Efficient Gröbner bases computation on GPU

Intern:
Dimitri Lesnoff

Internship duration:
from March 15 to August 31 2022

Supervisors:
Jérémy Berthomieu
Stef Graillat
Théo Mary

February 5, 2025

1. Summary

The general context

In LIP6, the joint laboratory in computer science between CNRS and Sorbonne Université, the team PolSys improves existing algorithms to solve polynomial systems exactly and efficiently. They have begun a recent joint work with another LIP6 team PEQUAN, specialized in high performance computing and quality of numerical algorithms, to implement efficiently these algorithms using parallel machines like GPUs.
Numerous scientific fields like biology, coding theory, combinatorics, and robotics have problems reducing to solving exactly multivariate polynomial systems, over a finite field or rational numbers. To study and solve exactly multivariate polynomial systems, we determine a Gröbner basis of the ideal generated by our system. This is a good representation of the set of solutions. When the number of solutions is finite, we proceed in two steps, we first compute a Gröbner basis for an order for which it easy to obtain a basis, and we then change the order of the basis to a lexicographic order, enabling us to get the solutions of the systems. Two traditional algorithms are used in the general case, either F_4 [Fau99] or F_5 [Fau02] to determine a basis, and Sparse-FGLM for the change of ordering. My internship at LIP6 research laboratory in the PEQUAN and PolSys team, that spanned from 15 March to 31 August concerns precisely the multiplication of matrices in the change of ordering algorithm Sparse-FGLM which has been designed by J.-C. Faugère and C. Mou [FM17]. Among the recent publications about algorithms for fast change ordering using Block-Wiedemann or improving Sparse-FGLM algorithms, we should cite [Hyu+19] and [BND22].
GPUs have already been used in several formal computation problems like bivariate only polynomial systems [MP12], multivariate polynomial determinant, [WC20], and multivariate polynomial constraints [Par+11]. There is also a work on a very similar project by Alan Steel [Ste15] that improves the Block-Wiedemann algorithm used in the actual implementation of Sparse-FGLM by leveraging the parallelism of the GPUs architectures but for a specific problem MinRank. The change of ordering is the current bottleneck in polynomial system resolution. Faster matrix multiplication will enable us to tackle larger practical challenges in cryptography and robotics. An intern in 2020 [Val20], wrote GPU kernels for this operation but with integer types only. We aim to improve the matrix product over finite fields with floating-point numbers instead. Integer types have slower arithmetic (while having more bits available for exact computation). We gain several orders of performance by porting these to GPU and use fast floating-point arithmetic.

My contributions

During this internship, I have compared matrix multiplication timings obtained with efficient linear algebra libraries on CPU using floating-point arithmetic (Eigen [GJ+10], OpenBLAS [ZQG11]). I have then benchmarked existing libraries (FFLAS [gro19], NTL [Sho21], FLINT [HJP13]) on CPU to multiply matrices over finite fields and determined the loss of performance (reals with floating-point types).
To port the algorithms to GPU, I have adapted techniques to reduce the number of modular reductions while avoiding overflow for dot product computation to matrix mulplication. I have transcribed rules to keep the arithmetic exact in function of the prime characteristic of the field. Finally, I wrote a GPU kernel for modular matrix multiplication that aim to be integrated into the open-source C library MSOLVE developed partly by researchers of LIP6.

Arguments supporting its validity

The algorithm that I developed achieves a performance of about 150 GFlops on a RTX Quadro 8000 graphics card which has 509 GFlops as maximal theoretic performance. This means we get a performance close to the one we would get in a perfect setting, i.e. without modular reductions and with only one large matrix multiplication.
When solving over rationals, the size of the characteristic of the prime fields scales with the size of polynomial systems. The bound for efficient computation is tight, and once the prime size gets over

this bound we need to change drastically our approach.

Summary and future work

Determining most efficient modular matrix product for large prime numbers over GPUs can be easily transposed to other formal computation problems.
The kernel I have proposed for matrix multiplication over finite field, has yet to be integrated into formal tools and tested inside Sparse-FGLM implementation. If the finite field elements are smaller, we could leverage tensor cores to speed up even more the computations. We will have to study the balance on the CPU and GPU loads inside Gröbner basis algorithms. There is one other implementation to improve in order to scale the resolution to larger problems in the F_4 algorithm that I will consider during my PhD thesis.

Contents

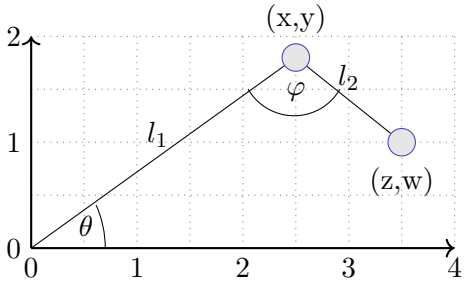


Figure 1: A schematic robotic arm

2. Introduction

Numerous problems arising from various fields such as biology, coding theory, combinatorics, robotics or aerospace engineering comes down to the resolution of polynomial systems $f_1 = \cdots = f_m = 0$ in variables $x_1, \ldots, x_n$ exactly over a finite field or rational numbers.

$$\begin{cases} f_1(x_1, \ldots, x_n) & = 0, \\ \quad \vdots & = \vdots \\ f_m(x_1, \ldots, x_n) & = 0. \end{cases}$$

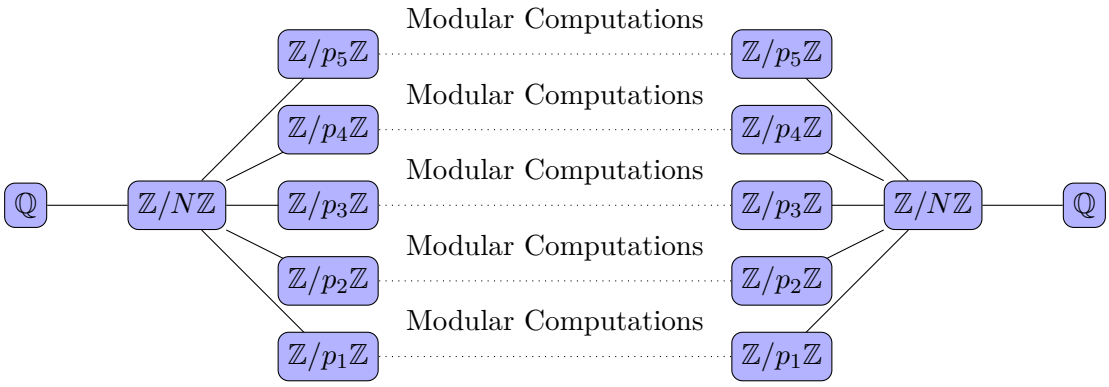
For example, when considering a robotic arm, one may want to find the singular positions, specific positions, in which movements of the arms are physically restricted. If the arm tries to stretch too far, it will damage itself. I have depicted in figure 1, a schematic robotic arm fixed at (0, 0). Its head, with position (z, w) will reach the ground for specific angles. The trigonometric functions of these angles satisfy the following system:

$$\begin{cases} (c_1 s_2 + s_1 c_2) l_2 + s_1 l_1 = 0, \\ c_1^2 + s_1^2 - 1 = 0, \\ c_2^2 + s_2^2 - 1 = 0. \end{cases}$$

where

$$\begin{aligned} c_1 &:= \cos(\theta), \qquad & s_1 &:= \sin(\theta), \\ c_2 &:= \cos(\varphi), \qquad & s_2 &:= \sin(\varphi). \end{aligned}$$

When solving polynomial systems exactly over the rationals, the intermediate results size explodes making the computations practically infeasible. To circumvent this problem, we split the system into multiple ones over finite fields, solve each one of them and reconstruct representations of the initial solution using the Chinese Remainder Theorem (CRT).[1]



[1] We actually represent solutions by polynomials, since the solutions are in extensions of the field of rationals which is not algebraically closed

If the initial problem requires solving over a finite base field, this decomposition with CRT is not possible. With rationals, we choose randomly the primes $p_1, \ldots, p_r$ and we may fall on some *wrong primes*. During computations, a prime may divide the numerator or denominator of a rational coefficient, making the partial result erroneous. In this case, we have to restart the computations by switching the wrong prime to another randomly selected prime. The larger the primes are, the lower the probability for a wrong prime to occur is . All in all, we want to solve polynomial systems over a finite prime field, whose characteristic is at least of the order of 2^{18}.
Solving polynomial systems (PoSSo) is NP-hard even if the base field is finite [FY79].
The end-user may ask several questions on the solution set: is it finite over the algebraic closure of the base field? In the rational case, are the complex or real solutions clustered? What is the dimension of the solution set if it is not finite? To solve these questions we introduce general formal tools.
If $m = n$ the solutions for $f_1, \ldots, f_m$ of respective degrees $d_1, \ldots, d_m$ may *generically* be described as the specific triangular system:

$$\begin{cases} g_n(x_n) & = 0, \\ x_{n-1} & = g_{n-1}(x_n), \\ \vdots & = \vdots \\ x_1 & = g_1(x_n). \end{cases}$$

The system is said to be *in shape position*. It is unique if $\deg(g_1), \ldots, \deg(g_{n-1}) < \deg(g_n)$. We can use this representation for almost all polynomial systems.
The set of polynomials

$$\{x_1 - g_1(x_n), \ldots, x_n - g_n(x_n), g_n(x_n)\}$$

is a *(lexicographical) Gröbner Basis* of the ideal generated by the polynomials of our system:

$$I = \langle f_1, \ldots, f_m \rangle = \{f \in \mathbb{K}[x_1, \ldots, x_n] \mid \exists \lambda_1, \ldots, \lambda_n \in \mathbb{K}[x_1, \ldots, x_n],\ f = \lambda_1 f_1 + \cdots + \lambda_n f_n\}.$$

We set: $D = \deg g_n$ the degree of the n-th polynomial, it is the degree of the system. To define formally a Gröbner basis, we first need to define an *admissible monomial order* and the *leading term* of a multivariate polynomial. A polynomial $f = 5x^3y + 7xy^2$ is constituted of terms e.g. $5x^3y$, with its coefficient, 5, and its monomial part, x^3y.

Definition 1 (Admissible ordering)**.** An order on the monomials $\leq$ (or equivalently on the ordered list of its exponents living in $\mathbb{N}^n$), is *admissible* if

1. $\leq$ is a total ordering on $\mathbb{N}^n$,
2. $\leq$ is compatible for the addition, i.e. if $a \leq b$ implies $\forall c \in \mathbb{N}^n, a + c \leq b + c$,
3. $a \geq 0$ for all $a \in \mathbb{N}^n$ (or every nonempty subset has a smallest element).

Definition 2 (Leading Term)**.** A polynomial f is comprised of monomials and coefficients. A monomial and a coefficient together are called a term. The largest one for an admissible monomial order $\leq$ is the leading term of f noted as: $LT_{\leq}(f)$.

Definition 3 (Gröbner Basis)**.** Let G be a finite subset of an ideal I. If for every $f \in I$, there exists a $g \in G$ such that $LT_{\leq}(g)$ divides $LT_{\leq}(f)$, then G is a Gröbner basis.

The computation of Gröbner bases is in general exponential in n, though some Gröbner bases are easier to compute than others. We usually proceed in two steps to obtain a lexicographic Gröbner basis.

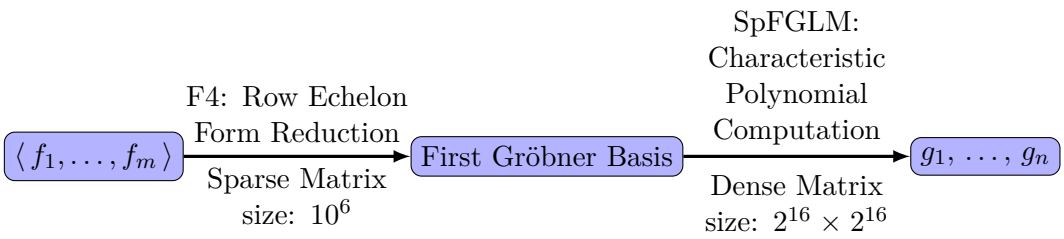


Figure 2: General Framework of Polynomial System Solving with Gröbner bases

First, we compute a Gröbner basis of the ideal, for a *total degree* ordering (usually *degree reverse lexicographic* ordering, also known as DRL) using Buchberger's or Faugère's F_4 [Fau99] and F_5 [Fau02] algorithms. Secondly, we change the order of the Gröbner basis using the FGLM algorithm or its faster variant, the Sparse-FGLM [FM11] algorithm.
In figure 2, we have given the sizes of the matrices for the challenges we want to tackle.
In the generic case, this second step yields the elements of the Gröbner Basis $g_1, \ldots, g_n$ in $O(D^3)$ operations, where D can be up to several thousands in the practical challenges we aim to solve.
My principal goal in this internship is to design efficient ways to improve current implementation of Sparse-FGLM. Indeed, the size of the matrices intervening in the linear algebra operations during the resolution are too big to do the computations in a feasible amount of time with current implementations and this even with parallelisation over CPUs.
Graphics Processing Units originally designed for floating-point numbers computation, arising in graphics computation, are also well-suited to process large blocks of data in parallel and to perform linear algebra routines.
I will start by a focus on the algorithm itself with two implementation variants and the actual challenges we want to tackle. Then I will give an overview of the GPU's architecture. Finally, I will present how I adapted fast linear algebra routines for the multiplication of matrices over finite fields and how to minimize modular reductions while avoiding overflows.

3. Formal Computation Problematics

3.1. Sparse-FGLM algorithm

The FGLM algorithm, designed in 1993 [Fau+93], is the first algorithm to change the ordering of a Gröbner basis of a zero dimensional ideal. In 2011 [FM11] and in 2017 [FM17], Faugère and Mou described the Sparse-FGLM which improves special cases like the systems in the shape position. The algorithm works for any ordering chosen as input or output. To solve polynomial system, we usually change from a degree reverse lexicographical ordered (DRL in short) Gröbner basis to a lexicographical ordered (LEX) Gröbner basis.
The original theoretical algorithm to solve polynomial systems proposed by B. Buchberger in 1976, we compute S-polynomials (a specific linear combination of two polynomials) for certain pairs of a set initialized with the polynomials of our system.
For a zero-dimensional ideal I of $\mathbb{K}[x]$, the algebra $A = \mathbb{K}[x]/I$ is finite dimensional. We can look at the linear application from A to itself that multiplies a polynomial by x_n. Since G the DRL Gröbner basisis a basis of I, we can consider the linear application:

$$f \mapsto \text{NormalForm}(f \cdot x_n, G, \text{DRL}),$$

where $\text{NormalForm}(f, G, \text{DRL})$ is the remainder division of the polynomial f by the set of polynomials G for the order DRL.

Definition 4 (Multiplication Matrix)**.** The matrix multiplication is given by the matrix of these normal forms for all monomials for the Gröbner basis.

Example 1. The companion matrix of an univariate polynomial P in $\mathbb{K}[x]/\langle P\rangle$ is a multiplication matrix by x.

The algorithm start by considering shape position ideals:

Definition 5. An ideal $I \subseteq K[x_1, \ldots, x_n]$ is in *shape position* if its Gröbner basis for LEX order is of the following form:

$$[f_1(x_1),\, x_2 - f_2(x_1),\, \ldots,\, x_n - f_n(x_1)]$$

Let $I \subseteq \mathbb{K}[x_1, \ldots, x_n]$ of degree D (the degree of the ideal is the dimension of $\mathbb{K}[x_1, \ldots, x_n]/I$ as extension of $\mathbb{K}$). If the ideal is in shape position, they apply successively two procedures, the first one being probabilistic while the second one is deterministic. They stop as soon as one of the algorithm succeeds. If none of the algorithm succeed, it must be that the ideal is not in shape position. The first algorithm (see 1) needs $\mathcal{O}(D(N_1 + n\log(D)^2))$ field operations, where N_1 is the number of non-zero elements in the first multiplication matrix.
In the general case (non shape position ideal), they use the BMS (Berlekamp-Massey-Sakata) algorithm.

3.1.1. General Principle

The algorithm for Sparse-FGLM is linked to the notion of Krylov subspace.

Definition 6 (Krylov subspace)**.** The l-order *Krylov subspace* generated by the multiplication matrix T and a vector r is the linear subspace spanned by the set of the powers of T multiplied by r, that is:

$$\text{Span}(r, Tr, T^2 r, \ldots, T^{l-1} r)$$

For r generic, the vectors $\{r, T \cdot r, T^2 \cdot r, \ldots, T^{l-1} \cdot r\}$ are linearly independent for all $l < \deg(\mu(T))$ where $\mu(T)$ is the minimal polynomial of T. In the two algorithms, we will exploit the relation between the powers of T and the minimal polynomial of T by constructing the linearly recurring sequence $\left\{r \cdot T^i \cdot e, \quad i = 0, \ldots, 2D-1\right\}$
where r is a randomly chosen vector, since otherwise if r is in a generalised proper space, the algorithm fails, e is the first element of the canonical basis: $\boldsymbol{e} = (1, 0, \ldots)^t$. Hence, we keep only the first coordinate.

Algorithm 1: Shape Position (Probabilistic) Algorithm

Input: G_1, Gröbner basis of a 0-dimensional ideal $I \subset \mathbb{K}[x_1, \ldots, x_n]$ w.r.t. $<_1$.
Output: G_2, Gröbner basis of I w.r.t LEX if the polynomial returned by `BerlekampMassey` () is of degree D; Fail, otherwise.

Compute the canonical basis of $\mathbb{K}[x_1, \ldots, x_n]/\langle G_1 \rangle$ and multiplication matrices $T_1, \ldots, T_n$
$\boldsymbol{e} := (1, 0, \ldots, 0)^t \in \mathbb{K}^{(D\times 1)}$
Choose $\boldsymbol{r} := \boldsymbol{r} \in \mathbb{K}^{(D\times 1)}$ randomly
for $i = 1, \ldots, 2D-1$ **do**
    $\boldsymbol{r}_i = (T_1^t)\boldsymbol{r}_{i-1}$
Generate the sequence $s := [\langle \boldsymbol{r}_i, \boldsymbol{e} \rangle : i = 0, \ldots, 2D-1]$
$f_1 := \texttt{BerlekampMassey}(s)$
if $\deg(f_1) = D$ **then**
    $H := H_D(s)$
    for $i = 2, \ldots, n$ **do**
        $\boldsymbol{b} := (\langle \boldsymbol{r}_j T_i \boldsymbol{e} \rangle : j = 0, \ldots, D-1)^t$
        Compute $\boldsymbol{c} = (c_1, \ldots, c_D)^t := H^{-1}\boldsymbol{b}$
        $f_i := \sum_{k=0}^{D-1} \boldsymbol{c}_{k+1} \boldsymbol{x}_1^k$
    return $[f_1, x_2 - f_2, \ldots, x_n - f_n]$
else
    return Fail

In 1, the BerlekampMassey finds the minimal polynomial of a linear recurrence sequence. We already have very efficient implementation of this algorithm.
We want to improve the existing implementation of the multiple products matrix-vector appearing in the first for loop (see algorithm 4 below).
The difficulty resides in the large size of the matrices. These are mathematically speaking sparse ($\geq 20\%$), but as we compute their powers, the density of the matrix increases with the exponent. We have made the choice to consider only the dense columns of our matrices and treat the matrices as dense matrices only.
This is justified by the maximum amount of five percent of non-zero coefficients among all coefficients of the matrix allowed in sparse matrices libraries like CuBLAS, which is lower than the percentage of non-zero coefficients observed in practice.
There is at least 20% non-zero coefficients (see [FM17] section 7 for exact percentages).

Algorithm 2: Specific portion of the Sparse-FGLM we want to implement efficiently

$\boldsymbol{s} := [\boldsymbol{r}_0]$
for $i = 1, \ldots, 2D-1$ **do**
    $\boldsymbol{r}_i = (T_1^t)\boldsymbol{r}_{i-1}$
    $\boldsymbol{s} = \boldsymbol{s} + [\langle \boldsymbol{r}_i, \boldsymbol{e} \rangle]$

We want to focus first into efficient matrix-matrix products.

3.1.2. Keller-Gehrig method

The Keller-Gehrig [Kel85] [DPW05] method has been developed for efficient computation of the Krylov sequences appearing in the computation of the characteristic polynomial of a matrix. He reduced in his article this computation essentially to the time of a matrix multiplication. Let A be an $n \times n$ matrix with indeterminate coefficients and $e \neq 0$ a vector from $\mathbb{K}^n$. Then $U = (\boldsymbol{e}, A\boldsymbol{e}, A^2\boldsymbol{e}, A^3\boldsymbol{e}, \ldots, A^{n-1}\boldsymbol{e})$ is a regular $n \times n$ matrix. We precompute powers of two among the powers of A by repetitive squaring, that is:

$$A, A^2, A^{2^2}, A^{2^3}, \ldots, A^{2^k}.$$

With these matrices, we can compute at each step twice more elements than elements of the vector that we already know, starting with e.

Algorithm 3: Keller-Gehrig fast computation of characteristic polynomial

$U := [\boldsymbol{e}, A \cdot \boldsymbol{e}]$
Powers$\leftarrow \emptyset$
$R \leftarrow A$
for $i = 1, \ldots, 2^k = n/2$ **do**
    $R \leftarrow R \times R$
    Powers[i] $= R$
for $i = 1, \ldots, \log(n)$ **do**
    $U[2^i, \ ..., \ 2 * 2^i - 1] = Powers[i] \cdot U$
return U

We discarded temporarily Keller-Gehrig method since it requires a subcubic matrix multiplication to compute efficiently the Krylov sequence, and it is not obvious if Strassen matrix multiplication is efficient on GPUs. The complexity of the algorithm is in $\mathcal{O}(n^\omega \log(n))$, where w is the matrix product exponent. The overhead of $\log(n)$ is usually compensated by the subcubic matrix multiplication. Keller-Gehrig presented another algorithm in his article, that is in $\mathcal{O}(n^\omega)$. The constant behind the Big-O notation in this algorithm is much more important. Over GPU, Strassen matrix multiplication and the Winograd variant have been implemented and analyzed in [LRS11]. They have long not been considered due to their poor numerical stability. These fast matrix multiplication algorithms are not used up to my knowledge in CuBLAS, and could be used in our context, at least with a limited amount of recursion levels.
In contrast to the Wiedemann algorithm, we can not consider only dense columns of our matrix, so Keller-Gehrig will need to compute exponents of larger matrices and keep as much as possible the sparsity of lower exponents of the multiplication matrix.
In this section, I have described the key algorithms for the change of ordering of Gröbner basis. The speed of current implementation of these algorithms is limited by the multiplication of modular matrices. I will present my implementation in the next section.

4. HPC and GPU Approaches

4.1. Modular Field Arithmetic

We would like to reduce the cost of modular reduction in large prime fields $\mathbb{Z}/p\mathbb{Z}$. Determining the best size for prime characteristic is complex and vary greatly in function of the coefficients of the polynomial systems. The characteristic of those large fields can be represented with 20 to 30 bits width integers ($2^{19} \leq p \leq 2^{29}$).

4.1.1. Representation of Finite Fields

Improving modular reduction has been extensively studied, especially for cryptographic applications [Mon85] [Bar87].
For small prime fields, we generally store in a table all the multiplication results, reducing the cost of the multiplication to the cost of a table lookup. When the prime is big, the table's size is too important for the table to be stored into cache, making the lookup more costly than the operation itself. Thus we can not rely on Zech's logarithm or other tabulation methods.
For larger prime fields, when using integer arithmetic, we generally use the Barrett's reduction [Bar87] or the Montgomery's modular reduction [Mon85].

4.1.2. Floating-Point Representation

To leverage fast arithmetic over GPU's, we would like to use floating-point types.
Even though we have less bits at our disposal, the theoretical speeds of arithmetic operations are faster with these types, especially on GPU. We also dispose of specific instructions for floating-point that enables us to do several operations in the same instruction (FMA computes both a multiplication and an addition).
On CPUs, the Advanced Vector Extensions (AVX) to the x86-64 instruction set enable to do multiple instructions in parallel with instructions operating on multiple registers at once (SIMD parallel processing). We could use them when doing operations heterogeneously between GPU and CPU. Finally, specific linear algebra libraries (BLAS), that optimizes cache memory accesses (at least on CPU), are designed only for these types. There is also no support for 64-bit integer type on GPU, limiting drastically the prime numbers we can represent.
To represent finite field elements with floating-point native types (`float` and `double`), we use the bits of the mantissa. The number of bits of the mantissa for half, single and double precision has been defined by the norm IEEE 754.

4.1.3. Reduction in Modular Fields

To compute the modular reduction with floating-point in prime fields, we can compute the quotient by dividing just like with reals and then check for rounding errors. To keep arithmetic exact, I have considered the algorithms described in [HLQ14], section 3. They described how to improve the modular multiplication, but we can use the same technique only for modular reduction (by multiplying by 1).
With numeric types (`float` or `double` in C/C++ programming languages), we can also represent integers and thus prime field elements using the 23/52 (single or double precision respectively) bits of trailing significant field. The mantissa of floating points is made of the trailing significant field and the sign bit (24/53 bits in total). In the following paragraphs, we only use the trailing significant field and the exponent field to represent the integers. The advantage gained by using a shifted representation is twofold: the sign can be stored in a specific bit, and the size of the result of the operations reduces a lot.
Classical representation of finite fields: integers modulo a prime number, the characteristic of the field. There is two choice, either we use the mathematical usual range between 0 and $p-1$, either we use a shifted or centered representation between $-(p-1)/2$ and $(p-1)/2$. It could be possible to use also the sign bit with a centered representation of the modular field (integers in the range

$-(p-1)/2 \leq n \leq (p-1)/2$). Introducing negative numbers makes implementation more tedious, since an extra care must be done in the operations used and the conversions between float and integers made for the arithmetic to be exact.
To reduce x modulo p with a floating-point type, we first precompute the floating point approximation of $1.0/p$. If c is an integer approximation of x/p, then $d = x - c * p$ is an approximation of x mod p at distance $\mathcal{O}(p)$. The authors of the aforementioned article propose the following algorithm to compute the modular reduction:

Algorithm 4: Modular reduction

Input : Integer x. The prime number p, characteristic of the field and the floating point approximation of its inverse $u = \text{fl}(1.0/p)$.
Output: a mod p

```
1 ModularReduction(x, p, u)
2     b = x * u
3     c = ⌊ b ⌋
4     d = a - c * p
5     if d ≥ p then
6         return d - p
7     if d ≤ p then
8         return d + p
9     return d
```

Let us call f the number of bits in the binary representation of p. The above algorithm works for all rounding modes, if and only if $f \leq t/2$ where t is the maximum number of bits to represent the integer. If the rounding mode is set towards infinity, we can discard the first if. If furthermore $f \leq (t-1)/2$, we can drop also the second if.
Removing conditionals as much as possible is important when programming for GPU. The SIMT model of parallelism forces threads to share the same instruction pointer inside warps (group of 32 threads for NVIDIA GPUs, 16 threads for AMD GPUs). This means that the same instruction is executed at the same time by 32 threads. When a branch occurs, the 32 threads have to do the computation independently and sequentially. It is not clear if the threads may be grouped by similar instruction's address after the branch.
Yet during our tests we could not observe a sensible difference when removing those ifs. This absence of difference may be due to a compiler optimisation of these two small conditionals with some branchless programming technique, like the boolean formula given in the next algorithm description.
This limitation on the number of bits of p is actually too restrictive for certain of our applications. We can not use float (which are more efficient than double) since $\lfloor 23/2 \rfloor = 11$ is strictly lower than the 20 bits required.
In the same article, the authors leverage the *Fused-Multiply-and-Add* FMA instruction that was at this time too recent to be present in all CPU architectures, but is widely supported as today. It is supported both on our CPU and GPU. This instruction computes a multiplication and an addition at the same time. This operation is also called AXPY in linear algebra libraries [2] More precisely, `fma(x, y, z)` computes the floating-point number $xy + z$. The floating-point rounding-error does not affect multiplication. We can thus gain an important increase in precision with the second algorithm:
With the FMA instruction, the multiplication stays exact for almost twice the number's bits ($t \leq f - 2$).

4.2. NVIDIA GPU Architecture

To offer the best performance, we searched how to leverage the parallelism of better computers architectures. We could program for FPGAs (Field Programmable Gate Arrays) or ASICs

[2] The name comes from the usual symbols to denote the operation: $z \leftarrow \alpha x + y$

Algorithm 5: Modular reduction with FMA algorithm

Input : Integer `x`. The prime number p, characteristic of the field and the floating point approximation of its inverse $u = \text{fl}(1.0/p)$.

Output: $a \bmod p$

```
ModularReduction(x, p, u)
    b = x * u
    c = ⌊ b ⌋
    d = fma(−c, p, x)
    return d * (d ≥ 0) * (d < p) + (d − p) * (d ≥ p) + (d + p) * (d < 0)
```

	NVIDIA Quadro RTX 8000	NVIDIA RTX 3070
FP16 (TFlops)	32.62	20.31
FP32 (TFlops)	16.31	20.31
FP64 (GFlops)	509.8	317.4
SM count	72	46
Base Clock (Mhz)	1395	1500
Memory Size (GB)	48	8

Table 1: Theoretical maximal throughput for different floating-point types on GPU

(Application Specific Integrated Circuits) or even design our algorithms on multi-core computers. We have chosen GPUs for their theoretical (and practical) high performance floating-point arithmetic. The other kind of parallel computers are designed with power efficiency in mind. We are not limited by power consumption. I had access during my internship to a NVIDIA Quadro RTX 8000 (Turing architecture) card whose characteristics is displayed in table 1 with a consumer's graphics card from the latest generation as a comparison.
The theoretical maximal performance increases as the number of bits of the type decreases. We notice a big difference of performance between a 32 bit and a 64 bit floating-point type on GPU. Cards like the Quadro designed for computational servers have a lower base clock, but a higher SM (Streaming Multiprocessor) count and larger memory. The memory size is much more limited on GPUs than on CPUs. On consumer's card, it is not possible to store large matrices (> 15000 coefficients) and their product.
Even though the algorithms described in this report does not depend on the GPU constructor, I have written my implementation and tests in CUDA, the C++ wrapper interface for NVIDIA cards. I will describe NVIDIA GPUs specific architecture (see figure 3 extracted from the Cuda Programming Guide []), but the core ideas are very similar to AMD's GPU architecture even if the vocabulary employed is different.
Very similar to the SIMD (single instruction multiple data) parallelism model, GPUs rely on a SIMT (single instruction multiple threads architectures). Threads are regrouped by warps (32 threads at most for NVIDIA, 64 for AMD) and share the same instruction pointer. It means that at every clock

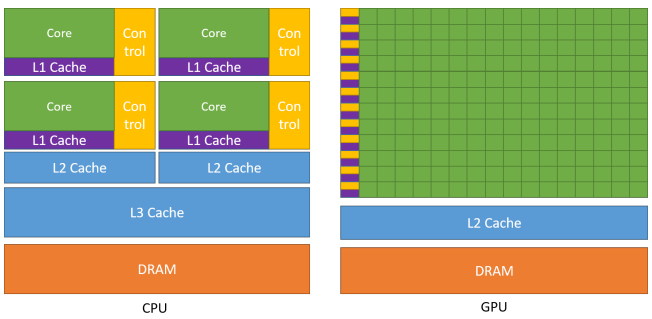


Figure 3: GPUs architectural differences with CPUs

cycle, they execute the exact same instruction. Contrarily to CPUs, threads are much more lightweight.
There can be at most (1024 threads/32 threads) per Streaming Multiprocessor at the same time. This limit one of the parameter to call when lauching a function on GPU, which are called *kernel*. To simplify architecture considerations and have a consistent API across all their GPUs, NVIDIA designed their API CUDA. We wrote programs in C++ with a few additions and a special compiler `nvcc`. The CPU is designed as *host* and one GPU is called a *device*. The functions are designed by three keywords `__global__`, `__device__` and `__host__` which are respectively for device functions called from host, GPU-specific functions and CPU-specific functions (the `main` function is a `__host__` function).
When launching a `__global__` function (also called *kernel*), we can specify two parameters between three angle brackets. The second parameter is the number of threads per block. It is a parameter between 32 and 1024. It is recommended to be a multiple of 32 to maximize *occupancy*. By testing on simple example for array addition or Hadamard product (element-wise multiplication), I noticed that my programs were faster with 512 or 256 threads, and slower with 1024 threads or less. The first parameter is the number of blocks. This parameter can go from 1 to 65536, and can be specified either as a regular unsigned integer, or as a vector with two or three dimensions. Using more dimensions, enable us to specify more blocks than the limit of 65536 for the one-dimensional case. In practice, this is not relevant as blocks are dispatched only when SMs does not have active threads running. The limit for simultaneous executing threads and blocks depend on the GPU and is equal to the number of SMs times the maximal number of threads per SM. For the Quadro RTX 8000, it is equal to 72*1024=73728.
The number of blocks should match the size of our problem's data divided by the number of threads per block.

4.3. Exact Computation of Matrix Product

The product of matrices can be seen, in the classical algorithm in $\mathcal{O}(n^3)$ operations, as the dot products of the row vectors of the left matrix with the column vectors of the right matrix.

4.3.1. Exact Finite Field Dot Product

We want to compute efficiently and exactly a dot product $a \cdot b = \sum_{i=0}^{N} a_i * b_i$ over large prime finite fields.
Usually to multiply two floating point numbers, we store only integers with characteristic p whose size is at most 26 bits wide. Thus, we can multiply them and represent them in a float, since the product has at most 52 bits. We can even perform additionnaly an addition, since the double precision float mantissa has 53 bits before a modular reduction becomes mandatory.

Proposition 1. *The product of k integers with at most n bits requires $n * k$ bits to be represented, whereas k sums requires only $n + \log_2(k+1)$ bits.*

We want to reduce the number of modular reductions computed during the dot-product algorithm. We present first the classical way to compute a dot-product in finite field.

Algorithm 6: Dot product over finite field

Input : Two vectors `a` and `b` of size N. The prime number p.

```
1 DotProduct(a, b, N, p)
2     res = 0
3     for i in 0, ..., N do
4         res ← res + (a[i] · b[i]) mod p // RED1
5         res ← res mod p // RED2
6     return res mod p // RED3
```

We can choose whether we do the first, second or third reduction. If we do fewer modular reductions, res will need in return more bits for the result to be exact. The `RED3` modular reduction step is only necessary if we do not do the last `RED2` modular reduction. As such, we obtain four algorithms:

Only RED3: one reduction

RED2: N modular reductions

RED1 (& RED3): $N + 1$ modular reductions

RED1 & RED2: $2N$ reductions

We want to compute modular reductions only when necessary. This principle, the delayed modular reduction (DMR) can be implemented in several ways. First, we have the λ-algorithm which computes the dot-product by reducing the temporary `res` (`RED2`) only after exactly λ additions. Let $k = q * \lambda + r$ the euclidean division of k by λ. We can then write the dot product as:

$$\begin{aligned} a \cdot b &= \sum_{i=1}^{N} a_i * b_i \\ &= \sum_{j=1}^{q} \sum_{i=j*\lambda+1}^{\lambda} a_i * b_i + \sum_{i=q*\lambda+1}^{r} a_i * b_i \end{aligned}$$

We can thus write the dot product sum with modular reductions as:

$$(a \cdot b) \bmod p = (\sum_{j=1}^{q} S_j \bmod p + S_{q+1} \bmod p) \bmod p \tag{1}$$

For this computation to be exact, we must ensure that each sum S_j is representable on a float mantissa. In other words, *lambda* must be such that $S_j(\lambda) \leq 2^m < S_j(\lambda + 1)$ for all $a,\ b$, where m is the size of the mantissa (53 bits). The maximal sum is reached for $\max_{a \in \mathbb{F}_p}(a) = \max_{b \in \mathbb{F}_p}(b) = p - 1$. We have thus:

$$\lambda \cdot (p-1)^2 \leq 2^m < (\lambda + 1) \cdot (p-1)^2 \tag{2}$$

$$\Leftrightarrow \lambda = \left\lfloor \frac{2^m}{(p-1)^2} \right\rfloor \tag{3}$$

Algorithm 7: λ-algorithm

Input : Two vectors `a` and `b` of size N. The prime number p.
A parameter λ such that $0 \leq \lambda \leq N$

```
1  DotProduct(a, b, N, p, λ)
2      res = 0
3      for j in 1, ..., ⌊N/λ⌋ do
4          for i in 0, ..., λ do
5              res ← res + (a[λ * j + i] · b[λ * j + i]) mod p // RED1
6          res ← res mod p
7      for i in λ⌊N/λ⌋, ..., N do
8          res ← res + (a[i] · b[i]) mod p
9      res ← res mod p
10     return res mod p // RED3
```

Like in the previous algorithm, we have the option to reduce after the multiplication (`RED1`). If we do not reduce, it is equivalent to compute a rational dot-product between the sub-vectors of size λ. We can thus leverage efficient linear algebra libraries that fine-tune the cache usage for us, and

Algo	Only RED3	DMR without RED1	Only RED2
Numbers of modular reductions	1	$\lceil \frac{N}{\lambda} \rceil$	N
Maximum bit size	$2f + \lfloor \log_2(N-1) \rfloor + 1$	$2f + \lfloor \log_2(\lambda - 1) \rfloor + 1$	$2f + 1$

Algo	Only RED1	DMR with RED1	RED1 & RED2
Numbers of modular reductions	$N + 1$	$N + \lceil \frac{N}{\lambda} \rceil$	$2N$
Maximum bit size	$f + \lfloor \log_2(N-1) \rfloor + 1$	$f + \lfloor \log_2(\lambda - 1) \rfloor + 1$	f + 1

Table 2: Comparison in bits requirements and number of modular reductions for different dot product algorithms

compute the dot-product efficiently. This version of dot product is more suited for parallelisation, since we can execute each for loop asynchronously.
To sum up, I listed in 10 all the variants of the modular dot product algorithm with the number of modular reductions required as well as the maximum bit size taken by the temporary.
Notice that we actually use twice the number of bits for the multiplication if we do RED1 too. So the actual bit requirements for RED1 algorithms are: $\max(2f, f + \lfloor \log_2(min(N, \lambda) - 1) \rfloor + 1)$.
For the reasons stated above, and similar to existing libraries, we have chosen the lambda algorithm. I have based my dot product routine on the λ-algorithm.

4.3.2. Other approaches

In [Dum04] J.-G. Dumas, proposed another DMR technique to let the overflow happen and correct the result. The error of overflow is always $2^m \bmod p$ with m being the number of bits of the trailing significant field and can be precalculated. The overflow is easily detected in this case, see 8.

Algorithm 8: Overflow-detection trick

Input : Two vectors `a` and `b` of size N. The prime number p.

```
1 CORR ← 2^m mod p // m is the number of bits of the T.S.F.
2 DotProduct(a, b, N, p)
3     res = 0
4     for i in 0, ..., N do
5         product ← a[i] · b[i]
6         res ← res + product
7         if res < product then
8             res ← res + CORR
9     return res
```

Since we want to optimize dot products intervening in matrix computation, we will have to reduce synchronously the dot products to maximize performance and parallelism, thus I have not considered further the overflow detection trick, which relies on a conditional statement at every iteration. A further work will consist in investigating hybrid approaches possibilities, that is to combine the overflow detection with a block approach.
In [JG10], the authors use a special instruction denoted FMA (*fused multiply-add*) to evaluate exactly the round-off term of the floating-point product. This operation performs:
$\text{FMA}(a, b, c) = a \times b + c$ They describe mainly two algorithms TwoProduct.
This algorithm could be used to compute the dot product separately on the lower and higher part like a multiprecision floating-point number.

4.3.3. BLAS Libraries in Dot Product

The λ-algorithm computes several small dot-products that we can compute in parallel. We store the partial results in a cache, before summing the intermediate dot-products to get the result.

Algorithm 9: TwoProduct

Require $a, b \in \mathbb{F}$, $a, b \geq 0$
Ensure $x \in \mathbb{F}$ *and* $y \in \mathbb{F}$ *such that* $ab = x + y$
$x \leftarrow \text{fl}(ab)$
$y \leftarrow \text{FMA}(a, b, -x)$
return (x, y)

In CUDA, we first used a `__shared__` table or array, so that threads access in parallel this same table. We then need to sum the elements in this table efficiently with a *reduction*. To do this, we can sum elements of the array two by two, and store the results in the first half of the array. We repeat the process with the half-sized array.
I have first proceed by implementing in C, the λ-algorithm with BLAS. I have replaced the second for loop by a call to the `ddot` routine of the cBLAS library with the openBLAS implementation.

4.3.4. Matrix Product

In the Block-Wiedemann Algorithm, we want to multiply a matrix $m \times k$ where $k = D \simeq 10^4$ and $m \simeq k/3$, with a matrix which has either $n = 32$ or $n = 64$ columns and K columns. We also call this second matrix a *fat vector* due to his reduced number of columns.
The λ-algorithm is easily adapted for the matrix product. We can compute the matrix product by block of matrices with λ columns, reduce coefficients modulo p and add each products together.

$$C = A * B = \begin{pmatrix} A_1 & \cdots & A_q & A_{q+1} \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ \vdots \\ B_q \\ B_{q+1} \end{pmatrix} = \sum_{i=1}^{q} A_i * B_i + A_{q+1} * B_{q+1} \tag{4}$$

If $n = \lambda q + r$, we denote the A_i's $m \times k$ block matrices B_i's $k \times n$ and A_{q+1} and B_{q+1} the $m \times r$ and $r \times n$ last matrices respectively. The matrix multiplications $A_i * B_i$ are computed in R. There is no difference, apart the entries from the λ-algorithm described in 7
For the matrix multiplication we have used the CUBLAS library. It is designed to be cache-efficient, and to optimize data locality for the GPU architecture. For the modular reduction of the matrix, I have written a kernel which reduces modulo p each coefficient of the matrix using 5.
I have fixed the number of threads at 256. I have not looked so far at the evolution of the performance of the modular reduction kernel in function of this parameter.
The usual CUBLAS routine to multiply matrices with `double` numbers is the `dgemm` routine. It stands for *double general matrix multiplication*.
For reference, I have included the listing of the matrix kernel in the appendix, see listing B on page 25. For the benchmarks, I have fixed $k = 3M$, $n = 32$ and increased the number of rows by steps of 200.
As a comparison, I have included the CPU timings in figure 5. We observe that the performance is the order of a few GFlops as opposed to about two hundred GFlops. The FFLAS library obtain better performance than the other modular field libraries (NTL and FLINT), due to the use of BLAS routine for cache optimisation and to a better asymptotic algorithm (they use a few rounds of the Winograd variant of the Strassen algorithm). We have not used something similar yet for our kernel.

4.3.5. Multi-Word Algorithm

To target larger primes p, I have written a multi-word version of our matrix algorithm. Let us decompose our A and B matrices into two matrices with smaller coefficients.

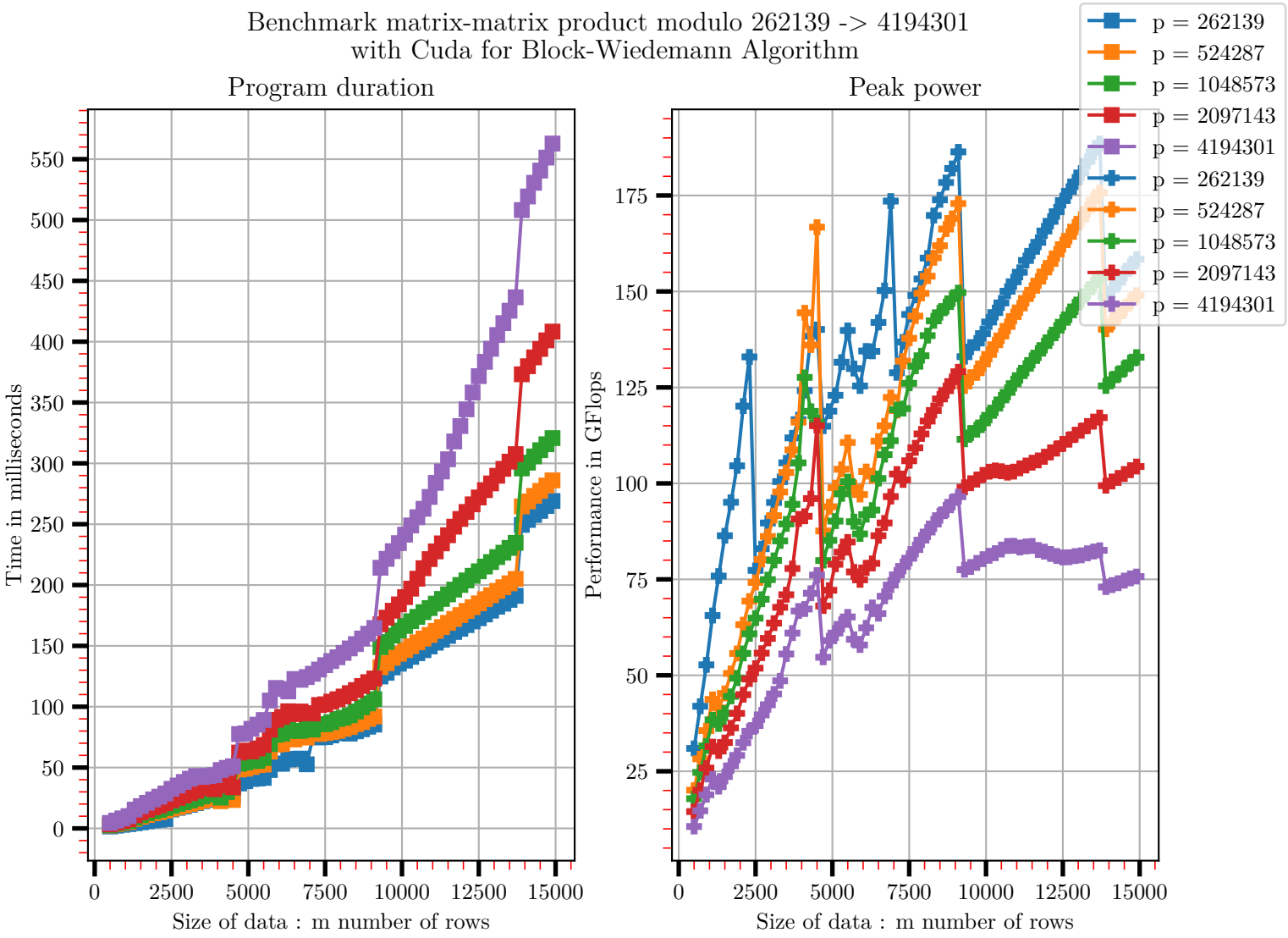


Figure 4: Timings and performance rate in Gflops of my algorithm for modular matrix multiplication with reduction on the Quadro RTX 8000

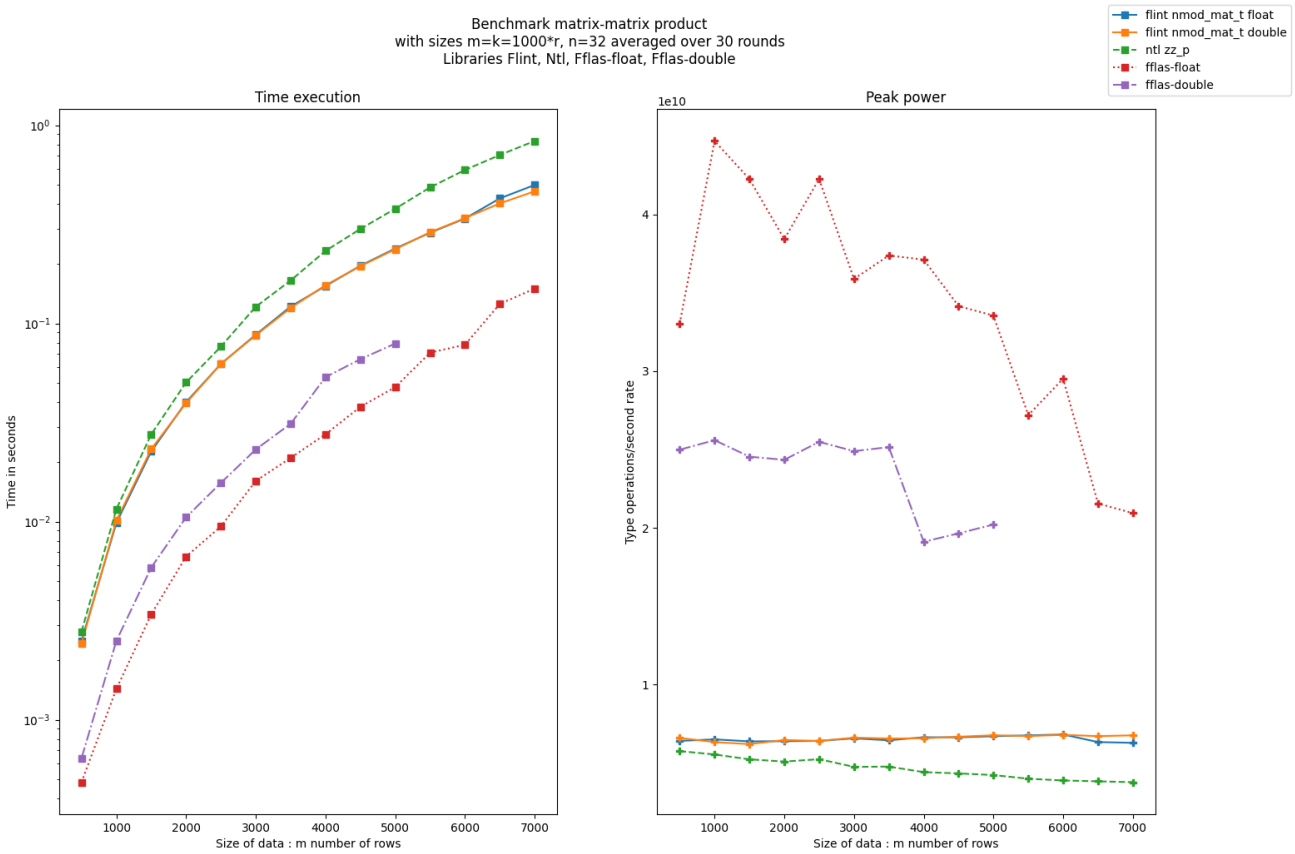


Figure 5: Timings and performance rate in Gflops of my algorithm for modular matrix multiplication with reduction on CPU

$$A = 2^{t/2} \cdot A_h + A_l \tag{5}$$
$$B = 2^{t/2} \cdot B_h + B_l \tag{6}$$

We get these two expressions for the matrix product $C = A \cdot B$:

- Naive
$$C = 2^t \cdot A_h B_h + 2^{t/2}(A_h \cdot B_l + A_l \cdot B_h) + A_l \cdot B_l \tag{7}$$
- Karatsuba
$$C = 2^t \cdot A_h B_h + 2^{t/2}[(A_h - A_l) \cdot (B_l - B_h) + A_h B_h + A_l B_l] + A_l B_l \tag{8}$$

The Karatsuba expression requires only 3 multiplications but introduces additionnal operations and temporaries to the usual expression of the matrix product. More precisely, I described below the number of operations for each expressions:

- 3 $\otimes$ with $\mathcal{O}(n^\theta)$ where θ is a feasible exponent.
- 2 $\ominus$, 3 $\oplus$ and two $\ll$ with modular reduction, all are $\mathcal{O}(n^2)$ operations.

Karatsuba extra cost is compensated by the difference in exponent.

Pseudo-codes We present the algorithms in pseudo-code below:

Algorithm 10: Multi-Word decomposition

Input : A matrix of `double`, b64 format, t decomposition parameter
Output: $(A_h,\ A_l)$ s.t. $A = 2^{t/2} \cdot A_h + A_l$ `double` matrices in b64 format

def `MWDecomp`(A)**:**
  $A_h \leftarrow$ `dfloor`(`DSCAL`$(A,\ 2^{-t/2})$) `/* `$A_h \leftarrow A$ `div` 2^u `*/`
  $A_l \leftarrow A -$ `DSCAL`$(A_h, 2^{t/2})$
return $(A_h,\ A_l)$

Algorithm 11: First Multi-Word algorithm: Karatsuba

Input : A, B matrices of `double`, b64 format
Output: $C = AB$, `double` b64 format

def `MWFFMatMul`**:**
  $M_1 \leftarrow (A_h B_h) \bmod p$ $M_2 \leftarrow (A_l B_l) \bmod p$
  $D_1 \leftarrow (A_l - A_h) \bmod p$
  $D_2 \leftarrow (B_h - B_l) \bmod p$
  $M_3 \leftarrow D_1 D_2 + M_1 + M_2$
  $M_3 \leftarrow M_3 \bmod p$
  $M_3 \leftarrow (2^{t/2} \cdot M_3) \bmod p$
  $M_1 \leftarrow (2^t \cdot M_1) \bmod p$
  $C \leftarrow M_1 + M_3 + M_2$
  $C \leftarrow C \bmod p$
  return C

On the figure 4, I have depicted the timings in seconds and performance in floating point operation per seconds (GFlops) of the algorithm for modular matrix multiplication with the aforementioned parameters, run on a Quadro RTX 8000. The performance is equal to:

$$\text{perf} = \frac{2 * mkn}{10^9 \cdot \tau},$$

where τ is the time in seconds. The performance is drawn on a log scale, so it scales exponentially and should converge to the maximal theoretical performance as the size of the matrices increase. We see three drops in the timings for size of about 4000, 7000 and 12000. We think it is due to a change of algorithm between the sizes. These changes keep the timings under 200 seconds, boosting the performance as the sizes increase.
There are five plots, for which I have used different characteristics, varying in bit size from 18 to 22 bits. I have taken each time the biggest prime for each bit size k, that is the lowest r for which $2^k - r$ is prime. Each time, I have also changed the lambda parameter accordingly, that is I have fixed λ to 2^{25-k} (it is the maximum size λ can take for the result to be exact). λ goes from $2^7 = 128$ to $2^3 = 8$. I was expecting the performance to drop as the bit size increase, since there are more reductions with a smaller λ. It went the opposite direction, the performance is better as the bit size of the characteristic increase.

5. Conclusion

The exact setting of the problem was unclear at the beginning of the internship. By discussing the parameters of the problem, we understood we could not leverage sparse matrix arithmetic for matrix multiplication for efficient implementation in Sparse-FGLM. I have compared existing libraries (FFLAS, NTL, FLINT) over CPU to multiply matrices over finite fields. I have written a GPU kernel for modular matrix multiplication using efficient floating-point arithmetic similar to those CPU libraries. I have benchmarked this kernel and obtained performance about one hundred more performant than on CPU. There is still room for improvement. To avoid wrong primes for CRT decomposition, we must adapt efficient techniques for larger characteristics. One solution would be to do multiprecision arithmetic with floating-point integers. Furthermore, we could improve our kernel by launching asynchronously our calls to CuBLAS, leveraging more the parallelism of the GPU. Sparse-FGLM is not the only algorithm for the polynomial system solving that could beneficiate from efficient GPU implementation, and we seek to improve F_4 in the future.

A. Glossary and References

RNS: Residue Number System

CRT: Chinese Remainder Theorem

DMR: Delayed Modular Reduction

BLAS: Basic Linear Algebra Subprograms: set of standardized functions with optimized implementations of linear algebra functions separated in three levels (scalar, vector and matrix operations).

MSB/LSB: Most (/Less) Significant Bit. Bit with the highest weight of a number. Leftmost bit with value 1 in a big endian representation, rightmost in a little endian representation.

GPU: General Processing Unit. Processor inside a graphics card.

CPU: Main processor of a computing unit.

HPC: High Performance Computing.

Gröbner Basis: Formal computation tool used in solving a polynomial system, it is a 'good' basis of an ideal of polynomials.

F4: Algorithm designed by Jean-Charles Faugère to find a Gröbner Basis of the polynomial ideal generated by a polynomial system.

FGLM: Algorithm to find a Gröbner basis for another monomial ordering.

SIMD: Single Instruction Multiple Data.

LEX: Lexicographic monomial ordering.

DRL: Degree Reverse Lexicographic monomial ordering.

References

[1] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". In: *Advances in Cryptology — CRYPTO' 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323. ISBN: 978-3-540-47721-1.

[2] Jérémy Berthomieu, Vincent Neiger, and Mohab Safey El Din. *Faster change of order algorithm for Gröbner bases under shape and stability assumptions*. 2022. DOI: 10.48550/ARXIV.2202.09226. URL: https://arxiv.org/abs/2202.09226.

[3] *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] Jean-Guillaume Dumas. "Efficient dot product over word-size finite fields". In: *CoRR* (May 2004), p. 13.

[5] Jean-Guillaume Dumas, Clément Pernet, and Zhendong Wan. "Efficient Computation of the Characteristic Polynomial". In: (July 2005), pp. 140–147.

[6] J.C. Faugère, P. Gianni, D. Lazard, and T. Mora. "Efficient Computation of Zero-dimensional Gröbner Bases by Change of Ordering". In: *Journal of Symbolic Computation* 16.4 (1993), pp. 329–344. ISSN: 0747-7171. DOI: https://doi.org/10.1006/jsco.1993.1051. URL: https://www.sciencedirect.com/science/article/pii/S0747717183710515.

[7] Jean-Charles Faugère. "A new efficient algorithm for computing Gröbner bases (F_4)". In: *Journal of Pure and Applied Algebra* (1999), p. 28.

[8] Jean-Charles Faugère. "A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5)". In: *Proceedings of the 2002 international symposium on Symbolic and algebraic computation - ISSAC '02*. the 2002 international symposium. Lille, France: ACM Press, 2002, pp. 75–83. ISBN: 978-1-58113-484-1. DOI: 10.1145/780506.780516. URL: http://portal.acm.org/citation.cfm?doid=780506.780516 (visited on 02/13/2022).

[9] Jean-Charles Faugère and Chenqi Mou. "Fast Algorithm for Change of Ordering of Zero-Dimensional Gröbner Bases with Sparse Multiplication Matrices". In: *Proceedings of the 36th International Symposium on Symbolic and Algebraic Computation*. ISSAC '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 115–122. ISBN: 9781450306751. DOI: 10.1145/1993886.1993908. URL: https://doi.org/10.1145/1993886.1993908.

[10] Jean-Charles Faugère and Chenqi Mou. "Sparse FGLM algorithms". In: *Journal of Symbolic Computation* 80 (2017), pp. 538–569. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2016.07.025. URL: https://www.sciencedirect.com/science/article/pii/S0747717116300700.

[11] A.S. Fraenkel and Y. Yesha. "Complexity of problems in games, graphs and algebraic equations". In: *Discrete Applied Mathematics* 1.1 (1979), pp. 15–30. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(79)90012-X. URL: https://www.sciencedirect.com/science/article/pii/0166218X7990012X.

[12] The FFLAS-FFPACK group. *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package*. v2.4.1. http://github.com/linbox-team/fflas-ffpack. 2019.

[13] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

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

[15] Joris van der Hoeven, Grégoire Lecerf, and Guillaume Quintin. "Modular SIMD arithmetic in Mathemagix". In: *CoRR* abs/1407.3383 (2014). arXiv: 1407.3383. URL: http://arxiv.org/abs/1407.3383.

[16] Seung Gyu Hyun, Vincent Neiger, Hamid Rahkooy, and Éric Schost. "Block-Krylov techniques in the context of sparse-FGLM algorithms". In: *arXiv:1712.04177 [cs]* (Jan. 15, 2019). arXiv: 1712.04177. URL: http://arxiv.org/abs/1712.04177 (visited on 02/13/2022).

[17] Jérémy Jean and Stef Graillat. "A Parallel Algorithm for Dot Product over Word-Size Finite Field Using Floating-Point Arithmetic". In: *2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2010). Timisoara: IEEE, Sept. 2010, pp. 80–87. ISBN: 978-1-4244-9816-1. DOI: 10.1109/SYNASC.2010.10. URL: http://ieeexplore.ieee.org/document/5715272/ (visited on 02/13/2022).

[18] Walter Keller-Gehrig. "Fast algorithms for the characteristics polynomial". In: *Theoretical Computer Science* 36 (1985), pp. 309–317. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(85)90049-0. URL: https://www.sciencedirect.com/science/article/pii/0304397585900490.

[19] Junjie Li, Sanjay Ranka, and Sartaj Sahni. "Strassen's Matrix Multiplication on GPUs". In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS* (Dec. 2011), pp. 157–164. DOI: 10.1109/ICPADS.2011.130.

[20] Marc Moreno Maza and Wei Pan. "Solving Bivariate Polynomial Systems on a GPU". In: *Journal of Physics: Conference Series* 341 (Feb. 2012), p. 012022. DOI: 10.1088/1742-6596/341/1/012022. URL: https://doi.org/10.1088/1742-6596/341/1/012022.

[21] Peter L. Montgomery. "Modular multiplication without trial division". In: (1985).

[22] Cheon-Hyeon Park, Gershon Elber, Ku-Jin Kim, Gye-Young Kim, and Joon-Kyung Seong. "A hybrid parallel solver for systems of multivariate polynomials using CPUs and GPUs". In: *Computer-Aided Design* 43.11 (2011). Solid and Physical Modeling 2011, pp. 1360–1369. ISSN: 0010-4485. DOI: https://doi.org/10.1016/j.cad.2011.08.030. URL: https://www.sciencedirect.com/science/article/pii/S0010448511002296.

[23] Victor Shoup. *NTL: a library for doing numbery theory*. 2021. URL: http://www.shoup.net.

[24] Allan Steel. "Direct Solution of the (11,9,8)-MinRank Problem by the Block Wiedemann Algorithm in Magma with a Tesla GPU". In: *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*. PASCO '15. Bath, United Kingdom: Association for Computing Machinery, 2015, pp. 2–6. ISBN: 9781450335997. DOI: 10.1145/2790282.2791392. URL: https://doi.org/10.1145/2790282.2791392.

[25] Alexandre Valencia-Blatère. "Développement et optimisation dune bibliothèque dalgèbre linéaire sur GPU". July 2020.

[26] Jianjun Wei and Liangyu Chen. "Optimized Multivariate Polynomial Determinant on GPU". In: (2020). DOI: 10.48550/ARXIV.2010.12117. URL: https://arxiv.org/abs/2010.12117.

[27] Wang Zhang Xianyi, Werner Saar Qian, and Kazushige Goto. *OpenBLAS*. 2011. URL: https://www.openblas.net/.

B. Code Listings

```
__host__
void ffmatMulWtcublas(cublasHandle_t handle, const double a[], const double b[],
                      const int m, const int k, const int n, const double u,
                      const double p, const uint32_t lambda, double c[]) {
  /* Compute the multiplication of matrices a and b of size m*k and k*n,
     in prime field of characteristic p with partial calls to CuBLAS dgemm.
     and returns result in matrix c, array of size m*n.
     u is the floating-point number 1.0/p given as an argument for precomputation.
     lambda is the maximum amount of summations we can do before remaindering.
     The larger it is, the more optimisations cublas can do.
   */
  unsigned long long int bound = k/lambda;

  // The alpha and beta parameters for dgemm: C <- \alpha*A*B + \beta*C
  double one = 1.0; double *alpha = &one;
  double zero = 0.0; double *beta = &zero;

  cublasStatus_t stat;

  double *partial_matrix;
  // HANDLE_ERROR is a macro exiting in case of a failed allocation
  HANDLE_ERROR(cudaMalloc(&partial_matrix, m*n*sizeof(double)));
  int nb_threads = 256;
  int numBlocks = n * m / nb_threads;
  for (uint32_t j=0; j < bound; ++j) {
    stat = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
        m, n, lambda,
        alpha, a+j*m*lambda, m,
        b+j*lambda, k,
        beta, partial_matrix, m);

    // Threaded version of modular reduction of each coefficients of the matrix
```

```
    gpu_matrix_fma_reduction<<<numBlocks, nb_threads>>>(partial_matrix, m, n, u, p);

    // Add the current block product result into the buffer
    stat = cublasDgeam(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n,
        alpha, partial_matrix, m,
        alpha, c, m,
        c, m);
  }
  gpu_matrix_fma_reduction<<<numBlocks, nb_threads>>>(c, m, n, u, p);
  cudaFree(partial_matrix);
}
```