HIGH-PERFORMANCE IMPLEMENTATION OF GAUSSIAN ELIMINATION
Progress Report

Mohamed Imad Eddine Ghodbane

March 2024

1. Introduction

Modular arithmetic serves as a cornerstone in various fields such as cryptography and computer algebra. Within these domains, Gaussian elimination in finite fields emerges as a critical algorithm, essential for solving systems of linear equations and performing matrix operations over finite fields. In cryptography, Gaussian elimination is fundamental for tasks such as error-correction coding, where it forms the backbone of decoding algorithms for codes defined over finite fields. Algebra libraries, as indispensable tools in computational mathematics, aim to provide efficient implementations of such algorithms to optimize computational processes. By focusing on the fast implementation of Gaussian elimination within finite fields, this report aims to address the specific computational challenges inherent in these contexts. Through advancements in implementation techniques tailored to finite fields, including the utilization of SIMD (AVX2) instructions, the goal is to significantly enhance computational efficiency of Gaussian elimination in finite field calculations.

2. 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 multiplication 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, thus preventing potential overflows. The choice of $n = 32$ bits and $n = 30$ is motivated by the need to strike a balance between computational efficiency and the range of representable integers. 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.

2.1. Modular sum

Given two integers a and b in 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 $(a + b) - p \leq p - 2$.

2.1.1. Unvectorized version

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;
}
```

2.1.2. Vectorized version

To optimize the modular sum operation from a scalar implementation to a vectorized form using AVX2 instructions, we transitioned from a straightforward scalar approach to a highly efficient 128-bit vectorized implementation. 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 additions at one time. In this case, 4 32-bit integers in one packed vector.

Function 2

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

```
add_mod_vec(va, vb, vp) {
    1. res = _mm_add_epi32(va, vb);
    2. mask = _mm_cmpgt_epi32(res, _mm_set1_epi32(p - 1));
    3. adjusted_res = _mm_sub_epi32(res, vp);
    4. res = _mm_blendv_epi8(res, adjusted_res, mask);
    5. return res;
}
```

This implementation takes two 128-bit vectors va and vb modulo p as input and outputs the element-wise sum of these vectors modulo p. Inside the function, the vectors are added element-wise using the `_mm_add_epi32` function, then compared with $(p-1)$ using `_mm_cmpgt_epi32` to generate a mask. This mask is used to conditionally blend the result of the addition with the difference between the result and p, achieved through subtraction with vp. The final result is returned as a 128-bit vector.

2.2. Modular subtraction

Similarly to the modular addition, given two integers a and b in 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 $\{0, 1, \ldots, p-1\}$.

2.2.1. Unvectorized version

In the scalar implementation, we avoid using the division overhead of the Euclidean algorithm and instead employ an implementation similar to that of addition.

Function 3

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 < 0) ? res + p : res;
}
```

2.2.2. Vectorized version

We observe that there is no difference between addition and subtraction in both versions. The reason for presenting the subtraction is that when we start implementing Gaussian elimination, we will not the addition operations, but we will use subtraction, multiplication, and inversion in F_p.

Function 4
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;
}
```

2.3. Modular multiplication

In our implementation, 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.

2.3.1. Uvectorized version

In this version `res` is a 64-bit integer to hold the product, and `mult_mod` returns 32-bit integer.

Function 5
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;
}
```

In our current implementation, the modular multiplication operation 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 [1], 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 $c = \left\lfloor \frac{h}{p} \right\rfloor$, 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 6

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.

2.3.2. Vectorized version

To vectorize function 6, we utilized 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, maintaining the integrity of the modular multiplication operation.

Function 7

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) {
    1. h = _mm256_mul_pd(x, y);
    2. l = _mm256_fmsub_pd(x, y, h);
    3. b = _mm256_mul_pd(h, u);
    4. c = _mm256_floor_pd(b);
    5. d = _mm256_fnmadd_pd(c, p, h);
    6. e = _mm256_add_pd(d, l);
    7. t = _mm256_sub_pd(e, p);
    8. e = _mm256_blendv_pd(t, e, t);
    9. t = _mm256_add_pd(e, p);
   10. return _mm256_blendv_pd(e, t, e);
}
```

2.4. 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.

3. Gaussian Elimination Implementation

In this section, we delve into the Gaussian elimination algorithm, a fundamental method used in various mathematical and computational contexts. Specifically, we explore its application in the PLUQ implementation, which incorporates Gaussian elimination as a key component.

3.1 PLUQ implimentation

The PLUQ algorithm extends the LU decomposition method, used to factorize a square matrix into lower and upper triangular matrices (L and U) and

permutation matrices (P and Q). The algorithm performs row and column permutations, and it uses Gaussian elimination to decompose the matrix into lower and upper triangular matrices.

Function 8

Input: Matrix A, with dimensions $m \times n$.

Output: LU, P, Q, rank, such that A = PLUQ.

```
pluq(A){
    rank = 0;
    nullity = 0;
    while (rank + nullity < m){
        pivot = rank;
        while (pivot < n and LU[rank,pivot] == 0)
            pivot += 1;
        if (pivot == n){
            P = row_rotation(LU,rank,P);
            nullity += 1;
        }else{
            Q = column_transposition(LU,rank,pivot,Q)
            inv = inevrse(LU[rank, rank], p);
            for (k = rank + 1; k < m; k++){
                LU[k,rank] = mul_mod(LU[k,rank], inv, p);
                for (j = rank + 1; j < n; j++){
                    tmp = mul_mod(LU[k,rank], LU[rank,j]);
                    LU[k,j] = sub_mod(LU[k,j], tmp);
                }
            }
            rank += 1
        }
    return LU,P,Q,rank;
    }
}
```

LU is set to A, P is the row permutation matrix, and Q is the column permutation matrix. The algorithm iterates through the rows of LU, identifying pivot elements and performing operations accordingly to achieve LU decomposition. If a pivot element is zero, indicating a potential row permutation, the row rotation function is invoked, updating P and incrementing nullity. Otherwise, if the pivot element is nonzero, column transposition is performed to update Q, followed by Gaussian elimination steps to modify LU accordingly.
Utilizing an array to store the matrix A with row-major storage, we can enhance computational efficiency by vectorizing this loop:

```
for (j = rank + 1; j < n; j++){
    tmp = mul_mod(LU[k,rank], LU[rank,j]);
    LU[k,j] = sub_mod(LU[k,j], tmp);
```

```
}
```

This process involves transforming the loop using vectorized modular operations, By replacing the loop with vectorized instructions, multiple iterations can be executed simultaneously, thereby accelerating the computation. Unrolling the loop further optimizes performance by reducing loop overhead and enhancing instruction-level parallelism.

3.2 Vectorized implementation

3.2.1 Vectorized row elimination

Initially, we set up vectors containing the inverse of the pivot, modulo p, and the reciprocal of p to optimize calculations. then we iterate through the rows starting from the $rank+1$ row, processing four columns at a time using SIMD instructions. Within this loop, it loads chunks of data from matrix A and converts them into vectors for efficient computation. It performs modular multiplication between the loaded data and the inverse of the pivot, storing the result back into the matrix after conversion to integer format. For the remaining columns not processed in groups of four, it computes the multiplication and subtraction individually, adhering to the modular arithmetic.

Function 9

Input: Matrix A, inverse of pivot, p, rank, and current column k.
Output: Matrix A after one iteration of elimination

```
rows_elimination_vec(A, inv_pivot, p, rank, k) {
    vc = _mm256_set1_pd(inv_pivot);
    vp = _mm256_set1_pd(p);
    vu = _mm256_set1_pd(1.0 / p);
    tmp;
    for (i = rank + 1; i + 3 < n; i += 4) {
        v1 = _mm_loadu_si128(&A[rank, i]);
        v2 = _mm_loadu_si128(&A[k, i]);
        vDouble = _mm256_cvtepi32_pd(v1);
        tmp = mul_mod(vc, vDouble, vu, vp);
        resultInt = _mm256_cvttpd_epi32(tmp);
        result = sub_vec(v2, resultInt, p);
        _mm_storeu_si128(&A[k, i], result);
    }

     for (; i < n; ++i){
        tmp = mult(A[k, rank], A[rank, i], p);
        A[k, i] = sub(A[k, i], tmp , p);
     }
}
```

References

[1] 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.