Gaussian Elimination
High-Performance Implementation

Mohamed Imad Eddine Ghodbane

May, 2024

SCIENCES SORBONNE UNIVERSITÉ

CONTENT

01 PROJECT OVERVIEW

02 PLUQ FACTORIZATION

03 MODULAR ARITHMETIC USING SIMD

04 SIMD IMPLEMENTATION OF PLUQ

05 CROUT METHOD

06 SIMD SCALAR PRODUCT

07 SIMD IMPLEMENTATION OF CROUT METHOD

08 RESULTS

PROJECT OVERVIEW

Primary Objective

- Develop a high-performance implementation of Gaussian elimination.
- Focus on exact linear algebra over finite fields ($Fp=Z/pZ$) with a prime number stored on 30 bits.

Existing Libraries

- FFLAS-FFPACK, Flint, NTL.
- High performance for large matrices and small prime fields.

Areas for Improvement

- Intermediate matrix dimensions (hundreds to thousands of elements).
- Larger prime numbers.

PLUQ FACTORIZATION

$$A = PLUQ$$

Input

- **Matrix A of size $m \times n$ with entries elements in the finite field Fp.**

Output

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

Key Operations

- **Swap rows and columns based on pivot (pivot = 0).**
- **Zero out elements below the pivot (Row Reduction).**

PLUQ FACTORIZATION

Column Transposition

- Let A be a 3×3 matrix:

$$A = \begin{pmatrix} \color{red}{a_{00}} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

pivot = 0

- Choose the next first non-zero entry in the row.
- Perform column transposition to move the non-zero entry to the pivot position.

PLUQ FACTORIZATION

Column Transposition

$$A = \begin{pmatrix} \color{red}{a_{01}} & 0 & a_{02} \\ a_{11} & a_{01} & a_{12} \\ a_{21} & a_{01} & a_{22} \end{pmatrix} \qquad Q = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

pivot = 0

- Choose the next first non-zero entry in the row.
- Perform column transposition to move the non-zero entry to the pivot position.

PLUQ FACTORIZATION

Row Rotation

$$A = \begin{pmatrix} 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

pivot row = [0, 0, 0]

- **Perform row rotation to move the zero row to the last row position.**

PLUQ FACTORIZATION

Row Rotation

$$A = \begin{pmatrix} \color{red}{a_{10}} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ 0 & 0 & 0 \end{pmatrix} \qquad P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

pivot row = [0, 0, 0]

- **Perform row rotation to move the zero row to the last row position.**

PLUQ FACTORIZATION

Row Reduction

$$A = \begin{pmatrix} \color{red}{a_{00}^{-1}} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

- **Compute the inverse of the pivot modulo p.**
- **Multiply the pivot inverse by the elements of the pivot row.**
- **Use the updated pivot row to eliminate the elements below the pivot in the current column.**

PLUQ FACTORIZATION

Row Reduction

$$\begin{pmatrix} l_{10} \\ l_{20} \end{pmatrix} = a_{00}^{-1} \cdot \begin{pmatrix} a_{10} \\ a_{20} \end{pmatrix}$$

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ \color{red}{l_{10}} & a_{11} & a_{12} \\ \color{red}{l_{20}} & a_{21} & a_{22} \end{pmatrix}$$

- **Compute the inverse of the pivot modulo p.**
- **Multiply the pivot inverse by the elements of the pivot row.**
- **Use the updated pivot row to eliminate the elements below the pivot in the current column.**

PLUQ FACTORIZATION

Row Reduction

$$\begin{pmatrix} a'_{11} \\ a'_{12} \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{12} \end{pmatrix} - l_{10} \cdot \begin{pmatrix} a_{01} \\ a_{02} \end{pmatrix}$$

$$\begin{pmatrix} a'_{21} \\ a'_{22} \end{pmatrix} = \begin{pmatrix} a_{21} \\ a_{22} \end{pmatrix} - l_{20} \cdot \begin{pmatrix} a_{01} \\ a_{02} \end{pmatrix}$$

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ l_{10} & \color{red}{a'_{11}} & \color{red}{a'_{12}} \\ l_{20} & \color{red}{a'_{21}} & \color{red}{a'_{22}} \end{pmatrix}$$

- **Compute the inverse of the pivot modulo p.**
- **Multiply the pivot inverse by the elements of the pivot row.**
- **Use the updated pivot row to eliminate the elements below the pivot in the current column.**

PLUQ FACTORIZATION

Output

$$P = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \qquad A = \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ l_{10} & u_{11} & u_{12} \\ l_{20} & l_{21} & u_{22} \end{pmatrix} \qquad Q = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix}$$

PLUQ FACTORIZATION

One Iteration

```
/*
 Find Pivot
*/

// 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;
}
```

Multiplication

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

MODULAR ARITHMETIC USING SIMD

Subtraction

```
__m128i res = _mm_sub_epi32(a, b);
__m128i mask = _mm_cmplt_epi32(res,
                 _mm_setzero_si128());
__m128i ar = _mm_add_epi32(res, p);
res = _mm_blendv_epi8(res, ar, mask);
```

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

```
__m128i res = _mm_sub_epi32(a, b);
__m128i mask = _mm_cmplt_epi32(res,
                  _mm_setzero_si128());
__m128i ar = _mm_add_epi32(res, p);
res = _mm_blendv_epi8(res, ar, mask);
```

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

<?

0	0	0	0

=

1	0	1	0

MODULAR ARITHMETIC USING SIMD

Subtraction

```
__m128i res = _mm_sub_epi32(a, b);
__m128i mask = _mm_cmplt_epi32(res,
                  _mm_setzero_si128());
__m128i ar = _mm_add_epi32(res, p);
res = _mm_blendv_epi8(res, ar, mask);
```

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

$+$

p	p	p	p

$=$

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

MODULAR ARITHMETIC USING SIMD

Subtraction

```
__m128i res = _mm_sub_epi32(a, b);
__m128i mask = _mm_cmplt_epi32(res,
                 _mm_setzero_si128());
__m128i ar = _mm_add_epi32(res, p);
res = _mm_blendv_epi8(res, ar, mask);
```

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

?

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

=

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

MODULAR ARITHMETIC USING SIMD

Multiplication

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

There is no modulus operation in AVX2 !!

Solution: compute the remainder of the division by p.

$$h = ab = cp + e$$

$$d = \frac{h}{p}$$

$$c = \lfloor d \rfloor$$

$$e = h - cp$$

MODULAR ARITHMETIC USING SIMD

Multiplication

```
__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);
```

a_1	a_2	a_3	a_4

$$*$$

b_1	b_2	b_3	b_4

$$=$$

h_1	h_2	h_3	h_4

MODULAR ARITHMETIC USING SIMD

Multiplication

```
__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);
```

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

$$*$$

$$\begin{array}{|c|c|c|c|}\hline u_1 & u_2 & u_3 & u_4 \\ \hline \end{array}$$

$$=$$

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

MODULAR ARITHMETIC USING SIMD

Multiplication

```
__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);
```

$\lfloor d_1 \rfloor$	$\lfloor d_2 \rfloor$	$\lfloor d_3 \rfloor$	$\lfloor d_4 \rfloor$

=

c_1	c_2	c_3	c_4

MODULAR ARITHMETIC USING SIMD

Multiplication

```
__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);
```

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 Product 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

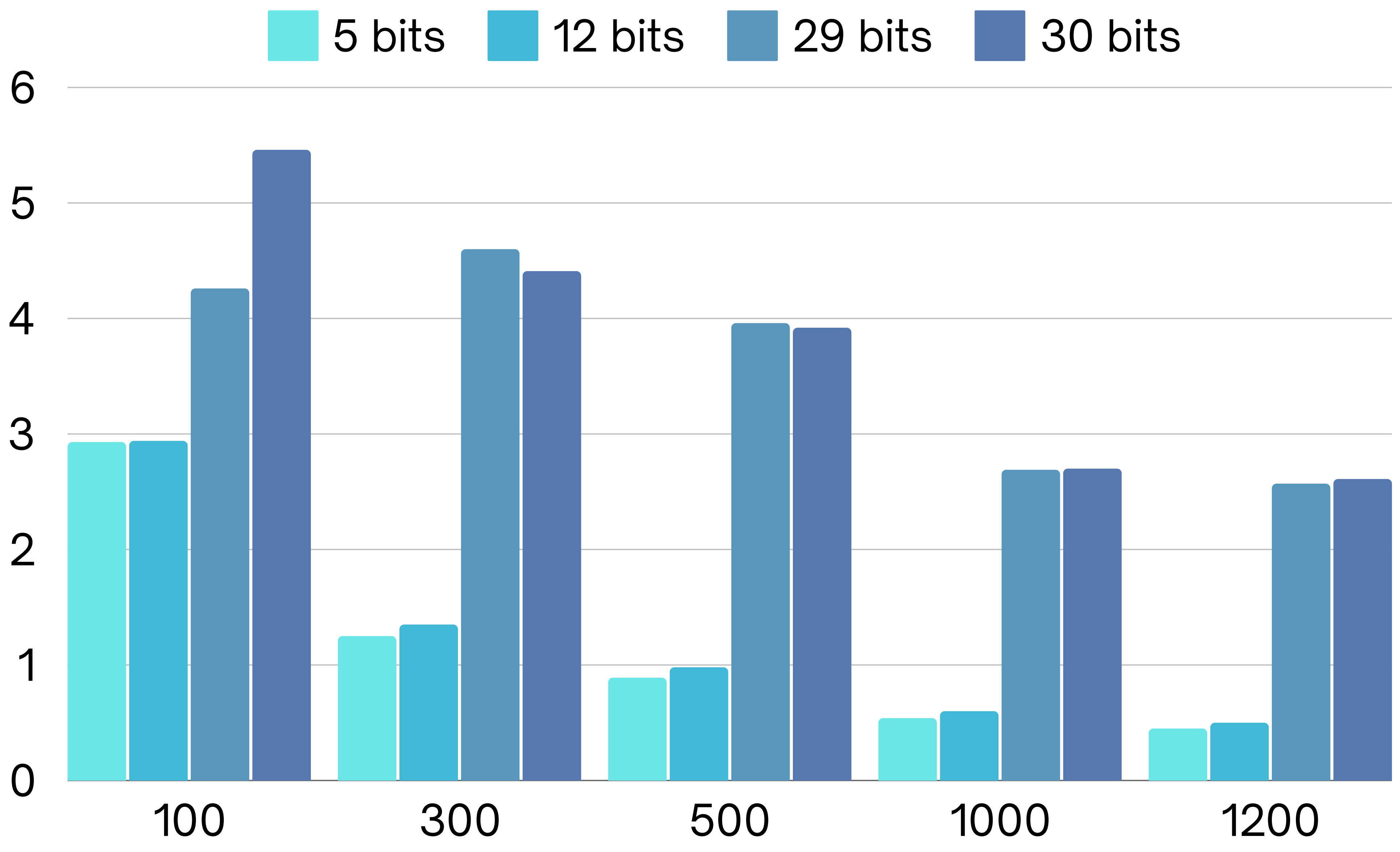
Sizes	100	300	500	1000	1200
Basic (ms)	0.69	18.70	86.57	694.06	1199.92
AVX2 (ms)	0.15	3.90	17.80	139.98	242.56
Speedup	4.60	4.79	4.86	4.95	4.95

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

CROUT METHOD

Key Operation

- **Computes uij and lij using the multiplication equation A=LU and the previously computed values of L and U.**

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} = P \cdot \begin{pmatrix} 1 & 0 & 0 \\ l_{10} & 1 & 0 \\ l_{20} & l_{20} & 1 \end{pmatrix} \cdot \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ 0 & u_{11} & u_{12} \\ 0 & 0 & u_{22} \end{pmatrix} \cdot Q$$

$$\begin{pmatrix} l_{10} \\ l_{20} \end{pmatrix} = \begin{pmatrix} a_{10} \times u_{00}^{-1} \\ a_{20} \times u_{00}^{-1} \end{pmatrix}$$

$$A = \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ l_{10} & a_{11} & a_{12} \\ l_{20} & a_{21} & a_{22} \end{pmatrix}$$

CROUT METHOD

$$A = \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ l_{10} & a_{11} & a_{12} \\ l_{20} & a_{21} & a_{22} \end{pmatrix} \quad A = \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ l_{10} & u_{11} & u_{12} \\ l_{20} & a_{21} & a_{22} \end{pmatrix} \quad A = \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ l_{10} & u_{11} & u_{12} \\ l_{20} & l_{21} & a_{22} \end{pmatrix}$$

$$A = \begin{pmatrix} u_{00} & u_{01} & u_{02} \\ l_{10} & u_{11} & u_{12} \\ l_{20} & l_{21} & u_{22} \end{pmatrix}$$

$$u_{ij} = a_{ij} - \sum_{k=0}^{j-1} l_{ij} \cdot u_{ji} \qquad l_{ij} = (a_{ij} - \sum_{k=0}^{i-1} l_{ik} \cdot u_{kj}) \cdot u_{jj}^{-1}$$

SIMD SCALAR PRODUCT

Scalar Product

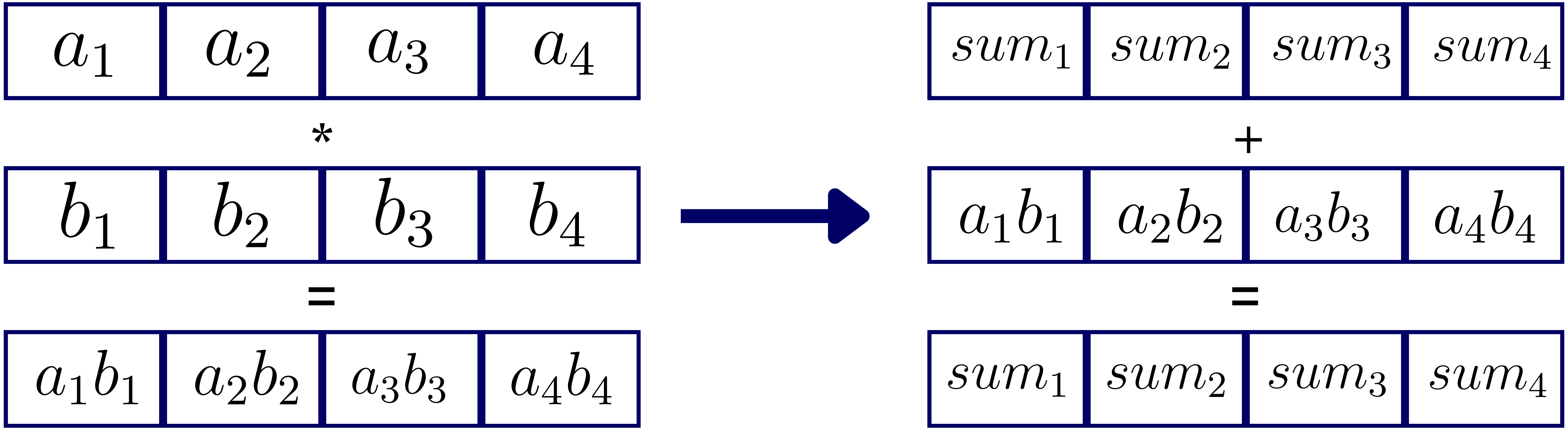
```
int sum = 0;
for (int i = 0; i < length; i++) {
    sum += a[i] * b[i];
}
return sum % p;
```

Independence of Iterations

- Each iteration of the loop computes a[i] * b[i] and adds it to sum.
- The computation of a[i] * b[i] is independent of the previous iterations.

SIMD SCALAR PRODUCT

Scalar Product



$$result = (sum_1 + sum_2 + sum_3 + sum_4) \ \ mod \ \ p$$

RESULTS

AVX2 vs Basic Crout PLUQ

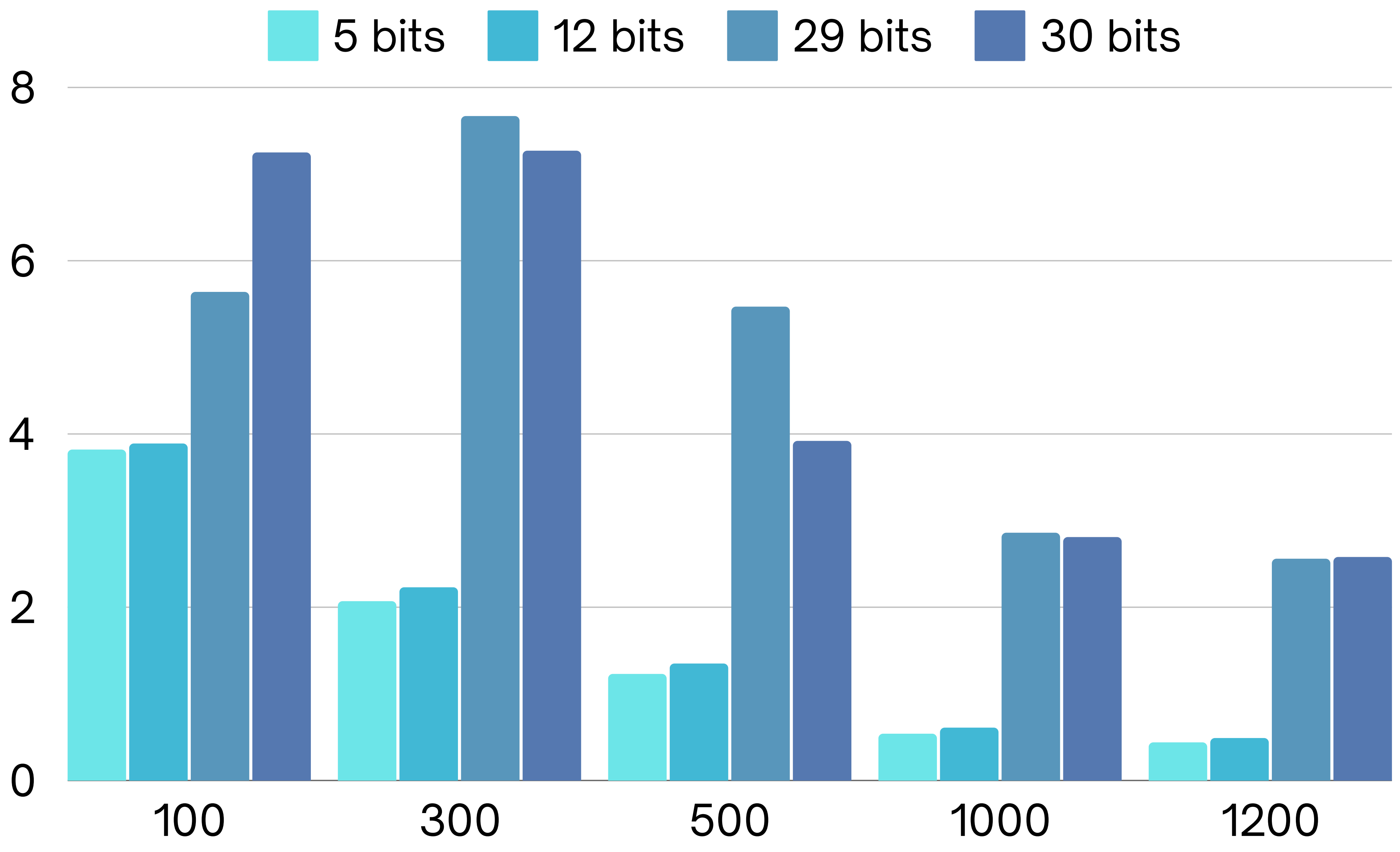
Sizes	100	300	500	1000	1200
Basic (ms)	0.71	19.00	87.40	697.97	1204.63
AVX2 (ms)	0.11	2.37	12.97	135.07	246.83
Speedup	6.45	8.01	6.73	5.16	4.86

Crout 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

Thank You

Questions?