}}{\alpha}(1-\epsilon) \right\rfloor \geq \frac{T_{i-1}}{\alpha}(1-\epsilon) - 1.$$

We deduce an upper bound on M_i for $i = 0\colon u-2$:

$$M_i = T_{i-1} - \alpha R_i \leq T_{i-1} - T_{i-1}(1-\epsilon) + \alpha = T_{i-1}\epsilon + \alpha \leq p\epsilon + \alpha \leq \alpha + 1,$$

where we have used the fact that for any i, $T_{i-1} \leq T_{-1} \leq p-1$.

It only remains to bound the last word $M_{u-1} = T_{u-2} = R_{u-2}$ from above. Reusing (3.3) we obtain the recurrence relation

$$R_i \leq (1+\epsilon)\frac{T_{i-1}}{\alpha} = \frac{(1+\epsilon)}{\alpha} R_{i-1}$$

which yields

$$R_i \leq \left(\frac{(1+\epsilon)}{\alpha}\right)^i R_0 \leq \left(\frac{(1+\epsilon)}{\alpha}\right)^{i+1} T_{-1} \leq \left(\frac{(1+\epsilon)}{\alpha}\right)^{i+1} p.$$

Using $\alpha = \lceil p^{1/u} \rceil \geq p^{1/u}$, we therefore obtain

$$M_{u-1} = R_{u-2} \leq (1+\epsilon)^{u-1} p^{-(u-1)/u} p = (1+\epsilon)^{u-1} p^{1/u} \leq (1+\epsilon)^{u-1}(\alpha+1) =: c.$$

We have therefore shown that no overflow occurs during the computation as long as $p < 2^t$ and $c < 2^t$. Moreover for $u \leq t$ the condition $c < 2^t$ is included in the conditon $p < 2^t$:

$$c = (1+\epsilon)^{u-1} p^{1/u} \leq 2^{u-1} p^{1/u} \leq 2^{u-1+t/u} = 2^{\left(u(u-1)+t\right)/u} \leq 2^{\left(t(u-1)+t\right)/u} = 2^t.$$

To conclude it suffices to observe that $M_i = T_{i-1} - \alpha R_i$ yields the recurrence relation $T_{i-1} = M_i + \alpha T_i$ for $i = 0\colon u-2$. Hence

$$T_{-1} = \sum_{i=0}^{u-2} \alpha^i M_i + \alpha^{u-1} T_{u-2}$$

which yields the desired decomposition $M = \sum_{i=0}^{u-1} \alpha^i M_i$ since $T_{-1} = M$ and $T_{u-2} = M_{u-1}$. ∎

3.2. Multiword matrix multiplication. We now explain how to use the multiword decomposition to compute the product $C = AB \bmod p$ with a much less restrictive condition on the size of p than with the single word approach.

We consider a general setting where the decompositions of A and B can use possibly different numbers of words, denoted as u and v respectively. We thus compute the decompositions

$$A = \sum_{i=0}^{u-1} \alpha^i A_i, \qquad B = \sum_{j=0}^{v-1} \beta^j B_j,$$

where $\alpha = \lceil p^{1/u} \rceil$ and $\beta = \lceil p^{1/v} \rceil$, and where the coefficients of the words A_i and B_j are respectively bounded by

$$c_A = (\alpha+1)(1+\epsilon)^{u-1}, \qquad c_B = (\beta+1)(1+\epsilon)^{v-1}.$$

The product AB is then given as

$$AB = \sum_{i=0}^{u-1} \sum_{j=0}^{v-1} \alpha^i \beta^j A_i B_j.$$

Therefore one approach to compute $C = AB \bmod p$ would be to compute for each pair (i,j) the product $A_i B_j \bmod p$ using the block matrix product in Algorithm 2.3, storing the result in a temporary workspace T, scaling all coefficients of T by $\gamma_{ij} = \alpha^i \beta^j$ using the modular product reduction in Algorithm 2.2, and finally adding the result $\gamma_{ij} T \bmod p$ in C.

Algorithm 3.2 describes a slightly more involved approach that does not require any temporary workspace. The idea is to add the result of $A_i B_j \bmod p$ directly into C before scaling by γ_{ij}. This is made possible by scaling C by γ_{ij}^{-1} beforehand, since $\gamma_{ij}(\gamma_{ij}^{-1} C + A_i B_j) = C + \gamma_{ij} A_i B_j$. This extra scaling has a negligible cost with respect to the matrix products, and avoids the need for any additional workspace. An important detail is that we do not actually compute γ_{ij}^{-1}, which is not an integer and thus not necessarily representable as a floating-point number, but rather $\delta_{ij} =$

$\gamma_{ij}^{-1} \bmod p$, the modular inverse of γ_{ij} (which is an integer less than p and thus in $\mathbb{F}_p$). As a remark, note that the use of the modular inverse requires p to be prime, since it might not exist otherwise. Therefore, if one wishes to use this multiword product with a composite p, the temporary workspace approach described above should be used.

Algorithm 3.2: Multiword matrix product

Input : $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, u, v.
Output: $C = AB \bmod p \in \mathbb{F}_p^{m \times n}$.

$\alpha = \lceil p^{1/u} \rceil$ and $\beta = \lceil p^{1/v} \rceil$
Decompose $A = \sum_{i=0}^{u-1} \alpha^i A_i$ `// Using Algorithm 3.1`
Decompose $B = \sum_{j=0}^{v-1} \beta^j B_j$ `// Using Algorithm 3.1`
$\lambda = \left\lfloor (2^t - p + 1) / \left((\alpha+1)(\beta+1)(1+\epsilon)^{u+v-2} \right) \right\rfloor$
Initialize $C = 0$
for $i = 0$ **to** $u - 1$ **do**
  for $j = 0$ **to** $v - 1$ **do**
    $\gamma = \alpha^i \beta^j \bmod p$ `// Using Algorithm 2.2`
    $\delta = \gamma^{-1} \bmod p$ `// Modular inverse`
    $C = \delta C \bmod p$ `// Using Algorithm 2.2`
    $C = C + A_i B_j$ `// Using Algorithm 2.3 with block size` λ
    $C = \gamma C \bmod p$ `// Using Algorithm 2.2`
return C

Before discussing the condition on the size of p for this multiword product to be correct, we first describe a variant thereof in Algorithm 3.3. This variant concatenates the matrices B_j in order to compute the products $A_i B_j$, for a fixed i and for all $j = 0 \colon v - 1$, as a single contiguous matrix product $A_i[B_0 \dots B_{v-1}]$. This is potentially more efficient than computing each $A_i B_j$ product independently because the concatenated product has a larger rightmost dimension (nv instead of n) and thus a higher arithmetic intensity. Note that a variant where we concatenate the A_i matrices instead of the B_j ones is also possible; in general one should try to maximize the smallest of the two outer dimensions of the product, hence concatening the B_j matrices when $n < m$ and the A_i ones when $n > m$.

PROPOSITION 3.2. *Algorithm 3.2 (and its concatenated variant Algorithm 3.3) computes exactly $C = AB \bmod p$ under the conditions $p < 2^{t-1}$, $\max(u, v) \leq t$, and*

$$c_A c_B + p - 1 = (\alpha+1)(\beta+1)(1+\epsilon)^{u+v-2} + p - 1 \leq 2^t. \tag{3.4}$$

Proof. We need to check the exactness of all steps. By Proposition 3.1 the multiword decompositions obtained by Algorithm 3.1 are exact if $p < 2^t$ and $\max(u, v) \leq t$. By Proposition 2.2, the computation of $\gamma = \alpha^i \beta^j \bmod p$ using Algorithm 2.2 is exact if $\alpha^i \bmod p \leq p$ and $\beta^j \bmod p \leq p$ are reduced modulo p before applying Algorithm 2.2. To compute δ efficiently, one computes it as $(\alpha^{-1})^i (\beta^{-1})^j$. To ensure it is computed exactly, it is necessary to perform a modular reduction at each step of modular powering. The scalings δC and γC are also exact since δ, γ, and all the coefficients of C are all bounded by p. Finally, the condition for the block product $C = C + A_i B_j$ to be exact using Algorithm 2.3 is given by (2.6) in Proposition 2.3:

$$\lambda \max(A_i) \max(B_j) + p - 1 \leq 2^t,$$

Algorithm 3.3: Multiword matrix product with concatenation

Input : $A \in \mathbb{F}_p^{m\times k}$, $B \in \mathbb{F}_p^{k\times n}$, u, v.
Output: $C = AB \bmod p \in \mathbb{F}_p^{m\times n}$.

Compute $\alpha = \lceil p^{1/u} \rceil$ and $\beta = \lceil p^{1/v} \rceil$
Decompose $A = \sum_{i=0}^{u-1} \alpha^i A_i$ // Using Algorithm 3.1
Decompose $B = \sum_{j=0}^{v-1} \beta^j B_j$ // Using Algorithm 3.1
$\lambda = \lceil (2^t - p + 1)/\big((\alpha+1)(\beta+1)(1+\epsilon)^{u+v-2}\big) \rceil$
Initialize $C = 0$
for $i = 0$ **to** $u-1$ **do**
  $[T_0 \dots T_{v-1}] = A_i[B_0 \dots B_{v-1}]$ // Using Algorithm 2.3 with block size λ
  for $j = 0$ **to** $v-1$ **do**
    $\gamma = \alpha^i\beta^j \bmod p$ // Using Algorithm 2.2
    $T_j = \gamma T_j \bmod p$ // Using Algorithm 2.2
    $C = C + T_j \bmod p$ // Using Algorithm 2.1

which yields (3.4) since by Proposition 3.1 $\max(A_i) \le c_A$ and $\max(B_j) \le c_B$.

Finally, it is easy to check that Algorithm 3.3 is equivalent to Algorithm 3.2 and leads to the same conditions. ∎

Proposition 3.2 provides in (3.4) a sufficient condition on the size of p for the multiword product to be exact. To obtain a more readable and easily interpretable condition, at the price of a harmless breach of correctness, we can replace c_A and c_B by $p^{1/u}+1$ and $p^{1/v}+1$ to obtain

$$p^{1/u+1/v} + p^{1/u} + p^{1/v} + p \le 2^t. \tag{3.5}$$

We will use this more readable and almost correct condition to make a few comments.

- Note first that by setting $u = v = 1$, (3.5) reduces to $p^2 + 3p \le 2^t$: we thus recover the condition $p \lesssim 2^{t/2}$ of the single word algorithm.
- Consider now the case where $u = v = 2$. Then (3.5) becomes $2p + 2p^{1/2} \le 2^t$, that is, $p + p^{1/2} \le 2^{t-1}$. Since $p^{1/2} \le p$, we thus obtain an almost ideal condition $p \le 2^{t-2}$. We conclude that two words for both A and B suffice to handle almost all primes that fit on the target floating-point arithmetic. Moreover, if we increase v (or u) to 3, (3.5) becomes $p^{1/2+1/3}+p^{1/2}+p^{1/3}+p \le 2^t$. When p is large, we have $p^{1/2+1/3} + p^{1/2} + p^{1/3} \le p$, and so the condition reduces to $p \le 2^{t-1}$, a slight improvement compared with $u = v = 2$. Note that this is an ideal condition since $p < 2^{t-1}$ is already required by the modular reduction operations (Algorithms 2.1 and 2.2).
- Interestingly, using $u = 1$ and $v > 1$ (or the converse) still provides a significant improvement to the single word condition: (3.5) yields $p^{1+1/v}+p^{1/v}+p \le 2^t$ or, neglecting the $p^{1/v} + p$ term, $p \lesssim 2^{tv/(v+1)}$. Thus for $v = 2$, the condition is $p \lesssim 2^{2t/3}$, for $v = 3$, it is $p \lesssim 2^{3t/4}$, and so on. As v tends to a larger and larger number of words, the condition tends towards the ideal $p \lesssim 2^t$.

3.3. Discussion on the cost of the algorithms. Now that we have determined the maximum p that a given pair (u, v) can handle, it remains to discuss the cost of the algorithm as function of u and v. Algorithm 3.2 performs uv matrix products of dimensions $m \times k \times n$, hence requiring $2uvmkn$ flops. This is a factor uv more

than the single word product. The multiword product also requires $uvmn(\lceil k/\lambda \rceil + 2)$ reductions, which is also about a factor uv more than the single word one. However, a key difference is that the block size λ is not the same: in the single word case $\lambda \approx 2^t/p^2$ whereas in the multiword case $\lambda \approx 2^t/p^{1/u+1/v}$. Therefore the multiword product can use a potentially much bigger block size λ, which results in a more efficient product since it reduces the relative cost of the reductions and also increases the arithmetic intensity of the matrix products. As for Algorithm 3.3, it performs the same flops as Algorithm 3.2, but is potentially more efficient thanks to an increased arithmetic intensity.

Based on this analysis, we can make some predictions on which approach is the best depending on the size of p. We will then check these predictions in our experiments. Throughout this discussion we assume $u \leq v$, with the understanding that the converse is also possible. We refer to the different variants as (u, v)-product.

The single word $(1, 1)$-product is the least expensive and so is expected to be the best choice as long as it can use a sufficiently large block size, that is, when $p \ll 2^{t/2}$. As p approaches this limit, the $(1, 1)$-product will become increasingly less efficient until it is no longer correct. Around this limit we should therefore switch to a multiword product with the smallest possible cost, that is, $u = 1$ and $v = 2$; this $(1, 2)$-product should be the best until p approaches its new limit $p \ll 2^{2t/3}$. At this point, we have the choice between increasing u or v; since $1 \times 3 < 2 \times 2$, the $(1, 3)$-product performs fewer flops than the $(2, 2)$-product and is therefore preferable as long as $p \ll 2^{3/4}$. At this point, we again have the choice between the $(1, 4)$-product and the $(2, 2)$-product, which perform the same number of flops. Since the limit for the $(1, 4)$-product, $p \ll 2^{4t/5}$, is more restrictive than that of the $(2, 2)$-product, $p \ll 2^t$, the latter may seem preferable than the former. However, when considering the concatenated variant of these algorithms, the $(1, 4)$-product increases the arithmetic intensity by a factor up to 4, instead of 2 for the $(2, 2)$-product. Hence in situations where the concatenated $(2, 2)$-product remains memory bound, the concatenated $(1, 4)$-product could outperform it as long as $p \ll 2^{4t/5}$. Finally, as mentioned before, the $(2, 2)$-product will remain correct for almost all representable values of p, $p \leq 2^{t-2}$; however, as p approaches this limit, the block size λ will tend to 1. Therefore, we might expect the $(2, 3)$-product, the next least expensive variant, to become more efficient for very large p.

TABLE 3.1
Summary of the comparison between the different (u, v)-product variants.

(u, v)	$(1, 1)$	$(1, 2)$	$(1, 3)$	$(1, 4)$	$(2, 2)$	$(2, 3)$
Normalized flops ($= uv$)	1	2	3	4	4	6
Approximate limit on p	$2^{t/2}$	$2^{2t/3}$	$2^{3t/4}$	$2^{4t/5}$	2^{t-2}	2^{t-1}
Limit on bitsize(p) for $t = 53$	26	35	39	42	51	52

We summarize this discussion in Table 3.1, which compares for each (u, v)-product its normalized flops cost (equal to uv) and its limit on p. To give a concrete indication of this limit we also print the maximum bitsize of p (that is, the limit on $\log_2 p$ exclusive), when the target floating-point arithmetic is double precision ($t = 53$).

In summary, the following (u, v)-product algorithms are best used for the following bitsizes of p:

- bitsize $p \in [1\ , 26]$: use the $(1, 1)$-product;
- bitsize $p \in [27, 35]$: use the $(1, 2)$-product;

- bitsize $p \in [36, 39]$: use the $(1, 3)$-product;
- bitsize $p \in [40, 42]$: use the $(2, 2)$-product or the $(1, 4)$-product;
- bitsize $p \in [43, 51]$: use the $(2, 2)$-product;
- bitsize $p \in [52, 52]$: use the $(2, 3)$-product;
- all of the above ranges should in practice be shifted down by a few bits due to the lower efficiency of the product when using a small block size.

We conclude this section by discussing the storage cost of our multiword approach. The (u, v)-product requires $k(um+vn)$ entries for the input words and mn entries for the output. Thus, the more words are used, the more storage is needed: the approach presents a trade-off between the bitsize of p that is supported and the memory usage. Moreover, the use of concatenation introduces an additional temporary workspace requiring vmn entries. Interestingly, in the case of a tall-and-skinny matrix B ($n \ll m, k$), the $(1, v)$-product variants require a negligible storage overhead compared with the storage of matrix A, which makes these variants much less storage intensive than variants with $u \geq 2$, such as the $(2, 2)$-product.

4. Performance benchmarks.

4.1. Experimental setting. We have developed two implementations of the proposed algorithms. The first one is written in FORTRAN and targets CPU architectures; the second one is written in CUDA and targets NVIDIA GPU architectures.

The CPU code was compiled using the `ifort` compiler (v19.1.3) and the Intel MKL (2019.5) library, which we used for all BLAS operations. It was run on two Intel Xeon Gold 6248 CPUs with 20 cores each at 2.50GHz, which have a double precision theoretical peak performance of about 1,600 Gflops/s.

The GPU code was compiled with CUDA v12.6 and the flags: `-arch=sm_80`, `g++ 11.4.0` and `-std=c++17`; all the CUDA instructions are executed on the default stream. We used cuBLAS for all BLAS operations. The code was run on an NVIDIA A100 GPU, which has a theoretical peak performance of about 19000 Gflops/s for double precision arithmetic using tensor cores.

We have written CUDA kernels for the few operations that were not directly available through cuBLAS. This includes in particular kernels to perform the elementise modular reductions and floor operations on a matrix.

As is common when comparing algorithms that perform different number of flops, we choose as performance metric the "effective" Gflops/s rate, defined as

$$\text{Effective Gflops/s} = \frac{2mkn}{t_{\text{avg}}} \times 10^{-9} \tag{4.1}$$

where t_{avg} is the execution time of the algorithm in seconds averaged over 10 runs and where $2mkn$ corresponds to the number of flops performed by one matrix product of dimensions $m \times k \times n$. This metric is best understood as a scaled inverse of the execution time; it can also provide some indication of how well the hardware is utilized, although care should be taken when comparing it to the theoretical Gflops/s peaks given above, since even the (1,1)-product performs more than $2mkn$ flops (due to the modular reductions).

Since the values of the matrix coefficients do not affect the performance of the algorithms, we simply generate them randomly. We consider two scenarios which differ on both the matrix dimensions and what is included in the execution time of the multiword algorithms.

- Large square matrices (Subsection 4.2.1): we first benchmark the algorithms in a general scenario involving large square matrices with $m = k = n = 10016$,

with no particular application in mind. In this scenario, the execution time of the multiword algorithms includes everything: the time for computing the product but also the time for computing the decomposition of both matrices. Since the matrices are large and square, the former requires $O(n^3)$ flops whereas the latter only requires $O(n^2)$ flops, so that the performance of the algorithms are driven by the performance of the product. We do not test the use of concatenation (Algorithm 3.3) in this scenario, since all matrix dimensions are large. We use dimensions that are multiples of 32 because this leads to more consistent and better performance on GPU.

- Unbalanced matrices (Subsection 4.2.2): in this second scenario, we consider a matrix product with unbalanced dimensions, $m = 10923$, $k = 32768$, and $n = 32$; B is thus a tall-and-skinny matrix. These dimensions of matrices are motivated by the polynomial system solving application where one needs to compute the minimal/characteristic polynomial of a square matrix of order k but with only m dense rows [1, 6]. The remaining $k-m$ rows are actually very sparse as they are rows of the identity matrix. This minimal/characteristic polynomial is computed using the block-Wiedemann algorithm [3, 10] whose bottleneck consists in performing $2k/n$ iterated products of the $m \times k$ matrix A with a $k \times n$ matrix B, where $n \ll k$ is a block size parameter under our control; $n = 32$ is a typical choice. Note that matrix A is fixed throughout all iterations. Therefore, in this scenario, we do not include the time for computing the multiword decomposition of matrix A, which can be computed only once and reused for all iterations. We thus only measure the time for computing the decomposition of B and for computing the product. Again, because the product requires $O(mkn)$ flops whereas the decomposition of B only requires $O(kn)$ flops, the cost of the decomposition of B is negligible. In this scenario we will test the use of concatenation on matrix B to increase its right dimension n, which is quite small.

Overall, our benchmark consider three scenarios (square matrices, and unbalanced matrices with or without concatenation), for two architectures (CPU and GPU). This leads to six different figures as summarized in Table 4.1.

TABLE 4.1
Summary of the benchmarks and the corresponding figures.

	CPU	GPU
Square matrices	Figure 4.1	Figure 4.2
Unbalanced matrices (without concatenation)	Figure 4.3	Figure 4.5
Unbalanced matrices (with concatenation)	Figure 4.4	Figure 4.6

4.2. Discussion of the results.

4.2.1. Square matrices. We begin by discussing the results for square matrices on CPU (Figure 4.1). All variants exhibit the same trend with two distinct regimes depending on the bitsize of p: first, a performance plateau which corresponds to the maximum performance achievable when p is small enough so that the cost of the reductions is negligible; then, a performance drop when p begins approaching its limit, due to a decreasing block size λ, which leads to a greater number of modular reductions and more inefficient matrix products.

For example, the (1,1)-product (the reference single word algorithm) achieves a

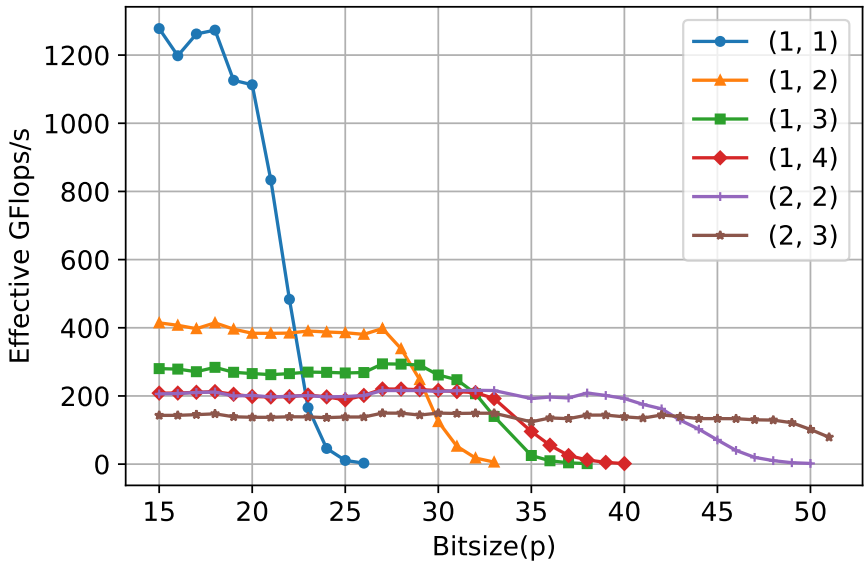

FIG. 4.1. *Performance benchmark for square matrices on CPU.*

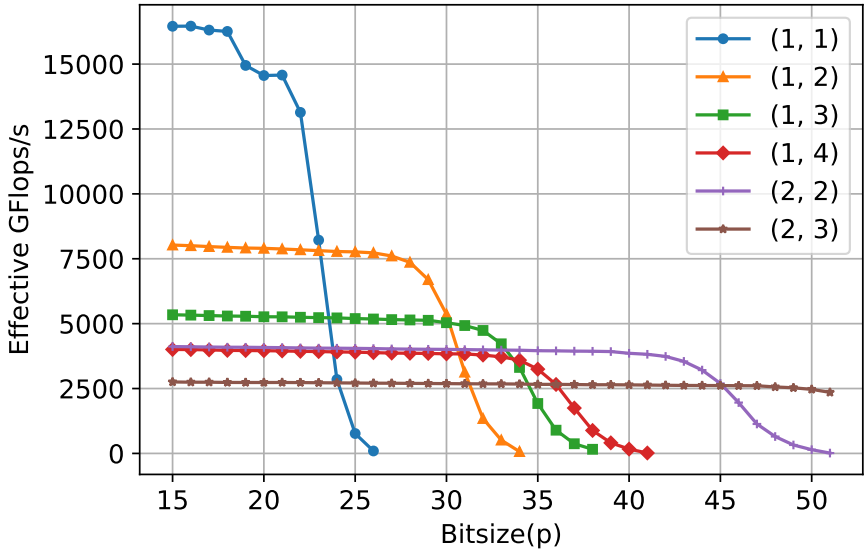

FIG. 4.2. *Performance benchmark for square matrices on GPU.*

performance plateau of 1200 Gflops/s which is reasonably close to the 1600 Gflops/s theoretical peak of the hardware. This confirms that when p is small enough, the (1,1)-product is very efficient and its performance is driven by the matrix product. However, when p becomes larger, the performance drops rapidly. Thus, although the (1,1)-product still produces correct results for primes with 24, 25 and 26 bits, the performance in these cases is too low to be practical.

Our benchmarks therefore confirm the interest of the proposed multiword vari-

ants, which can handle larger primes while maintaining high performance. In particular, the $(1,2)$-product outperforms the $(1,1)$-product for bitsize$(p) \geq 23$. It achieves a performance plateau of 400 Gflops/s, about $3\times$ lower than the performance plateau of the $(1,1)$-product. Note that this $3\times$ time increase (which is larger than the $2\times$ flops increase) can be explained by analyzing the time breakdown of the $(1,2)$-variant. While the $(1,1)$-variant essentially consists of a single block matrix product (Algorithm 2.3), the $(1,2)$-variant also requires computing the multiword decomposition of matrix B and the scalings by δ and γ with Algorithm 2.2. Despite requiring a negligible amount of flops, in practice these extra operations are less efficient than the block product and thus become non-negligible: they represent about 26% and 6% of the total time for the $(1,2)$-variant, respectively.

While the $(1,2)$-product remains correct until bitsize$(p) \leq 35$ the $(1,3)$-product starts outperforming it for bitsize$(p) \geq 29$, with a performance plateau of about 280 Gflops/s. The $(1,4)$ and $(2,2)$-products both require 4 products and thus achieve the same performance plateau of about 200 Gflops/s, which starts outperforming the $(1,3)$-product when bitsize$(p) \geq 33$. In this scenario, the $(1,4)$-product therefore never significantly outperforms the $(2,2)$-product, which maintains its plateau for far larger primes. As expected, the $(2,2)$-product remains correct for all tested primes; however, its performance eventually drops and gets surpassed by that of the $(2,3)$-product, when bitsize$(p) \geq 43$. Even for such large primes, the $(2,3)$-product allows for an almost constant performance of about 150 GFlops/s, which is quite satisfactory given the size of p. Moreover this shows that using more than $3 \times 2 = 6$ subproducts would not be useful.

All of the above comments on the CPU benchmark also apply to the GPU one (Figure 4.2), which exhibits similar trends. The performance of the $(1,1)$-product plateaus at 16000 Gflops/s for small primes, but is rapidly surpassed by that of the multiword variants when p gets larger. One notable observation is that the performance plateau of the (u,v)-product is almost perfectly equal to that of the $(1,1)$-product divided by uv, which suggests that the performance is entirely driven by the matrix product. Thus, the $(1,2)$-product plateaus at 8000 Gflops/s, the $(1,3)$-product at 5300 Gflops/s, etc. The points of crossover (points for which the best algorithm changes), while not exactly equal as in the CPU benchmark, remain similar.

4.2.2. Unbalanced matrices and effect of concatenation. Figures 4.3–4.6 show the performance benchmarks for unbalanced matrices. We can observe the same overall trends as for square matrices, with one notable difference: the matrix products in this case have much lower arithmetic intensity. Thus, the absolute performance values are smaller, that is, farther from the theoretical peak: the $(1,1)$-product plateaus at about 500 Gflops/s on CPU (Figure 4.3) and 4000 Gflops/s on GPU (Figure 4.5). Nevertheless, the relative performance of the multiword variants remains similar than previously and, in particular, we confirm once more the ability of these variants to handle larger primes while retaining satisfactory performance.

Moreover, because of the lower arithmetic intensity of the product, using concatenation in the multiword product becomes interesting. This is illustrated in the performance benchmarks of Figure 4.4 (CPU) and Figure 4.6 (GPU), for which we replace the multiword product (Algorithm 3.2) with its concatenated variant (Algorithm 3.3). The benchmarks show indeed that the performance of the multiword variants can be significantly improved by the use of concatenation (note that the (1,1)-product is unaffected by this change and its performance remains identical). Table 4.2 plots the increase of the performance plateau of the multiword variants achieved by

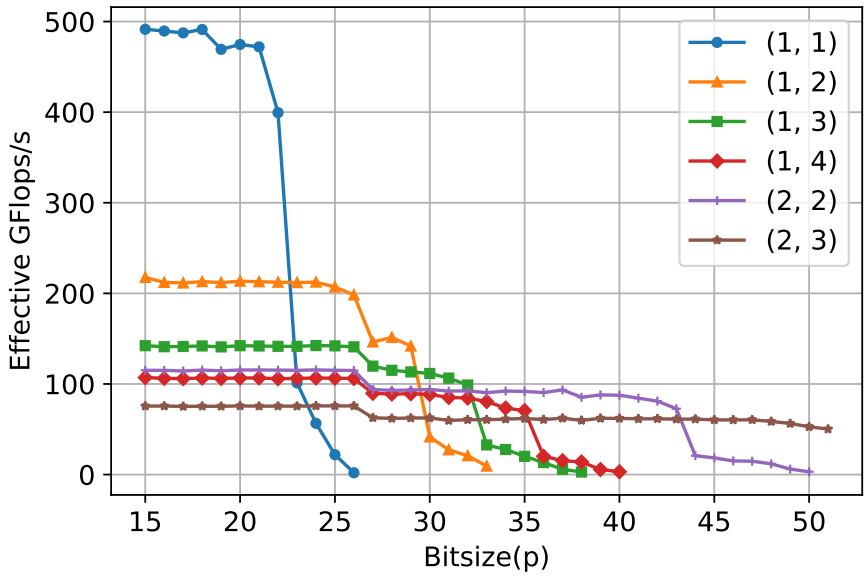

FIG. 4.3. *Performance benchmark for unbalanced matrices on CPU.*

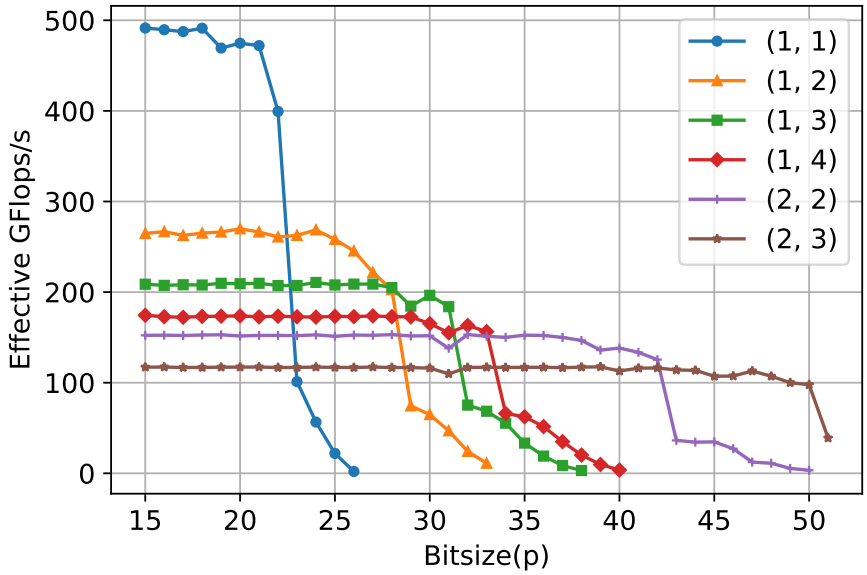

FIG. 4.4. *Performance benchmark for unbalanced matrices on CPU, with concatenation.*

the use of concatenation. On CPU, we observe greater performance increases for greater values of v (for example, 21%, 47%, and 63% increase for the $(1,2)$, $(1,3)$ and $(1,4)$ variants, respectively). This is expected since a larger v corresponds to a larger increase of the arithmetic intensity. To a lesser extent, greater values of u also lead to greater performance increases (for example, 22% vs 32% increase for the $(1,2)$ and $(2,2)$ variants). An interesting consequence of this behavior is that, thanks to con-

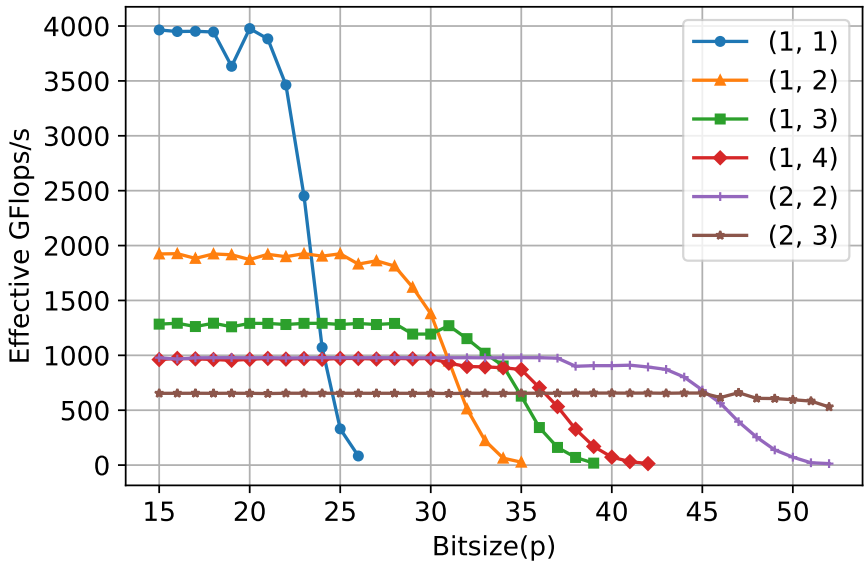

FIG. 4.5. *Performance benchmark for unbalanced matrices on GPU.*

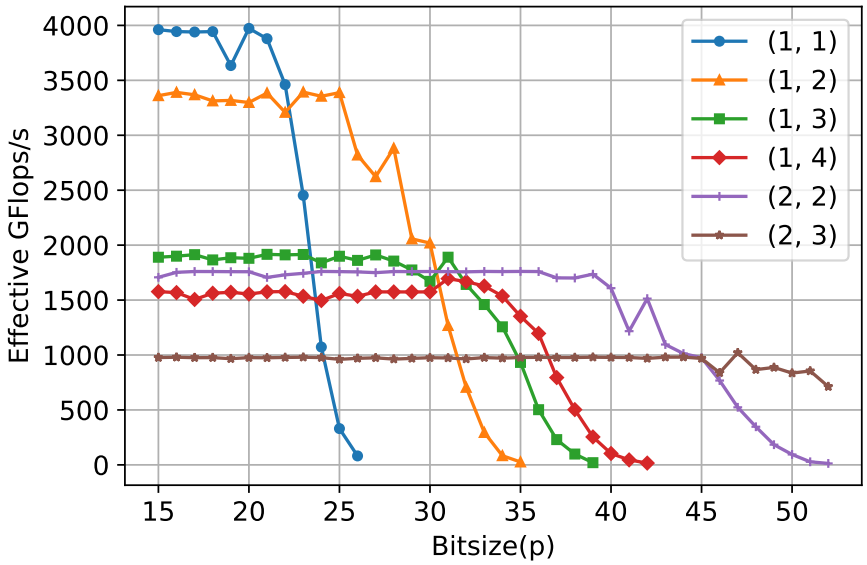

FIG. 4.6. *Performance benchmark for unbalanced matrices on GPU, with concatenation.*

catenation, the $(1,4)$-product achieves a performance plateau of 174 Gflops/s which is higher than that of the $(2,2)$-product (152 Gflops/s). Therefore, for bitsize$(p) = 32$ or 33, the $(1,4)$-product slightly outperforms the $(2,2)$ one (see Figure 4.4).

While concatenation also leads to significant performance increases on GPU, the trend for different (u,v) variants is more unexpected. As shown in Table 4.2, the variants with $v = 2$ benefit from concatenation much more than the other variants, especially those with $v = 3$. After investigating this surprising behavior, we have

determined that this is in fact because the cuBLAS matrix product performance is actually lower for $n = 96$ (corresponding to $v = 3$) than for $n = 64$ (corresponding to $v = 2$). As a result of this behavior, the $(1, 3)$ and $(1, 4)$ variants are never better than the $(2, 2)$ one.

TABLE 4.2
Improvement of the performance plateau (Gflops/s) of multiword variants by the use of concatenation (see Figures 4.3–4.6).

		$(1,2)$	$(1,3)$	$(1,4)$	$(2,2)$	$(2,3)$
CPU	Non-concatenated	218	142	107	115	76
	Concatenated	265	209	174	152	117
	Increase	22%	47%	63%	32%	55%
GPU	Non-concatenated	1932	1314	980	995	663
	Concatenated	3438	1921	1600	1776	1004
	Increase	78%	46%	63%	79%	52%

4.2.3. Summary: variant selection. Table 4.3 summarizes the conclusions of these experiments by indicating, for each of the six benchmarks of Table 4.1, the range of bitsizes for which a given (u, v) variant is the best. We can see that the crossover bitsizes (where the best variant changes), while not exactly equal, are very similar from one benchmark to the other. In particular, the existing $(1, 1)$ approach is systemetically outperformed before its theoretical limit of 26 bits, with crossover bitsizes between 23 and 25. Moreover, the table also shows that each of the multiword variants considered in our benchmarks can be the best for some range of bitsizes, which confirms the importance of adapting (u, v) for optimizing the cost of the product.

TABLE 4.3
Synthesis of the bitsizes for which a given (u, v) variant performs best.

	$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$	$(2,2)$	$(2,3)$
Theory (Subsection 3.3)	[1,26]	[27,35]	[36,39]	—	[40,51]	[52,52]
CPU square	[1,22]	[23,28]	[29,31]	—	[32,42]	[43,52]
CPU unbalanced	[1,22]	[23,29]	[30,32]	—	[33,43]	[44,52]
CPU unbalanced concat	[1,22]	[23,27]	[28,31]	[32,33]	[34,42]	[43,52]
GPU square	[1,23]	[24,30]	[31,33]	—	[34,45]	[46,52]
GPU unbalanced	[1,23]	[24,30]	[31,33]	—	[34,44]	[45,52]
GPU unbalanced concat	[1,22]	[23,30]	[31,31]	—	[32,43]	[44,52]

5. Conclusion. We have presented a new approach to efficiently compute modular matrix multiplication $C = AB \bmod p$ in floating-point arithmetic. The existing single word product is limited to bitsizes of p less than 26 and becomes very inefficient when p approaches this limit. We have proposed in Algorithm 3.2 a new multiword product that decomposes A and B into u and v words, respectively, and computes C with uv modular matrix products. We have also described a concatened variant in Algorithm 3.3 which can be more efficient when the products have low arithmetic intensity. We have proved in Proposition 3.2 the correctness of this approach and determined the maximum size of p that can be handled for a given (u, v) choice. As

summarized in Table 3.1, our multiword approach allows for handling bitsizes as large as 52, and its cost can be optimized by adapting (u, v) depending on the size of p. Our performance benchmarks on CPU and GPU architectures (see Table 4.1) confirm the efficiency of this new approach.

This work opens several perspectives for further performance improvements. First, the block products A_jB_j in Algorithm 2.3 could be computed in parallel via batched matrix products kernels, at the cost of extra memory storage. Second, the multiword approach could be extended to perform the A_iB_j matrix products in lower precision arithmetic. While this would require a greater number of words (and therefore matrix products) to handle a given bitsize of p, it would also allow the use of low precision hardware, in particular GPU tensor cores [2].

Acknowledgements. This work was performed using HPC resources from GENCI-IDRIS (Grant 2024-103516). It was partially supported by the the joint ANR-FWF ECARP (ANR-19-CE48-0015) project, and by the EAGLES (ANR-22-CE91-0007), De Rerum Natura (ANR-19-CE40-0018), InterFLOP (ANR-20-CE46-0009), NuSCAP (ANR-20-CE48-0014), MixHPC (ANR-23-CE46-0005-01), and NumPEx Exa-MA (ANR-22-EXNU-0002) projects of the French National Agency for Research (ANR).

REFERENCES

[1] J. Berthomieu, V. Neiger, and M. Safey El Din, *Faster change of order algorithm for Gröbner bases under shape and stability assumptions*, in Proceedings of the 2022 International Symposium on Symbolic and Algebraic Computation, ISSAC '22, New York, NY, USA, 2022, Association for Computing Machinery, p. 409–418, https://doi.org/10.1145/3476446.3535484.

[2] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, *Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores*, SIAM J. Sci. Comput., 42 (2020), pp. C124–C141, https://doi.org/10.1137/19M1289546.

[3] D. Coppersmith, *Solving homogeneous linear equations over* GF(2) *via block Wiedemann algorithm*, Math. Comp., 62 (1994), pp. 333–350, https://doi.org/10/b724r7.

[4] J.-G. Dumas, P. Giorgi, and C. Pernet, *Dense linear algebra over word-size prime fields: the fflas and ffpack packages*, 35 (2008), pp. 1–42, https://doi.org/10/dj6zp4.

[5] M. Fasi, N. J. Higham, F. Lopez, T. Mary, and M. Mikaitis, *Matrix multiplication in multiword arithmetic: Error analysis and application to GPU tensor cores*, SIAM J. Sci. Comput., 45 (2023), pp. C1–C19, https://doi.org/10.1137/21m1465032.

[6] J.-C. Faugère and C. Mou, *Sparse FGLM algorithms*, Journal of Symbolic Computation, 80 (2017), pp. 538–569, https://doi.org/10.1016/j.jsc.2016.07.025.

[7] T. Granlund and the GMP development team, *GMP: The GNU Multiple Precision Arithmetic Library*, 6.2.1 ed., 2023. http://gmplib.org/.

[8] T. F.-F. group, *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package*, v2.5.0 ed., 2023. http://github.com/linbox-team/fflas-ffpack.

[9] W. Hart, F. Johansson, and S. Pancratz, *FLINT: Fast Library for Number Theory*, 2013. Version 2.4.0, http://flintlib.org.

[10] S. G. Hyun, V. Neiger, H. Rahkooy, and Éric Schost, *Block-Krylov techniques in the context of sparse-FGLM algorithms*, Journal of Symbolic Computation, 98 (2020), pp. 163–191, https://doi.org/10.1016/j.jsc.2019.07.010. Special Issue on Symbolic and Algebraic Computation: ISSAC 2017.

[11] Y. Nievergelt, *Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit*, ACM Trans. Math. Softw., 29 (2003), pp. 27–48, https://api.semanticscholar.org/CorpusID:16228275.

[12] T. Ogita, S. M. Rump, and S. Oishi, *Accurate sum and dot product*, SIAM J. Sci. Comput., 26 (2005), pp. 1955–1988, https://doi.org/10.1137/030601818.

[13] V. Shoup, *NTL: a library for doing number theory*, 2021, http://www.shoup.net.

[14] J. van der Hoeven, G. Lecerf, and G. Quintin, *Modular SIMD arithmetic in Mathemagix*, 43 (2016), https://doi.org/10/f82vvw.