SCIENCES SORBONNE UNIVERSITÉ

Master 1 Project

Gaussian Elimination High-Performance Implementation

Mohamed Imad Eddine Ghodbane
supervised by Vincent Neiger

Abstract

Gaussian elimination in finite fields emerges as a critical algorithm, essential for solving systems of linear equations and performing matrix operations over finite fields. This work presents a high-performance implementation of Gaussian elimination over a finite field $\mathbb{F}_p$ with a prime modulus p. We first develop efficient arithmetic operations in the finite field, utilizing the AVX2 instruction set for vectorization. These operations are then integrated into specialized PLUQ algorithms, where Gaussian Elimination serves as the main operation. Our implementation demonstrates a significant performance improvement, achieving a speedup of 7x over basic implementations (without AVX2). Moreover, for prime numbers stored on 29 bits, our implementation outperforms the FLINT library's LU factorization by a factor of 8x. These results offer valuable insights into optimizing linear algebra libraries, particularly when dealing with matrices of intermediate dimensions.

1 Introduction

Linear algebra provides a powerful framework for solving systems of linear equations and studying geometric objects such as vectors, matrices, and linear transformations. One of its key strengths lies in the development of efficient algorithms for manipulating these objects. Choosing the right algorithm can indeed significantly impact the performance of computations. We start by comparing two algorithms for matrix factorization: the PLUQ algorithm and the Crout PLUQ algorithm [2]. Both algorithms use the Gaussian elimination and both are used for LU decomposition with partial pivoting, where a matrix is decomposed into a product of a row permutations matrix P, a lower triangular matrix L, an upper triangular matrix U, and a column permutations matrix Q. The PLUQ algorithm and the Crout PLUQ algorithm have different characteristics that may affect their suitability for vectorization. By comparing the performance of these two algorithms under vectorized operations, we can gain insights into their efficiency and identify the most suitable algorithm for high-performance implementations over finite fields.
In the context of linear algebra algorithms over finite fields, modular operations play a crucial role in ensuring the arithmetic operations stay within the bounds of the finite field. These operations include addition, subtraction, multiplication, and inversion, all of which are essential for our Implementation. Once the modular operations are successfully vectorized, the main algorithm can be more easily vectorized by applying the same principles. This involves identifying opportunities for parallelism within the algorithm, such as processing multiple rows or columns of a matrix simultaneously. By leveraging SIMD instructions, we can accelerate the computation of LU decomposition with partial pivoting, leading to significant performance gains over basic implementations.

The document is structured as follows:

- **Section 2:** discusses the PLUQ factorization, a specialized variant of LU decomposition. It also introduces the Crout PLUQ algorithm as an alternative approach.
- **Section 3:** explores how modular operations in finite fields can be efficiently vectorized using SIMD instructions, highlighting the challenges and strategies involved.
- **Section 4:** delves into the implementation details, including the data structures used to store matrices and insights into implementing the PLUQ and Crout PLUQ algorithms.
- **Section 5:** presents numerical results and performance analyses, demonstrating the significant speedup achieved by our optimized implementation compared to basic implementations and existing libraries like FLINT [4].
- **Section 6:** summarizes the key findings and contributions of this work, as well as potential avenues for future optimizations.

2 Gaussian Elimination

The first step in the Gaussian Elimination is to choose a pivot element, which must be nonzero and invertible. As we are working over a field $\mathbb{F}_p$, where p is prime, all nonzero elements are invertible. Therefore, any nonzero element in $\mathbb{F}_p$ can be chosen as a pivot. So, in each iteration, we need to locate the first non-zero entry in the row. Assuming we are in iteration k, and let the entries of the

matrix A be a_{ij} with $0 < i, j < n$, this means that we will check if $a_{kk} \neq 0$. If this condition is not met, it implies that the pivot element a_{kk} is zero. In such a scenario, we need to perform column rotation. The algorithm involves computing the multiplication of the inverse multiplicative of the pivot modulo p with all the entries of the same column k as the pivot, denoted as a_{ik}, where the pivot row is k and i ranges from the pivot row $+1$ to $n - 1$.
Consider a matrix A of size $n \times n$ over $\mathbb{F}_p$, where $A = [a_{ij}]$, if the pivot element is a_{kk}, then for each i such that $k < i < n$, we perform the operation:

$$a_{ik} \leftarrow a_{ik} \times a_{kk}^{-1} \mod p$$

This operation is suitable for vectorization, as it involves a fixed number of operations per column and can be efficiently implemented using vector instructions. We can utilize a vector that holds the value of the inverse pivot to perform this operation on multiple elements of the column simultaneously.
After computing a_{ik}, with $k < i < n$, for each row $l > k$, we compute the multiplication of a_{ik} with the row of the pivot starting from the pivot to $n - 1$. Assuming we are computing this for the row l, we need to subtract the multiplication of a_{lk} with a_{kj}, with $k < j < n$, from the entries of the row l, a_{lj}.
This can be represented as:

$$a_{lj} \leftarrow a_{lj} - (a_{lk} \times a_{kj}) \mod p \quad \text{for} \quad k < j < n$$

We will keep performing the same operations in each iteration of Gaussian elimination. Utilizing SIMD instructions enables performing the same operation on multiple data elements simultaneously, which is precisely what we need for both computing $a_{ik} \leftarrow a_{ik} \times a_{kk}^{-1} \mod p$ and $a_{lj} \leftarrow a_{lj} - (a_{lk} \times a_{kj}) \mod p$ in parallel for multiple elements.

Algorithm 1: Crout PLUQ Algorithm

```
Data: Input matrix $A$ of size $m \times n$ in the field $\mathbb{F}p$
Result: Matrix $L$ representing the lower triangular part of $A$ and matrix
        $U$ representing the upper triangular part of $A$ with
        permutation matrices $P$ and $Q$ and rank of $A$
rank ← 0; nullity ← 0; while $rank + nullity < m$ do
    for $j = rank$ to $n$ do
        sum ← 0; for $k = 0$ to $rank$ do
            sum ← sum + $(a$rank$, k \times a_{k,j})$ mod $p$;
        end
        $a_{\text{rank},j} \leftarrow a_{\text{rank},j} - \text{sum} \mod p$;
    end
    Find the pivot column index ; if $pivot == n$ then
        Perform row rotation on $A$ and update $P$; nullity + +; else
            for $i = rank + 1$ to $m - nullity$ do
                sum ← 0; for $j = 0$ to $rank$ do
                    sum ← sum + $(a_{i,j} \times a_{j,\text{pivot}})$ mod $p$;
                end
                $a_{i,\text{pivot}} \leftarrow a_{i,\text{pivot}} - \text{sum}$ mod $p$; $a_{i,\text{pivot}} \leftarrow$
                 $a_{i,\text{pivot}} \times a^{-1}_{\text{rank},\text{pivot}} \mod p$;
            end
            Perform column rotation from rank to pivot and update $P$;
            rank + +;
        end
    end
end
```

Algorithm 2: PLUQ Algorithm

```
Data: Input matrix $A$ of size $m \times n$ in the field $\mathbb{F}_p$
Result: Matrix $L$ representing the lower triangular part of $A$ and matrix
        $U$ representing the upper triangular part of $A$ with
        permutation matrices $P$ and $Q$ and rank of $A$
rank ← 0; nullity ← 0; while $rank + nullity < m$ do
    Find the pivot column index ; if $pivot = n$ then
        Perform row rotation on $A$ and update $P$; nullity + +;
    else
        if $pivot \neq rank$ then
            Perform column transposition on $A$ and update $Q$;
        end
        for $k = rank + 1$ to $m$ do
            $a_{k,rank} \leftarrow a_{k,rank} \times a^{-1}_{rank,rank}$ mod $p$; for $j = rank + 1$ to $n$ do
                $a_{k,j} \leftarrow a_{k,j} - (a_{k,rank} \times a_{rank,j})$ mod $p$;
            end
        end
        rank++;
    end
end
end
```

2.1 PLUQ algorithm

Let A be the input matrix of size $m \times n$, where $A = [a_{ij}]$. Additionally, let P and Q be permutation matrices representing row and column permutations, respectively. The algorithm 2 iterates through

each row of the matrix, updating the rank and nullity counters accordingly. For each row, it finds the pivot column index and performs row rotation if the pivot column is all zeros or column transposition otherwise. It then applies row operations to update the pivot column and subtracts the appropriate products to achieve triangular form. After completing the operations for all rows, the algorithm outputs matrices L and U along with permutation matrices P and Q, and the rank of matrix A.
The algorithm 2 uses two loops, one iterates over the rows while the other iterates over the columns. This is what puts some challenges in choosing the type of storing the matrix, whether as major rows or as major columns. We need to look at the most expensive operation. Then we can observe that the inner loop is executed more times than the outer loop and involves more operations such as multiplication and subtraction, while in the outer loop we only perform multiplication. Considering this, using row-major storage is the best choice.
As the outer loop operations are negligible compared to the inner loop, we need to focus more on vectorizing the inner loop, because vectorizing the outer one will not have any effect.

2.2 Crout PLUQ algorithm

The Crout PLUQ method has shown good potential for vectorization using AVX2. The algorithm 1 demonstrates how it actually generates the PLUQ factorization.
This algorithm accomplishes the same task as the previous one but employs different operations, notably utilizing scalar product in the initial inner loop. The outer loop traverses the columns of the matrix from the *rank* to the n, while at the start of each iteration, the sum variable accumulates scalar product results. The update of $a_{\text{rank},j}$ involves subtracting the computed sum (modulo p) from its current value. This inner loop effectively computes new values for each element in the rank column of matrix A by utilizing scalar products with elements from the *rank* row of matrix A. The second loop iterates over rows starting from the row after the *rank* row up to $m - nullity$ row. At each iteration, it initializes 'sum' to zero, accumulating scalar product results between elements in the current row and corresponding pivot column elements.
Similar to the first loop, defining the type of storage to store the matrix isn't straightforward due to the simultaneous iteration and utilization of rows and columns within each loop. The algorithm's structure necessitates accessing and modifying elements across rows and columns interchangeably during the operations within each loop. Therefore, a simple row-major or column-major storage format may not be optimal for efficiently accessing and manipulating matrix elements in this context.

3 Modular Operations

In modular computation, all arithmetic operations, including addition, subtraction, multiplication, and inversion, are performed modulo a prime number p. This approach ensures that the results remain within a fixed range, providing computational efficiency and facilitating error handling. Let p be a nonnegative integer of bit-size at most m, where m is less than or equal to n the bit size of integers in the implementation. For optimization purposes, n and m are assumed to be known at compilation time, while the actual value of p is determined only at execution time. Given that dynamically selecting the appropriate modular arithmetic algorithm based on the bit-size of p incurs considerable overhead, we opt to predefine n as 32 bits and m as 30. This decision aims to mitigate potential issues arising from arithmetic overflow during addition and subtraction operations. In 32-bit arithmetic, the multiplication of two 30-bit integers may exceed the representable range, leading to inaccuracies or unexpected behavior. By restricting m to 30 bits, we ensure that the product of two integers within this range remains within bounds using an intermediate integer with bit-size$= 2n$ to store the result of the multiplication then use modular reduction to keep it in the fixed range($0 <$ result $< p$), thus preventing potential overflows. In this framework, modulo p integers are stored as their representatives in the set $\{0, 1, ..., p-1\}$, allowing for straightforward manipulation within the designated finite field.

3.1 Modular addition and subtraction

Given two integers a and b in $\mathbb{F}_p$, the sum $a+b$ is computed by performing the addition in the usual way and then taking the result modulo p. If the result exceeds p, it is adjusted by subtracting p, ensuring that the sum remains within the range $\{0, 1, \ldots, p-1\}$. Since both a and b are $\leq p-1$, represented in 30 bits each, the sum $a+b \leq 2p-2$, represented at most in 31 bits. Therefore, the result $(a+b)-p \leq p-2$.
The algorithm presented for addition modulo p offers computational advantages over the Euclidean division method, particularly in terms of processor efficiency. In Euclidean division, the modulo operation involves costly division operations, which can be computationally intensive and time-consuming, especially for large values of p. These division operations typically require more processor cycles compared to simpler arithmetic operations like addition and subtraction. This algorithm efficiently computes the sum modulo p without the overhead associated with division.

Function 1
Input: a and b 32-bit integers modulo p.
Output: $(a+b) \mod p$.

```
add_mod(a, b, p) {
    1. res = a + b;
    2. return (res >= p) ? res - p : res;
}
```

Similarly to the modular addition, given two integers a and b in $\mathbb{F}_p$, the difference $a-b$ will not result in an overflow, and it is computed by performing the subtraction in the usual way and then taking the result modulo p. If the result is negative, it is adjusted by adding p, ensuring that the result remains within the range $\{0, 1, \ldots, p-1\}$.

Function 2
Input: a and b 32-bit integers modulo p.
Output: $(a-b) \mod p$.

```
sub_mod(a, b, p) {
    1. res = a - b;
    2. return (res < 0) ? res + p : res;
}
```

3.2 Vectorized modular subtraction

When revisiting the previous section, it becomes evident that addition is only used in the scalar product, which we will discuss later. Instead, our focus will be directed towards optimizing the subtraction operation. To optimize the modular subtraction operation from a scalar implementation to a vectorized form using AVX2 instructions, we transitioned from a straightforward scalar approach to highly efficient 128-bit and 256-bit vectorized implementations. We chose to use 128-bit vectors rather than 256-bit vectors because the multiplication of two integers requires double the bit size to hold the result. Hence, for 256-bit vectors, each multiplication operation would need to handle 4 32-bit integers, and as we are using these two operations in Gaussian elimination, the number of multiplications at one time should be the same as the number of subtractions at one time. In this case, 4 32-bit integers in one packed vector. In the 256-bit implementation, subtraction needs to be computed in a separate loop. In this scenario, utilizing a 256-bit vector could prove to be the most advantageous option. This implementation takes two 128-bit vectors va and vb modulo p as input and outputs the element-wise subtraction of these vectors modulo p. Inside the function, the vectors are subtracted element-wise using the `mm_sub_epi32` function, then compared with 0 using `mm_cmplt_epi32` to generate a mask. This mask is used to conditionally blend the result of the subtraction with the sum between the result and p, achieved through addition with vp. The final result is returned as a 128-bit vector.

Function 3
Input: va and vb 128-bit vectors modulo p.
Output: $(va - vb) \mod p$ 128-bit vector.

```
sub_mod_vec(va, vb, vp) {
    1. res = _mm_sub_epi32(a, b);
    2. mask = _mm_cmplt_epi32(res, _mm_setzero_si128());
    3. adjusted_res = _mm_add_epi32(res, vp);
    4. res = _mm_blendv_epi8(res, adjusted_res, mask);
    5. return res;
}
```

3.3 Modular Multiplication

We achieve modular multiplication by first computing the product of the two operands, a and b, then utilizing the Euclidean division offered by the modulo (%) operation in C to obtain the result modulo p. This approach ensures that the product remains within the finite field F_p. Given that both a and b are represented as 32-bit integers, with values less than p which is represented in 30 bits, it is crucial to consider the possibility of the result exceeding the representation range. To circumvent this issue, we cast the result into a temporary 64-bit integer, allowing for sufficient space to accommodate the potentially larger product. Subsequently, we take the result modulo p and store it in a 32-bit integer, ensuring that the final result remains within the representation range and adheres to the properties of modular arithmetic. In this version `res` is a 64-bit integer to hold the product, and `mult_mod` returns 32-bit integer.

Function 4
Input: a and b 32-bit integers modulo p.
Output: $(a * b) \mod p$

```
mult_mod(a, b, p) {
    1. res = a * b;
    2. return res % p;
}
```

This implementation cannot be directly vectorized using AVX2 instructions due to the absence of intrinsic functions specifically designed for modular reduction. Instead, we will use the implementation used in [5] [3]. We resort to utilizing numeric types such as float or double to perform the modular multiplication. However, this approach introduces certain drawbacks, notably the need for extra conversions between numeric and integer types. Given that we are working with 32-bit integers, we opt to use double representation with a 52-bit mantissa.
Let x and y be two floating-point numbers, and let $z = x \times y$. We compute an approximation of z starting by calculating the high part $h = x \times y$. We then compute l which represents the low part using fused multiplication $l = \text{fma}(x, y, -h)$ to capture any residual or rounding error that may occur during the computation of the product. Next, we calculate $c = \lfloor \frac{h}{p} \rfloor$, and use this result in $d = h - c \times p$ using fused multiplication $\text{fma}(-c, p, h)$. Then, we compute $e = d + l$ to get the exact result, and adjust it if it's out of the range.

Function 5
Input: x and y 64-bit floating-point numbers modulo p and $u = \frac{1}{p}$
Output: $(x * y) \mod p$

```
mul_mod(x, y, u, p) {
    1. h = a1 * a2;
    2. l = fma (a1, a2, -h);
    3. b = h * u;
    4. c = floor (b);
    5. d = fma (-c, p, h);
    6. e = d + l;
    7. if (e >= p) return e - p;
    8. if (e < 0) return e + p;
    9. return e;
}
```

This function leverages the properties of Montgomery multiplication to achieve modular reduction efficiently, making it suitable for vectorization. By utilizing a precomputed constant u, the function avoids the need for division by p.

3.4 Vectorized modular Multiplication

To vectorize function 6, we use the offered multiplication, addition, and subtraction capabilities provided by the AVX2 architecture, along with the fused multiplication FMA. To adjust the result, two conditions are checked to ensure that the result falls within the specified range. If the result e is greater than or equal to vp, indicating that it exceeds the upper bound of the range, each element of e is subtracted by vp to bring it within the range. This operation is performed using the `mm256_sub_pd` intrinsic, which subtracts the packed elements of vp from the corresponding elements of e. Similarly, if any element of e is less than 0, implying that it falls below the lower bound of the range, each element of e is added by vp to ensure it remains within the specified range. This addition operation is achieved using the `mm256_add_pd` intrinsic, which adds the packed elements of vp to the corresponding elements of e. These adjustments ensure that the final result returned by the function is within the desired range.

Function 6

Input: vx and vy 256-bit double floating-point vectors modulo p.

Output: $(vx * vy) \mod p$ 256-bit double floating-point vector.

```
mul_mod_vec(vx, vy, vu, vp) {
    h = _mm256_mul_pd(x, y);
    l = _mm256_fmsub_pd(x, y, h);
    b = _mm256_mul_pd(h, u);
    c = _mm256_floor_pd(b);
    d = _mm256_fnmadd_pd(c, p, h);
    e = _mm256_add_pd(d, l);
    t = _mm256_sub_pd(e, p);
    e = _mm256_blendv_pd(t, e, t);
    t = _mm256_add_pd(e, p);
    return _mm256_blendv_pd(e, t, e);
}
```

3.5 Modular Inversion

We utilize the extended Euclidean algorithm to compute the multiplicative inverse of an integer in a finite field. Unlike other operations such as modular multiplication, we do not require a vectorized implementation for modular inversion. This is because, during each iteration of the Gaussian elimination algorithm, we only need to compute the inverse of the pivot once. Therefore, the extended Euclidean algorithm suffices for efficiently computing modular inverses.

Function 7

Input: a 32-bit integer modulo p.

Output: a^{-1} the inverse of a modulo p.

```
inverse(a, p) {
    m0 = p;
    t = p;
    q = p;
    x0 = 0;
    x1 = 1;
    while (a > 1) {
       q = a / p;
       t = p;
       p = a % p, a = t;
       t = x0;
       x0 = x1 - q * x0;
       x1 = t; }
    if (x1 < 0)
       x1 += m0;
    return x1;
}
```

4 Implementation

In the C implementation, we opted to define matrices as a single array using major row presentation. To access the element $a_{i,j}$ in matrix A, we employ the expression $A[i \times n + j]$. Leveraging scalar finite field arithmetic, we crafted implementations for two methods: `pluq_inplace`, which computes the decomposition and stores the results within the same matrix. For efficient storage, we chose to represent the P and Q matrices using only the indices of rows and columns, respectively. Thus, the initial values of P and Q, which are the identity matrices, are encoded as arrays ranging from 0 to $m - 1$ for P and $n - 1$ for Q. Additionally, we devised the `PLUQ` function, utilizing `pluq_inplace` to store the result in a different matrix. Within `pluq_inplace_avx2`, we designed the `rows_elimination` function to simultaneously process 4 32-bit integers. This was achieved by loading them using `_mm_loadu_si128` on `&A_data[matrixRank * n + i]`, representing the pointer value. Subsequently, we converted this 128-bit integers vector to a 256-bit double vector via `_mm256_cvtepi32_pd`. This enabled us to perform multiplications using a previously defined multiplication function, which handles the multiplication of pivot row elements with pivot column elements. Following the multiplication, we reverted the result back to a 128-bit vector using `_mm256_cvttpd_epi32`. Then, we executed the subtraction utilizing the `sub_mod_avx2` function to perform reduction on the current row. Finally, we stored the resultant values using the `_mm_storeu_si128` function.
For the Crout method, we developed a vectorized version for the scalar product, capable of processing 4 32-bit integers simultaneously. The process begins by initializing the vector that holds the result to zeros. Since the data is not aligned due to dealing with columns while our matrix is in row-major storage, we utilize the `_mm256_set_epi64x` instruction. Next, we perform element-wise multiplication between the two vectors and accumulate the result by adding it to a temporary array. Once we reach the end, we reduce the values in the temporary array by adding the 4 elements horizontally. We then apply this function to vectorize the two loops in the Crout method. For the first loop, we utilize the `scalar_product_avx2`. Subsequently, we update the matrix by employing the `sub256_avx2` function. This function is similar to `sub128_avx2` but operates on more integers, specifically 8 32-bit integers. We utilize it to vectorize the operation:

$$a_{\text{rank},j} \leftarrow a_{\text{rank},j} - \text{sum} \mod p$$

used in algorithm 1. Similarly, for the second loop, we commence by invoking the `scalar_product_avx2`. Subsequently, we update the matrix by vectorizing these two operations:

$$a_{i,\text{pivot}} \leftarrow a_{i,\text{pivot}} - \text{sum} \mod p;$$
$$a_{i,\text{pivot}} \leftarrow a_{i,\text{pivot}} \times a_{\text{rank},\text{pivot}}^{-1} \mod p;$$

In this process, we need to convert from the 128-bit vector to the 256-bit doubles vector, as done previously. Then, we perform the multiplication using the vectorized version and convert it back to a 128-bit vector to compute the subtraction. Finally, we store it back to the matrix. As we are dealing with columns in this part, we cannot store the result directly due to alignment issues. Therefore, we utilize a temporary array to hold the result and manually assign each result to its position.
For each loop, we unrolled each it to the width of the vector to achieve optimal performance. Additionally, we incorporated a scalar loop to handle the remaining size less than the vector width.

The source code files are:

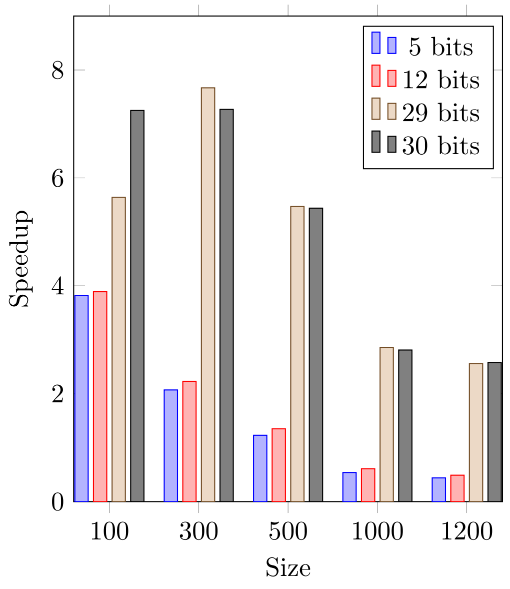
- **PLUQ.c:** contains implementations of various PLUQ algorithms (`pluq_crout_avx2`, `pluq_crout`, `pluq_inplace_avx2`, `pluq_crout`, and `PLUQ`)
- **matrix.c:** create matrix, random matrix, zeros matrix, multiply matrices, compare matrices, print matrix, and copy matrix.
- **finite_field.c:** contains all the finite field arithmetic functions (`add`, `sub`, `mult`, `inverse`).
- **permutations.c:** contains functions for swapping two rows and columns, permuting matrix columns and rows according to a given permutation array, row and column transposition, and row and column rotation.
- **utilities.c:** contains functions to create a range, print an array, check if a matrix is upper or lower triangular, check many PLUQ decompositions to test their correctness, and check one PLUQ decomposition to verify only one PLUQ.
- **avx2.c:** contains all the functions that use AVX2, including `sub_avx2`, `scaller_product_avx2`, `mult_mod_p`, row elimination used in `pluq_inplace`, matrix vector multiplication used in `crout_pluq`, and functions to update the `crout_pluq` function.
- **main.c:** used to run the tests.
- **main_bench.c:** used to run the benchmarks of different algorithms.

5 Performance Analysis and Results

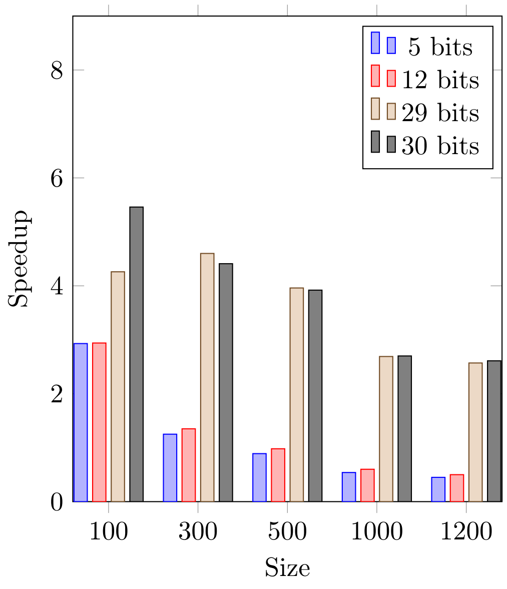
The figure below (Figure 1) compares the speedup between the FLINT implementation and the versions employing the vectorization strategy elucidated earlier, using Ubuntu 24.04 Linux 6.8.0-31-generic, AMD Ryzen™ 7 PRO 7840U with Radeon™ 780M Graphics $\times$ 16. Our implementation takes advantage of the AVX2 capabilities provided by the processor. To compile the C code, we utilized the following flags:

```
-O3 -mfma -mavx2 -lflint -lgmp
```

The grouped bar chart (Figure 1a) illustrates the speedup attained by the vectorized implementation of the Crout PLUQ method in contrast to the FLINT version across various squared matrix sizes and different prime bit lengths. The x-axis represents the sizes of the squared matrices, while the y-axis represents the speedup achieved. Each bar color represents a different prime bit length, providing a visual distinction between the performance of implementations across varying precision levels. The second Chart (Figure 1b) represents the speedup of the basic PLUQ vectorized implementation compared to the FLINT implementation, utilizing the same characteristics as the first chart (Figure 1a).
As observed, for the smallest matrix size of 100, the vectorized Crout version achieves a speedup ranging from 3.82 to 7.25, with the highest speedup observed at a bit length of 30. As the matrix size increases to 300, the speedup remains consistently high, with values ranging from 2.07 to 7.27 across the same bit lengths, with the highest speedup observed at the same bit length (30 bits).
At a size of 500, the speedup begins to diminish and struggles to maintain peak values, fluctuating between 0.89 and 3.96. This indicates that for larger matrices, the FLINT strategy begins to leverage



(a) Crout AVX2 speedup



(b) Pluq AVX2 speedup

Figure 1: AVX2 implementations speedup against FLINT implementation

recursive methods, which prove to be more effective for handling the increased size of matrices. However, for the largest matrix sizes of 1000 and 1200, the speedup tends to decrease, especially for lower bit lengths, indicating potential limitations. It is noteworthy that the FLINT implementation starts to exhibit better execution times at smaller bit lengths, whereas our implementation maintains superiority for larger bit lengths, particularly 29 and 30. This observation suggests that the FLINT implementation may be employing an approach to handle large integers greater than 29, which could contribute to its slower performance in this scenario.
The chart depicting the basic PLUQ vectorized version exhibits similar trends to the Crout method chart, albeit with smaller speedups. The highest speedup is observed at a matrix size of 100 and bit length of 30, mirroring the behavior seen in the Crout method chart. This confirms that the implementation of FLINT encounters difficulties specifically at primes with 29 and 30 bits length and small matrix sizes.
Overall, these observations underscore the effectiveness of the vectorized implementations, particularly at smaller matrix sizes and higher prime bit lengths, it also highlights potential challenges encountered by FLINT at certain matrix and bit sizes.

The evaluation table (Table 1) presents the execution times and speedup achieved by the vectorized implementations compared to the original versions across different input sizes using 12 bit length prime.
The vectorized versions of both the Basic and Crout PLUQ algorithms consistently exhibit lower execution times compared to their Original counterparts across all matrix sizes. Additionally, the speedup values for the vectorized versions indicate significant improvements in performance over the original implementations. Therefore, based on the presented data, it can be inferred that the

Sizes	100	300	500	1000	1200
Original Basic PLUQ (ms)	0.69	18.70	86.57	694.06	1199.92
Original Crout PLUQ (ms)	0.71	19.00	87.40	697.97	1204.63
Vectorized Basic PLUQ (ms)	0.15	3.90	17.80	145.03	249.08
Vectorized Crout PLUQ (ms)	0.11	2.37	12.97	135.07	246.83
Speedup of Basic PLUQ	4.60	4.79	4.86	4.95	4.95
Speedup of Crout PLUQ	6.45	8.01	6.73	5.16	4.86

Table 1: Basic and Crout PLUQ Speedups using AVX2 and 12 Bits Length Prime

vectorized versions of both the Basic and Crout PLUQ algorithms are better in terms of computational efficiency and speed compared to their original counterparts.

For FFLAS-FFPACK [1], the library accepts primes with a maximum length of 15 bits in its 32-bit integer representation. The table (Table 2) below illustrates the execution times of the library implementation and the corresponding achieved speedup using 12 bits length prime.

Sizes	100	300	500	1000	1200
FFLAS-FFPACK (ms)	0.34	4.43	15.09	62.32	88.28
Speedup of Vectorized Basic PLUQ (ms)	2.26	1.13	0.85	0.42	0.35
Speedup of Vectorized Crout PLUQ (ms)	3.09	2.37	1.16	0.46	0.36

Table 2: Basic and Crout PLUQ Speedups Against FFLAS-FFPACK using AVX2 and 12 Bits Length Prime

For smaller matrices (100 to 300), both implementations demonstrate notable speedups compared to FFLAS-FFPACK. However, as matrix size increases, the speedup diminishes, particularly evident for the largest matrix size of 1200. This library utilizes a more efficient routine for larger sizes, leveraging recursive matrix multiplication.

6 Conclusion

In conclusion, our efforts to create a faster PLUQ implementation than the FLINT library for medium-sized matrices have paid off, as shown by the data. However, improving performance doesn't mean we're done optimizing. We could try things like using OpenMP for multiple threads or using wider SIMD instructions like AVX-512 to make better use of parallel processing and memory. These ideas could boost the algorithm's performance even more, helping us keep pushing for better efficiency in numerical and linear algebra calculations. Alongside these technical optimizations, considerations of software design principles such as modularity, abstraction, and algorithmic complexity reduction can contribute to overall system efficiency.

References

[1] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. Dense linear algebra over word-size prime fields: the fflas and ffpack packages. *ACM Transactions on Mathematical Software*

(TOMS), 35(3):1–42, 2008.

[2] Jean-Guillaume Dumas, Clément Pernet, and Ziad Sultan. Fast computation of the rank profile matrix and the generalized bruhat decomposition. *Journal of Symbolic Computation*, 83:187–210, 2017.

[3] Pierre Fortin, Ambroise Fleury, François Lemaire, and Michael Monagan. High-performance simd modular arithmetic for polynomial evaluation. *Concurrency and Computation: Practice and Experience*, 33(16):e6270, 2021.

[4] William B Hart. Flint: Fast library for number theory. *Computeralgebra Rundbrief*, 2013.

[5] Joris Van Der Hoeven, Grégoire Lecerf, and Guillaume Quintin. Modular simd arithmetic in mathemagix. *ACM Transactions on Mathematical Software (TOMS)*, 43(1):1–37, 2016.