Gaussian Elimination High-Performance Implementation

Mohamed Imad Eddine Ghodbane

May, 2024

SCIENCES SORBONNE UNIVERSITÉ

CONTENT

PROJECT OVERVEW

- Gaussien elimination over a finite field Fp.
- p prime, less than 30 bits in length.
- Libraries include FFLAS-FFPACK, Flint known for high performance.
- Room for improvement for matrices of intermediate dimensions.
- AVX2 vectorization for enhanced performance.
- Aim for superior performance compared to existing libraries.

PLUQ FACTORIZATION

Input:

- Matrix A of size m×n with entries in the field Fp.

Output:

- LU decomposition of A, with permutation matrices P and Q.

Operations:

- Pivoting: Involves row and column rotations.
- Row Reduction: Process of reducing rows to obtain the LU decomposition.

PLUQ FACTORIZATION

- **Input** $m = n = 3$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

PLUQ FACTORIZATION

- Pivoting

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$ = [0, 0, 0]

PLUQ FACTORIZATION

- Pivoting

$$\begin{bmatrix} a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ \color{red}{a_{11}} & \color{red}{a_{12}} & \color{red}{a_{13}} \end{bmatrix}$$

PLUQ FACTORIZATION

- Pivoting

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$ = [0, *, *]

PLUQ FACTORIZATION

- Pivoting

$$\begin{bmatrix} a_{21} & \color{red}{a_{11}} & a_{13} \\ a_{22} & \color{red}{a_{21}} & a_{23} \\ a_{32} & \color{red}{a_{31}} & a_{33} \end{bmatrix}$$

PLUQ FACTORIZATION

- **Row Reduction**

$$a_{11}^{-1} \cdot \begin{pmatrix} a_{21} \\ a_{31} \end{pmatrix} = \begin{bmatrix} a_{11}^{-1} & a_{12} & a_{13} \\ l_{21} & a_{22} & a_{23} \\ l_{31} & a_{32} & a_{33} \end{bmatrix}$$

PLUQ FACTORIZATION

- **Row Reduction**

$$\begin{bmatrix} {\color{red}a_{11}^{-1}} & a_{12} & a_{13} \\ l_{21} & \boxed{a'_{22} \quad a'_{23}} \\ l_{31} & \boxed{a'_{32} \quad a'_{33}} \end{bmatrix} \begin{matrix} \\ = \begin{pmatrix} a_{22} \\ a_{23} \end{pmatrix} - l_{21} \cdot \begin{pmatrix} a_{12} \\ a_{13} \end{pmatrix} \\ = \begin{pmatrix} a_{32} \\ a_{33} \end{pmatrix} - l_{31} \cdot \begin{pmatrix} a_{12} \\ a_{13} \end{pmatrix} \end{matrix}$$

PLUQ FACTORIZATION

- Row Reduction

$$\begin{bmatrix} a_{11}^{-1} & a_{12} & a_{13} \\ l_{21} & a'_{22} & a'_{23} \\ l_{31} & a'_{32} & a'_{33} \end{bmatrix}$$

PLUQ FACTORIZATION

- **Output**

$$P \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & u_{33} \end{bmatrix} \cdot Q$$

PLUQ IMPLEMENTATION

```
/*
 Pivoting
*/

// Row Reduction
for (int k = matrixRank + 1; k < m; k++) {
    A->data[k * n + matrixRank] = mult(
    A->data[k * n + matrixRank], inv, p);
    for (int j = matrixRank + 1; j < n; j++)
        A->data[k * n + j] = sub(
        A->data[k * n + j],
        mult(A->data[k * n + matrixRank],
        A->data[matrixRank * n + j], p), p);
}
```

MODULAR ARITHMETIC USING SIMD

- Subtraction

```
int sub(int a, int b, int p) {
    int r = a - b;
    return r < 0 ?  r + p : r;
}
```

MODULAR ARITHMETIC USING SIMD

- Subtraction `_mm_sub_epi32`

a_1	a_2	a_3	a_4

-

b_1	b_2	b_3	b_4

=

$a_1 - b_1$	$a_2 - b_2$	$a_3 - b_3$	$a_4 - b_4$

MODULAR ARITHMETIC USING SIMD

- Subtraction `_mm_cmplt_epi32`

$a_1 - b_1$	$a_2 - b_2$	$a_3 - b_3$	$a_4 - b_4$

<?

0	0	0	0

=

1	0	1	0

MODULAR ARITHMETIC USING SIMD

- **Subtraction** `_mm_add_epi32`

$a_1 - b_1$	$a_2 - b_2$	$a_3 - b_3$	$a_4 - b_4$

$$+$$

p	p	p	p

$$=$$

$ar1$	$ar2$	$ar3$	$ar4$

MODULAR ARITHMETIC USING SIMD

- Subtraction `_mm_blendv_epi8`

$a_1 - b_1$	$a_2 - b_2$	$a_3 - b_3$	$a_4 - b_4$

?

$ar1$	$ar2$	$ar3$	$ar4$

=

$ar1$	$a_2 - b_2$	$ar3$	$a_4 - b_4$

MODULAR ARITHMETIC USING SIMD

- Subtraction

```
__m128i sub_avx2(__m128i a, __m128i b, __m128i vp) {
    __m128i result = _mm_sub_epi32(a, b);
    __m128i mask = _mm_cmplt_epi32(result, _mm_setzero_si128());
    __m128i adjusted_result = _mm_add_epi32(result, vp);
    result = _mm_blendv_epi8(result, adjusted_result, mask);
    return result;
}
```

MODULAR ARITHMETIC USING SIMD

- Multiplication

```
int mult(int a, int b, int p) {
    long long r = (long long)a * b;
    return r % p;
}
```

MODULAR ARITHMETIC USING SIMD

- **Multiplication** `_mm256_mul_pd`

$$h = ab = cp + e$$

$$\begin{array}{|c|c|c|c|}\hline a_1 & a_2 & a_3 & a_4 \\ \hline \end{array}$$

$$*$$

$$\begin{array}{|c|c|c|c|}\hline b_1 & b_2 & b_3 & b_4 \\ \hline \end{array}$$

$$=$$

$$\begin{array}{|c|c|c|c|}\hline h_1 & h_2 & h_3 & h_4 \\ \hline \end{array}$$

MODULAR ARITHMETIC USING SIMD

- **Multiplication** `_mm256_mul_pd`

$$h = ab = cp + e$$

$$u = \frac{1}{p}, d = hu$$

h_1	h_2	h_3	h_4

$*$

u_1	u_2	u_3	u_4

$=$

d_1	d_2	d_3	d_4

MODULAR ARITHMETIC USING SIMD

- **Multiplication** `_mm256_floor_pd`

$$h = ab = cp + e$$

$$u = \frac{1}{p}, d = hu$$

$$c = \lfloor d \rfloor$$

$$\begin{array}{|c|c|c|c|}\hline \lfloor d_1 \rfloor & \lfloor d_2 \rfloor & \lfloor d_3 \rfloor & \lfloor d_4 \rfloor \\ \hline \end{array}$$

=

$$\begin{array}{|c|c|c|c|}\hline c_1 & c_2 & c_3 & c_4 \\ \hline \end{array}$$

MODULAR ARITHMETIC USING SIMD

- **Multiplication** `_mm256_fnmadd_pd`

$$h = ab = cp + e$$

$$u = \frac{1}{p}, d = hu$$

$$c = \lfloor d \rfloor$$

$$e = h - cp$$

h_1	h_2	h_3	h_4

-

c_1p	c_2p	c_3p	c_4p

=

e_1	e_2	e_3	e_4

MODULAR ARITHMETIC USING SIMD

- Multiplication

```
__m256d mul_mod_p(__m256d a, __m256d b, __m256d u, __m256d p) {
    __m256d h = _mm256_mul_pd(a, b);
    __m256d d = _mm256_mul_pd(h, u);
    __m256d c = _mm256_floor_pd(d);
    __m256d e = _mm256_fnmadd_pd(c, p, h);
    return e;
}
```

The resulting product often exceeds the 52-bit length allocated for the mantissa in double floating-point representation. This can lead to a loss of precision in the final result !!

MODULAR ARITHMETIC USING SIMD

- Multiplication

```
__m256d mul_mod_p(__m256d a, __m256d b, __m256d u, __m256d p) {
    __m256d h = _mm256_mul_pd(a, b);
    __m256d l = _mm256_fmsub_pd(x, y, h);
    __m256d d = _mm256_mul_pd(h, u);
    __m256d c = _mm256_floor_pd(d);
    __m256d b = _mm256_fnmadd_pd(c, p, h);
    __m256d e = _mm256_add_pd(b, l);
    __m256d 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);
}
```

SIMD IMPLEMENTATION OF PLUQ

```
rows_elimination_avx2(int *A_data,int n, int matrixRank, int c,int p ,
__m256d vp, __m256d vu ,__m128i vp_128, int k) {
    __m256d vc = _mm256_set1_pd(c);
    __m256d tmp;
    int i;
    for (i = matrixRank + 1; i + 3 < n; i += 4) {
        __m128i v1 = _mm_loadu_si128((__m128i *)&A_data[matrixRank*n+i]);
        __m128i v2 = _mm_loadu_si128((__m128i *)&A_data[k*n+i]);
        __m256d vDouble = _mm256_cvtepi32_pd(v1);
        tmp = mul_mod_p(vc, vDouble, vu, vp);
        __m128i resultInt = _mm256_cvttpd_epi32(tmp);
        __m128i result = sub_avx2(v2, resultInt, vp_128);
        _mm_storeu_si128((__m128i *)&A_data[k * n + i], result);
    } // loop handles elements that don't fit into chunks of 4
}
```

RESULTS

- AVX2 vs Basic PLUQ

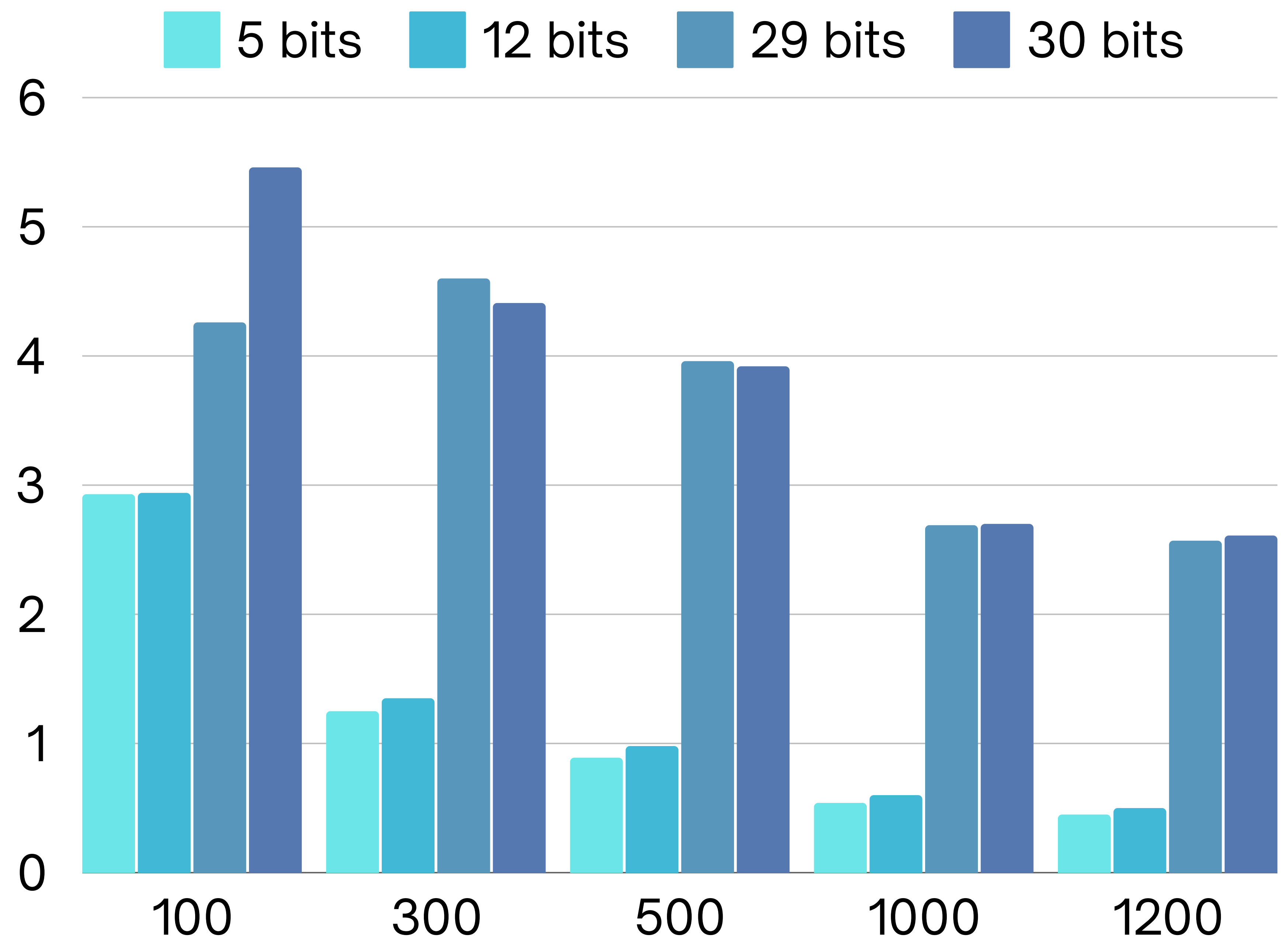
Sizes	100	300	500	1000	1200
Basic (ms)	0.69	18.70	86.57	694.06	1199.92
AVX2 (ms)	0.15	3.90	17.80	139.98	242.56
Speedup	4.60	4.79	4.86	4.95	4.95

PLUQ Speedups using AVX2 and 12 Bits Length Prime

AMD Ryzen™ 7 PRO 7840U w/ Radeon™ 780M Graphics × 16

RESULTS

- **AVX2 vs FLINT**



AMD Ryzen™ 7 PRO 7840U w/ Radeon™ 780M Graphics × 16