به نام خدا

تقریب خطی جدول جانشینی الگوریتم SNOW2.0

علی شیخ عطار
دکتر وحید امین غفاری

دانشکده مهندسی کامپیوتر علم و صنعت
۱۴۰۳

این document توضیحی جامع از کد ارائه‌شده و بهینه‌سازی‌های عملکردی آن است. هدف این کد محاسبه **بایاس تقریب خطی S-box در الگوریتم SNOW 2.0** است که شامل S-box الگوریتم AES و تبدیل MixColumns می‌شود.

تعریف کلی کد

در کد مراحل زیر را انجام می‌شود:

- **پیش‌محاسبه جداول تبدیل** برای بهینه‌سازی عملکرد MixColumns.
- **پردازش ماسک ورودی ۳۲ بیتی** به ازای ورودی کاربر.
- **محاسبه برابری پاریتی (Parity) ورودی‌ها و خروجی‌های ماسک‌شده**.
- **شمارش تعداد تطابق‌ها** که در آن پاریتی ورودی و خروجی یکسان است.
- **محاسبه بایاس** بر اساس این شمارش.
- **نمایش بایاس و زمان اجرا** برای هر پردازش.

توضیح دقیق کد

۱. وارد کردن هدرهای ضروری

```
#include <iostream>
#include <cstdint>
#include <vector>
#include <array>
#include <chrono>
#include <iomanip>
#include <cmath>
#include <omp.h>
#include <immintrin.h>
```

- وارد کردن هدرهای استاندارد و خاص سیستم که در برنامه استفاده می‌شوند.
- `<iostream>`: برای ورودی/خروجی.
- `<cstdint>`: برای استفاده از انواع داده‌ای با عرض ثابت، مثل `uint8_t` و `uint64_t`.
- `<vector>` و `<array>`: برای استفاده از کلاس‌های کانتینر.
- `<chrono>`: برای اندازه‌گیری زمان اجرا.
- `<iomanip>`: برای قالب‌بندی خروجی.
- `<cmath>`: برای توابع ریاضیاتی.
- `<omp.h>`: برای پردازش موازی با استفاده از OpenMP.
- `<immintrin.h>`: برای استفاده از دستورات SIMD (البته در این کد به طور مستقیم استفاده نشده است).

تعریف S-box الگوریتم AES

```
alignas(16) const uint8_t AES_SBOX[256] = {

    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
    0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
    0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
    0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
    0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
    0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
    0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
    0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
    0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
    0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
    0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
    0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
    0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
    0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
    0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};
```

- جعبه S در AES یک جعبه جایگشتی غیرخطی است که یک ورودی ۸ بیتی را به یک خروجی ۸ بیتی نگاشت می‌دهد.
- در SNOW 2.0، جعبه S از AES به عنوان بخش غیرخطی استفاده می‌شود.

`alignas(16)` چیست؟

- این دستور آرایه AES_SBOX را روی مرزهای حافظه ۱۶ بایتی تنظیم می‌کند و تضمین می‌کند که آدرس شروع آرایه مضربی از ۱۶ بایت باشد.
- این کار می‌تواند در سیستم‌هایی که از عملیات SIMD استفاده می‌کنند، عملکرد را بهبود بخشد.
- دسترسی به داده‌های تراز شده در برخی معماری‌ها سریع‌تر است.

تعریف ثابت‌ها

```
const uint64_t ms_n = 1ULL << 32;
```

- ms_n تعداد کل ورودی‌های ممکن ۳۲ بیتی (که برابر با 2^{32} است) را نشان می‌دهد.
- از آنجا که جعبه S روی ورودی‌های ۳۲ بیتی عمل می‌کند، باید تمام این ترکیب‌ها را پردازش کرد.

چگونه این کار انجام می‌شود:

- از عملگر شیفت چپ (<<) استفاده شده تا مقدار 1ULL << 32 محاسبه شود. این مقدار عدد 1 را ۳۲ بیت به سمت چپ شیفت می‌دهد.
- این روش از تایپ اشتباه اعداد بزرگ جلوگیری می‌کند.

ضرب در میدان گالوا (`gf_mul`)

```
inline uint8_t gf_mul(uint8_t a, uint8_t b) {
    uint8_t p = 0;
    for (int i = 0; i < 8 && b; ++i) {
        if (b & 1)
            p ^= a;
        uint8_t hi_bit_set = a & 0x80;
        a <<= 1;
        if (hi_bit_set)
            a ^= 0x1B;  // x^8 + x^4 + x^3 + x + 1
        b >>= 1;
    }
    return p;
```

- انجام ضرب دو بایت در میدان گالوا GF(2^8)، که برای تبدیل MixColumns ضروری است.

چگونه این کار انجام می‌شود:

- p برابر صفر قرار داده می‌شود تا حاصل‌ضرب را ذخیره کند.
- **حلقه**:
  - روی هر بیت b تکرار می‌کند.
  - اگر بیت مربوطه b مقدار ۱ داشته باشد، مقدار a به p اضافه می‌شود.
  - a در ۲ ضرب می‌شود (شیفت به چپ)، که شبیه ضرب در x در میدان گالوا است.
  - اگر بیت بالایی a قبل از شیفت مقدار ۱ داشت، a با چندجمله‌ای AES (0x1B) کاهش مدولار می‌شود.
  - b یک بیت به راست شیفت می‌شود تا بیت بعدی بررسی شود.
- مقدار p که حاصل‌ضرب نهایی است را برمیگرداند.
- **چرا از `inline` استفاده شده**:
  - پیشنهاد می‌کند که کامپایلر کد تابع را در محل فراخوانی گسترش دهد، که می‌تواند سربار فراخوانی تابع را کاهش دهد.

پیش‌محاسبه جداول تبدیل

```
void precompute_tables(std::array<std::array<uint8_t, 4>, 256>& T0,
                       std::array<std::array<uint8_t, 4>, 256>& T1,
                       std::array<std::array<uint8_t, 4>, 256>& T2,
                       std::array<std::array<uint8_t, 4>, 256>& T3) {
    uint8_t x = 0x02;   // x
    uint8_t x1 = 0x03;  // x + 1
    for (int a = 0; a < 256; ++a) {
        uint8_t SR = AES_SBOX[a];
        uint8_t mul_x_SR = gf_mul(x, SR);
        uint8_t mul_x1_SR = gf_mul(x1, SR);

        T0[a][0] = mul_x_SR;
        T0[a][1] = SR;
        T0[a][2] = SR;
        T0[a][3] = mul_x1_SR;

        T1[a][0] = mul_x1_SR;
        T1[a][1] = mul_x_SR;
        T1[a][2] = SR;
        T1[a][3] = SR;

        T2[a][0] = SR;
        T2[a][1] = mul_x1_SR;
        T2[a][2] = mul_x_SR;
        T2[a][3] = SR;

        T3[a][0] = SR;
        T3[a][1] = SR;
        T3[a][2] = mul_x1_SR;
        T3[a][3] = mul_x_SR;
    }
}
```

- تمام مقادیر مورد نیاز برای تبدیل MixColumns برای تمام مقادیر بایت‌های ممکن (۰ تا ۲۵۵) را پیش‌محاسبه می‌کند.
- این جداول (T0, T1, T2, T3) از محاسبات اضافی در زمان اجرا (online) جلوگیری می‌کنند و عملکرد را بهبود می‌دهند.

$$\begin{pmatrix} r0 \\ r1 \\ r2 \\ r3 \end{pmatrix} = S_R(w_0)\begin{pmatrix} x \\ 1 \\ 1 \\ x+1 \end{pmatrix} + S_R(w_1)\begin{pmatrix} x+1 \\ x \\ 1 \\ 1 \end{pmatrix} + S_R(w_2)\begin{pmatrix} 1 \\ x+1 \\ x \\ 1 \end{pmatrix} + S_R(w_3)\begin{pmatrix} 1 \\ 1 \\ x+1 \\ x \end{pmatrix}$$

در واقع از فرمول بالا برای این محاسبه این جداول استفاده میکنیم.

تا به جداول زیر برسیم

$$T_0(a)=\begin{bmatrix} xS_R(a) \\ S_R(a) \\ S_R(a) \\ (x+1)S_R(a) \end{bmatrix}, T_1(a)=\begin{bmatrix} (x+1)S_R(a) \\ xS_R(a) \\ S_R(a) \\ S_R(a) \end{bmatrix},$$

$$T_2(a)=\begin{bmatrix} S_R(a) \\ (x+1)S_R(a) \\ xS_R(a) \\ S_R(a) \end{bmatrix}, T_3(a)=\begin{bmatrix} S_R(a) \\ S_R(a) \\ (x+1)S_R(a) \\ xS_R(a) \end{bmatrix}$$

- **x و x1**: ضرایب ضرب مورد استفاده در MixColumns.
- **حلقه روی تمامی مقادیر بایت (۰ تا ۲۵۵)**:
  - **جایگزینی بایت (SR)**: مقدار ورودی با استفاده از S-box جایگزین می‌شود.
  - **محاسبات ضرب**:
    - `mul_x_SR`: نتیجه ضرب SR در x.
    - `mul_x1_SR`: نتیجه ضرب SR در `x + 1`.
  - **پر کردن جداول**:
    - جداول T0، T1، T2 و T3 نتایج پیش‌محاسبه‌شده را ذخیره می‌کنند.

- **هدف از پیش‌محاسبه**:
  - تبدیل MixColumns شامل ضرب‌ها و جمع‌های متعدد در GF(2^8) است.
  - پیش‌محاسبه این مقادیر، سربار محاسباتی را در زمان اجرا کاهش می‌دهد.
- **مزایای عملکردی**:
  - محاسبات مکرر با دسترسی به جدول جایگزین می‌شوند.
  - performance در هنگام پردازش داده‌های بزرگ بهبود می‌یابد.

تبدیل MixColumns

```
inline void mixcolumn_transform(const uint8_t* w,
                                uint8_t* r,
                                const std::array<std::array<uint8_t, 4>, 256>& T0,
                                const std::array<std::array<uint8_t, 4>, 256>& T1,
                                const std::array<std::array<uint8_t, 4>, 256>& T2,
                                const std::array<std::array<uint8_t, 4>, 256>& T3) {
    for (int i = 0; i < 4; ++i) {
        r[i] = T0[w[0]][i] ^ T1[w[1]][i] ^ T2[w[2]][i] ^ T3[w[3]][i];
    }
}
```

- اعمال تبدیل MixColumns روی کلمه 4 بایتی ورودی w و تولید خروجی r.

چگونه این کار انجام می‌شود:

- **حلقه روی هر موقعیت بایت (۰ تا ۳):**
  - از جداول پیش‌محاسبه‌شده برای انجام تبدیل بدون محاسبات اضافی استفاده می‌کند.
  - `w[n]` در واقع nمین بایت کلمه ورودی.

چرا این کار انجام می‌شود:

- **MixColumns در AES و SNOW 2.0:**
  - هدف این تبدیل، انتشار (Diffusion) است که بایت‌های حالت را با یکدیگر ترکیب می‌کند.

محاسبه بایاس تقریب خطی

```
double compute_linear_approximation(uint32_t mask,
    const std::array<std::array<uint8_t, 4>, 256>& T0,
    const std::array<std::array<uint8_t, 4>, 256>& T1,
    const std::array<std::array<uint8_t, 4>, 256>& T2,
    const std::array<std::array<uint8_t, 4>, 256>& T3) {

    auto start_time = std::chrono::high_resolution_clock::now();

    alignas(16) uint8_t w[4] = {
        static_cast<uint8_t>((mask >> 24) & 0xFF),
        static_cast<uint8_t>((mask >> 16) & 0xFF),
        static_cast<uint8_t>((mask >> 8) & 0xFF),
        static_cast<uint8_t>(mask & 0xFF)
    };

    uint64_t match_count = 0;

    const uint64_t max_iterations = 1ULL << 32;
    #pragma omp parallel
    {
        uint64_t thread_match_count = 0;
        #pragma omp for schedule(static)
```

محاسبه بایاس تقریب
خطی برای یک ماسک خاص روی تمامی ورودی‌های ممکن ۳۲ بیتی.

```
auto start_time = std::chrono::high_resolution_clock::now();
```

شروع اندازه‌گیری زمان

```
alignas(16) uint8_t w[4] = {
    static_cast<uint8_t>((mask >> 24) & 0xFF),
    static_cast<uint8_t>((mask >> 16) & 0xFF),
    static_cast<uint8_t>((mask >> 8) & 0xFF),
    static_cast<uint8_t>(mask & 0xFF)
};
```

- **استخراج بایت‌های ماسک**: ماسک ۳۲ بیتی به چهار مؤلفه ۸ بیتی w[0] تا w[3] تقسیم می‌شود.
- **تراز حافظه**: `alignas(16)` تراز ۱۶ بایتی را تضمین می‌کند که عملکرد را بهبود می بخشد.

```
uint64_t match_count = 0;
```

تعداد تطابق‌های پاریتی ورودی و خروجی را شمارش می‌کند.

```
const uint64_t max_iterations = 1ULL << 32;
```

تعداد کل ورودی‌های ممکن (1^32)

پردازش موازی

```
#pragma omp parallel
{
    uint64_t thread_match_count = 0;
    #pragma omp for schedule(static)
    for (uint64_t input_value = 0; input_value < max_iterations; input_value += 8) {

        uint8_t input_ws[8][4];
        uint8_t output_ws[8][4];
        uint8_t input_maskeds[8];
        uint8_t output_maskeds[8];

        for (int batch = 0; batch < 8; ++batch) {
            uint64_t val = input_value + batch;

            input_ws[batch][0] = static_cast<uint8_t>((val >> 24) & 0xFF);
            input_ws[batch][1] = static_cast<uint8_t>((val >> 16) & 0xFF);
            input_ws[batch][2] = static_cast<uint8_t>((val >> 8) & 0xFF);
            input_ws[batch][3] = static_cast<uint8_t>(val & 0xFF);

            mixcolumn_transform(input_ws[batch], output_ws[batch], T0, T1, T2, T3);

            uint8_t input_masked = 0;
            uint8_t output_masked = 0;

            input_masked = (input_ws[batch][0] & w[0]) ^ (input_ws[batch][1] & w[1]) ^
                           (input_ws[batch][2] & w[2]) ^ (input_ws[batch][3] & w[3]);

            output_masked = (output_ws[batch][0] & w[0]) ^ (output_ws[batch][1] & w[1]) ^
                            (output_ws[batch][2] & w[2]) ^ (output_ws[batch][3] & w[3]);

            input_maskeds[batch] = input_masked;
            output_maskeds[batch] = output_masked;
        }

        for (int batch = 0; batch < 8; ++batch) {
            if (__builtin_parity(input_maskeds[batch]) == __builtin_parity(output_maskeds[batch])) {
                thread_match_count += 1;
            }
        }
    }

    #pragma omp atomic
    match_count += thread_match_count;
}
```

- **بخش موازی**: با استفاده از `#pragma omp parallel` یک بخش موازی ایجاد می‌شود.
- **متغیر محلی**: `thread_match_count` از شرایط رقابتی جلوگیری می‌کند.
- **تقسیم کار**: `#pragma omp for` تکرارهای حلقه را بین thread ها تقسیم می‌کند.
- **پردازش گروهی**: در هر تکرار، ۸ ورودی (batch) پردازش می‌شود.
- **عمل اتمی**: اطمینان حاصل می‌کند که به‌روزرسانی‌های `match_count` ایمن هستند (جلوگیری از race condition).

پردازش گروهی (batch) و مقایسه پاریتی

```
for (int batch = 0; batch < 8; ++batch) {
    uint64_t val = input_value + batch;

    input_ws[batch][0] = static_cast<uint8_t>((val >> 24) & 0xFF);
    input_ws[batch][1] = static_cast<uint8_t>((val >> 16) & 0xFF);
    input_ws[batch][2] = static_cast<uint8_t>((val >> 8) & 0xFF);
    input_ws[batch][3] = static_cast<uint8_t>(val & 0xFF);

    mixcolumn_transform(input_ws[batch], output_ws[batch], T0, T1, T2, T3);

    uint8_t input_masked = 0;
    uint8_t output_masked = 0;

    input_masked = (input_ws[batch][0] & w[0]) ^ (input_ws[batch][1] & w[1]) ^
                   (input_ws[batch][2] & w[2]) ^ (input_ws[batch][3] & w[3]);

    output_masked = (output_ws[batch][0] & w[0]) ^ (output_ws[batch][1] & w[1]) ^
                    (output_ws[batch][2] & w[2]) ^ (output_ws[batch][3] & w[3]);

    input_maskeds[batch] = input_masked;
    output_maskeds[batch] = output_masked;
}
```

1. استخراج بایت‌های ورودی
2. انجام تبدیل MixColumns
3. محاسبه ورودی و خروجی ماسک‌شده

```
for (int batch = 0; batch < 8; ++batch) {
    if (__builtin_parity(input_maskeds[batch]) == __builtin_parity(output_maskeds[batch])) {
        thread_match_count += 1;
    }
}
```

4. مقایسه پاریتی

```
#pragma omp atomic
match_count += thread_match_count;
```

اضافه کردن مقدار محاسبه شده هر thread به مقدار اصلی

- **پردازش ورودی**:
  - بایت‌های ورودی از مقدار ۳۲ بیتی `val` استخراج می‌شوند.
  - تبدیل MixColumns روی ورودی اعمال می‌شود تا خروجی تولید شود.
- **ماسک‌گذاری**:
  - ماسک `w` روی ورودی و خروجی اعمال می‌شود.
  - XOR بایت‌های ماسک‌شده محاسبه می‌شود تا `input_masked` و `output_masked` به دست آید.
- **مقایسه پاریتی**:
  - از `__builtin_parity()` برای محاسبه پاریتی استفاده می‌شود.
  - اگر پاریتی‌ها برابر باشند، `thread_match_count` افزایش می‌یابد.

محاسبه بایاس

```
uint64_t total_count = max_iterations;

double bias = static_cast<double>(std::abs(static_cast<int64_t>(match_count - total_count / 2))) / total_count;
```

- بایاس اختلاف مطلق بین احتمال مشاهده‌شده و ۰.۵ است.
- بایاس نزدیک به ۰ نشان‌دهنده خروجی یکنواخت‌تر و مقاومت بیشتر در برابر تحلیل خطی است.

نمایش نتایج

```
auto end_time = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> diff = end_time - start_time;

std::cout << "Mask: 0x" << std::hex << std::setw(8) << std::setfill('0') << mask
          << ", Bias: " << std::dec << std::fixed << std::setprecision(15) << bias
          << ", Time: " << std::dec << std::fixed << std::setprecision(5) << diff.count() << " seconds" << std::endl;

return bias;
```

- **اندازه‌گیری زمان**: زمان سپری‌شده محاسبه می‌شود.
- **خروجی**:
  - **ماسک**: به‌صورت هگزادسیمال (hex) نمایش داده می‌شود.
  - **بایاس**: با ۱۵ رقم اعشار نمایش داده می‌شود.
  - **زمان**: زمان اجرا به ثانیه با ۵ رقم اعشار نمایش داده می‌شود.
  - مقدار bias برگردانده میشود (در صورت نیاز به محاسبات اضافه در (main).

Main

```cpp
int main() {
    std::array<std::array<uint8_t, 4>, 256> T0, T1, T2, T3;
    precompute_tables(T0, T1, T2, T3);

    uint32_t number;
    while (1)
    {
        std::cout << "Enter a 32-bit hexadecimal mask (e.g., 1A3F22BE): ";
        std::cin >> std::hex >> number;

        compute_linear_approximation(number, T0, T1, T2, T3);
    }

    return 0;
}       You, 5 days ago • project
```

```cpp
int main() {
    std::array<std::array<uint8_t, 4>, 256> T0, T1, T2, T3;
    precompute_tables(T0, T1, T2, T3);
```

پیش‌محاسبه جداول

```cpp
uint32_t number;
while (1)
{
    std::cout << "Enter a 32-bit hexadecimal mask (e.g., 1A3F22BE): ";
    std::cin >> std::hex >> number;

    compute_linear_approximation(number, T0, T1, T2, T3);
}

return 0;
```

محاسبه تقریب خطی برای ماسک‌ورودی که توسط کاربر داده میشود.

- **پیش‌محاسبه جداول**: قبل از پردازش ورودی‌ها، جداول تبدیل مقداردهی می‌شوند.
-
- **پردازش روی ماسک ورودی**:
  - روی ماسک ورودی این عملیات تکرار می‌شود تا بایاس محاسبه شوند.

بهینه‌سازی‌ها و افزایش performance

۱. موازی‌سازی با OpenMP

از هسته‌های CPU برای پردازش همزمان داده‌ها استفاده می‌کند.

پیاده‌سازی:

- **`#pragma omp parallel`**: یک بخش موازی ایجاد می‌کند.
- **`#pragma omp for schedule(static)`**: تکرارهای حلقه را بین رشته‌ها تقسیم می‌کند.

مزایا:

- **کاهش زمان اجرا**: با تقسیم کار، محاسبات سریع‌تر انجام می‌شود.
- **استفاده بهینه از CPU**: از تمامی منابع پردازشی موجود استفاده می‌کند.

---

۲. پردازش گروهی (Batch Processing)

چندین ورودی را در یک تکرار حلقه پردازش می‌کند.

پیاده‌سازی:

- **پردازش ۸ ورودی در هر تکرار**:
  - با استفاده از `input_value += 8`
  - و حلقه داخلی `for (int batch = 0; batch < 8; ++batch)`.

مزایا:

- **کاهش سربار حلقه**: تعداد تکرارهای حلقه بیرونی کمتر می‌شود.
- **بهبود عملکرد کش**: پردازش داده‌ها به‌صورت گروهی باعث استفاده بهتر از حافظه کش می‌شود.

---

۳. جداول پیش‌محاسبه‌شده

نتایج موردنیاز برای تبدیل MixColumns را ذخیره می‌کند.

پیاده‌سازی:

- تابع `precompute_tables` تمامی نتایج لازم را محاسبه و ذخیره می‌کند.

مزایا:

- **حذف محاسبات تکراری**: از محاسبه مقادیر مشابه در زمان اجرا جلوگیری می‌کند.
- **افزایش سرعت بخش‌های حیاتی**: تبدیل MixColumns میلیاردها بار اجرا می‌شود و بهینه‌سازی آن تأثیر زیادی دارد.

۴. تراز حافظه (Memory Alignment)

چه کاری انجام می‌دهد:

- داده‌ها را روی مرزهای حافظه خاصی تراز می‌کند.

چگونه پیاده‌سازی شده:

- **`alignas(16)`**: هنگام تعریف آرایه‌های AES S-box و ماسک‌ها استفاده شده است.

مزایا:

- داده‌های تراز شده سریع‌تر دسترسی پیدا می‌کنند.
- تراز بودن داده‌ها برای عملیات SIMD مفید است.

۵. عملیات بیت‌وایز کارآمد

- از عملیات بیت‌وایز برای ماسک‌گذاری و محاسبه پاریتی استفاده می‌کند.

پیاده‌سازی:

- **ماسک‌گذاری و ترکیب بیت‌ها**:
  - از عملگرهای AND (&) و XOR (^) استفاده می‌شود.
- **محاسبه پاریتی**:
  - **`__builtin_parity()`** : تابع داخلی که پاریتی را به‌صورت بهینه محاسبه می‌کند.

مزایا:

- عملیات بیت‌وایز به‌صورت مستقیم در سطح پردازنده انجام می‌شود.
- بخش‌های پرتکرار کد بهینه می‌شوند.

---

۶. توابع Inline

چه کاری انجام می‌دهد:

- به کامپایلر پیشنهاد می‌دهد که کد تابع مستقیماً در محل فراخوانی قرار گیرد.

چگونه پیاده‌سازی شده:

- **`inline`**: برای توابع کوچک و مهم مانند `gf_mul` و `mixcolumn_transform` استفاده شده است.

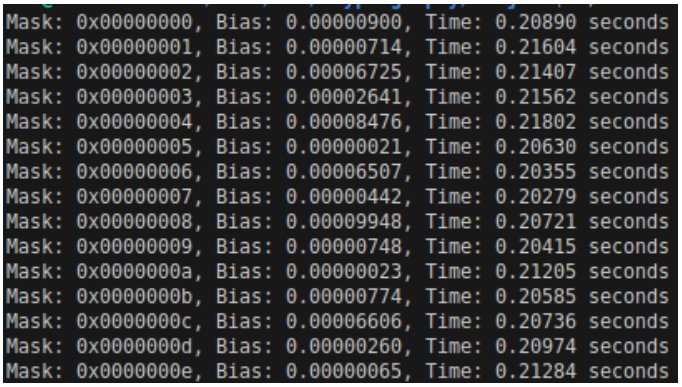
مزایا:

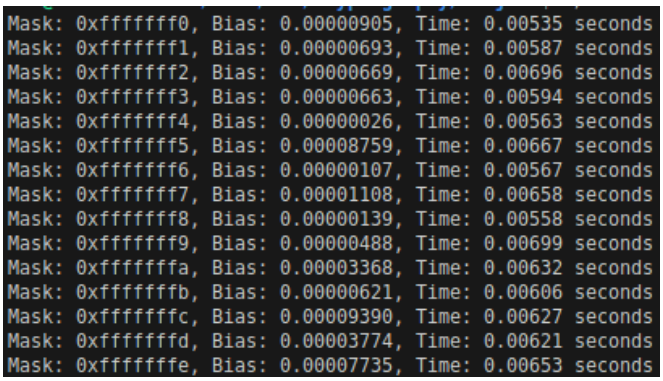
- از سربار مرتبط با فراخوانی تابع جلوگیری می‌کند.
- کامپایلر می‌تواند کد را بهتر بهینه کند.

نتایج

این بهینه‌سازی‌ها باعث شده که پردازش حجم زیادی از داده‌ها در زمان معقول امکان‌پذیر باشد.

با آزمایش در حالت عادی و موازی سازی به نتایج زیر می رسیم.

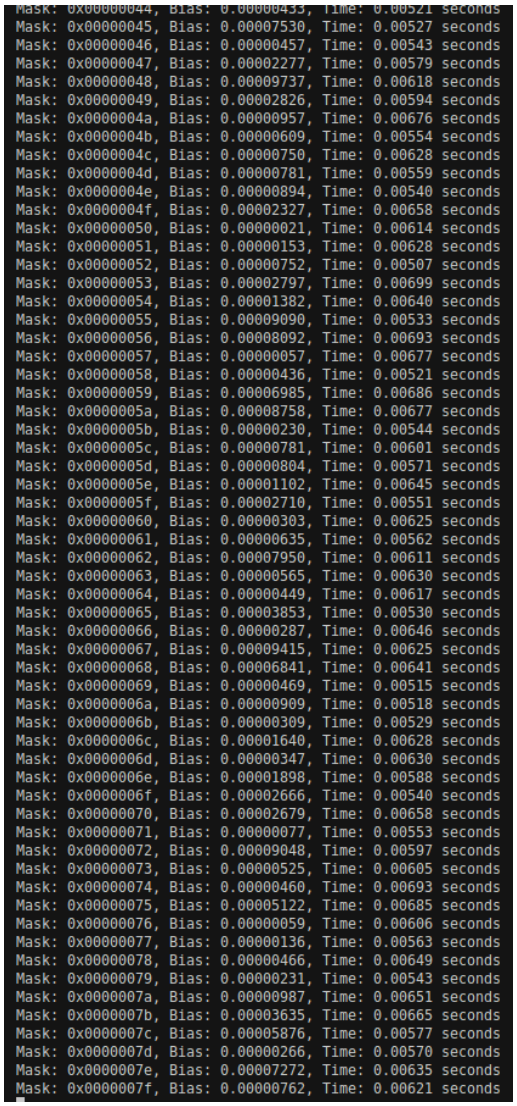


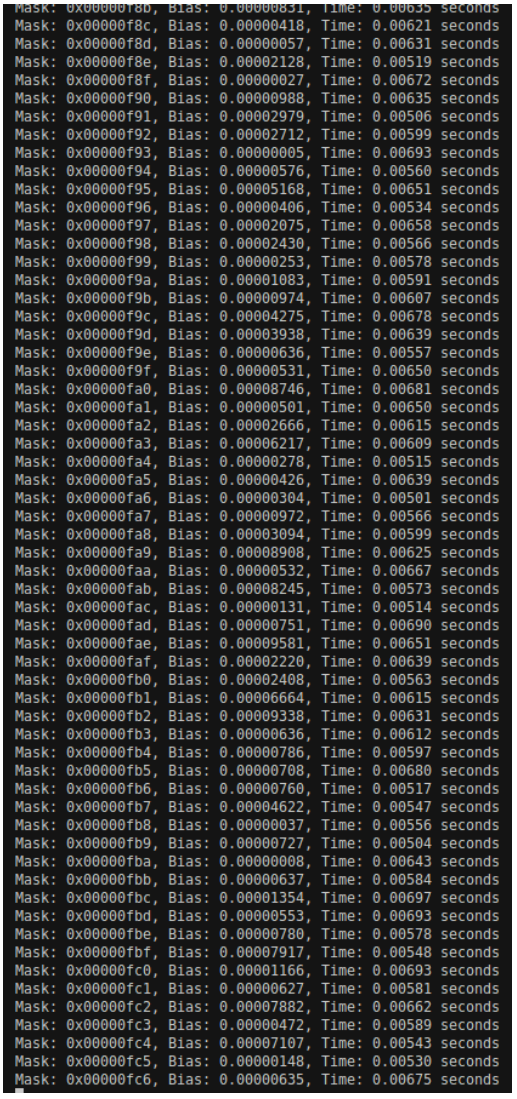


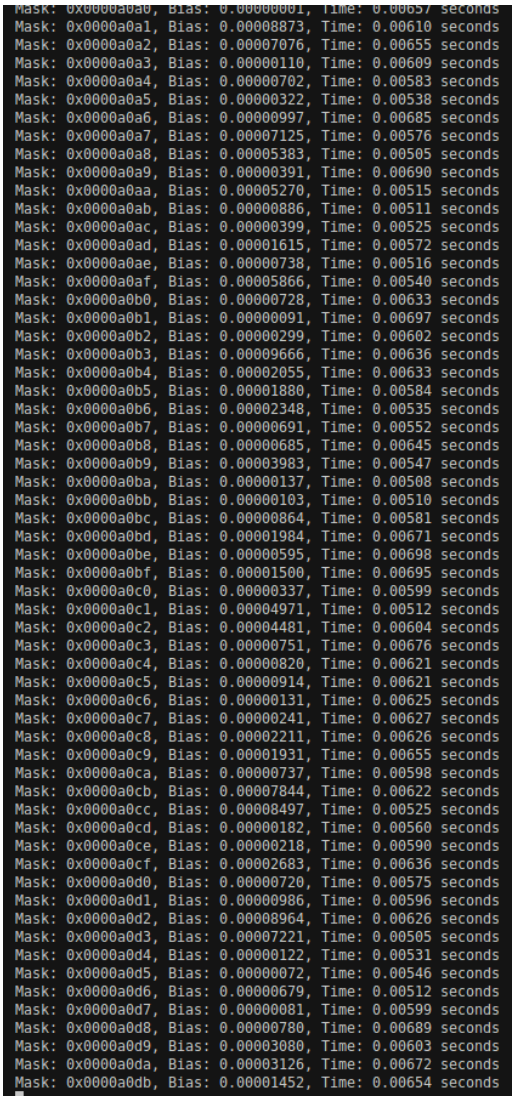
در حالت **بدون** موازی سازی و بهینه سازی در حالت **موازی سازی و بهینه سازی**

همانطور که مشاهده میکنیم زمان محاسبه از حدود **۰.۲** ثانیه به **۰.۰۰۵** الی **۰.۰۰۶** ثانیه می رسد که یک چهلم می شود و عملکرد بسیار افزایش می باید.

خروجی به ازای ارایه ای از همه ی ماسک ها :



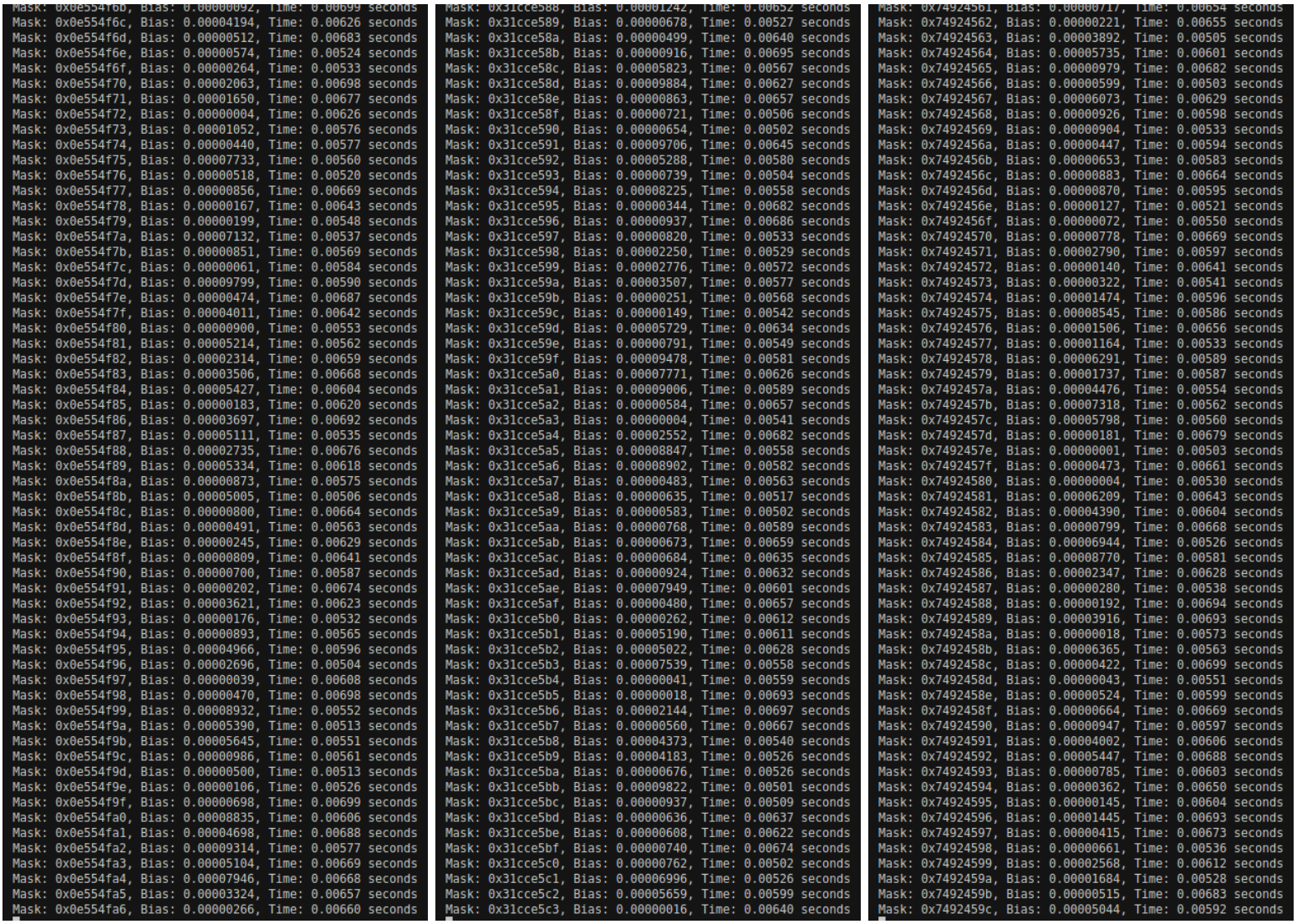




```
Mask: 0x003947ee, Bias: 0.00000790, Time: 0.00655 seconds
Mask: 0x003947ef, Bias: 0.00000032, Time: 0.00533 seconds
Mask: 0x003947f0, Bias: 0.00000046, Time: 0.00623 seconds
Mask: 0x003947f1, Bias: 0.00002318, Time: 0.00609 seconds
Mask: 0x003947f2, Bias: 0.00009212, Time: 0.00587 seconds
Mask: 0x003947f3, Bias: 0.00004177, Time: 0.00579 seconds
Mask: 0x003947f4, Bias: 0.00002731, Time: 0.00568 seconds
Mask: 0x003947f5, Bias: 0.00000497, Time: 0.00561 seconds
Mask: 0x003947f6, Bias: 0.00003729, Time: 0.00577 seconds
Mask: 0x003947f7, Bias: 0.00000244, Time: 0.00514 seconds
Mask: 0x003947f8, Bias: 0.00007608, Time: 0.00689 seconds
Mask: 0x003947f9, Bias: 0.00000774, Time: 0.00660 seconds
Mask: 0x003947fa, Bias: 0.00002437, Time: 0.00585 seconds
Mask: 0x003947fb, Bias: 0.00002751, Time: 0.00568 seconds
Mask: 0x003947fc, Bias: 0.00000967, Time: 0.00579 seconds
Mask: 0x003947fd, Bias: 0.00000610, Time: 0.00653 seconds
Mask: 0x003947fe, Bias: 0.00004144, Time: 0.00512 seconds
Mask: 0x003947ff, Bias: 0.00006867, Time: 0.00554 seconds
Mask: 0x00394800, Bias: 0.00000055, Time: 0.00562 seconds
Mask: 0x00394801, Bias: 0.00000958, Time: 0.00538 seconds
Mask: 0x00394802, Bias: 0.00008944, Time: 0.00527 seconds
Mask: 0x00394803, Bias: 0.00000919, Time: 0.00651 seconds
Mask: 0x00394804, Bias: 0.00009921, Time: 0.00502 seconds
Mask: 0x00394805, Bias: 0.00000872, Time: 0.00514 seconds
Mask: 0x00394806, Bias: 0.00000025, Time: 0.00640 seconds
Mask: 0x00394807, Bias: 0.00000139, Time: 0.00565 seconds
Mask: 0x00394808, Bias: 0.00003325, Time: 0.00535 seconds
Mask: 0x00394809, Bias: 0.00004658, Time: 0.00543 seconds
Mask: 0x0039480a, Bias: 0.00000629, Time: 0.00507 seconds
Mask: 0x0039480b, Bias: 0.00000337, Time: 0.00656 seconds
Mask: 0x0039480c, Bias: 0.00008142, Time: 0.00663 seconds
Mask: 0x0039480d, Bias: 0.00000956, Time: 0.00541 seconds
Mask: 0x0039480e, Bias: 0.00000947, Time: 0.00670 seconds
Mask: 0x0039480f, Bias: 0.00003108, Time: 0.00595 seconds
Mask: 0x00394810, Bias: 0.00003348, Time: 0.00642 seconds
Mask: 0x00394811, Bias: 0.00003967, Time: 0.00588 seconds
Mask: 0x00394812, Bias: 0.00000607, Time: 0.00633 seconds
Mask: 0x00394813, Bias: 0.00007145, Time: 0.00556 seconds
Mask: 0x00394814, Bias: 0.00000022, Time: 0.00696 seconds
Mask: 0x00394815, Bias: 0.00000311, Time: 0.00539 seconds
Mask: 0x00394816, Bias: 0.00008074, Time: 0.00635 seconds
Mask: 0x00394817, Bias: 0.00006923, Time: 0.00506 seconds
Mask: 0x00394818, Bias: 0.00009185, Time: 0.00506 seconds
Mask: 0x00394819, Bias: 0.00005287, Time: 0.00531 seconds
Mask: 0x0039481a, Bias: 0.00000939, Time: 0.00669 seconds
Mask: 0x0039481b, Bias: 0.00004960, Time: 0.00611 seconds
Mask: 0x0039481c, Bias: 0.00000579, Time: 0.00506 seconds
Mask: 0x0039481d, Bias: 0.00000251, Time: 0.00648 seconds
Mask: 0x0039481e, Bias: 0.00000737, Time: 0.00536 seconds
Mask: 0x0039481f, Bias: 0.00002302, Time: 0.00614 seconds
Mask: 0x00394820, Bias: 0.00005296, Time: 0.00625 seconds
Mask: 0x00394821, Bias: 0.00006112, Time: 0.00618 seconds
Mask: 0x00394822, Bias: 0.00003935, Time: 0.00523 seconds
Mask: 0x00394823, Bias: 0.00000583, Time: 0.00563 seconds
Mask: 0x00394824, Bias: 0.00005279, Time: 0.00530 seconds
Mask: 0x00394825, Bias: 0.00004609, Time: 0.00520 seconds
Mask: 0x00394826, Bias: 0.00000407, Time: 0.00650 seconds
Mask: 0x00394827, Bias: 0.00003088, Time: 0.00699 seconds
Mask: 0x00394828, Bias: 0.00006064, Time: 0.00559 seconds
Mask: 0x00394829, Bias: 0.00007592, Time: 0.00572 seconds
```

```
Mask: 0x00f12089, Bias: 0.00000016, Time: 0.00659 seconds
Mask: 0x00f1208a, Bias: 0.00000825, Time: 0.00624 seconds
Mask: 0x00f1208b, Bias: 0.00000794, Time: 0.00591 seconds
Mask: 0x00f1208c, Bias: 0.00002514, Time: 0.00516 seconds
Mask: 0x00f1208d, Bias: 0.00000050, Time: 0.00508 seconds
Mask: 0x00f1208e, Bias: 0.00005797, Time: 0.00544 seconds
Mask: 0x00f1208f, Bias: 0.00000188, Time: 0.00679 seconds
Mask: 0x00f12090, Bias: 0.00000643, Time: 0.00569 seconds
Mask: 0x00f12091, Bias: 0.00000686, Time: 0.00654 seconds
Mask: 0x00f12092, Bias: 0.00000531, Time: 0.00643 seconds
Mask: 0x00f12093, Bias: 0.00000185, Time: 0.00526 seconds
Mask: 0x00f12094, Bias: 0.00000422, Time: 0.00535 seconds
Mask: 0x00f12095, Bias: 0.00009449, Time: 0.00679 seconds
Mask: 0x00f12096, Bias: 0.00000695, Time: 0.00586 seconds
Mask: 0x00f12097, Bias: 0.00000958, Time: 0.00665 seconds
Mask: 0x00f12098, Bias: 0.00000128, Time: 0.00588 seconds
Mask: 0x00f12099, Bias: 0.00009999, Time: 0.00690 seconds
Mask: 0x00f1209a, Bias: 0.00007400, Time: 0.00694 seconds
Mask: 0x00f1209b, Bias: 0.00002983, Time: 0.00571 seconds
Mask: 0x00f1209c, Bias: 0.00000267, Time: 0.00672 seconds
Mask: 0x00f1209d, Bias: 0.00001131, Time: 0.00591 seconds
Mask: 0x00f1209e, Bias: 0.00000653, Time: 0.00642 seconds
Mask: 0x00f1209f, Bias: 0.00001349, Time: 0.00568 seconds
Mask: 0x00f120a0, Bias: 0.00004937, Time: 0.00691 seconds
Mask: 0x00f120a1, Bias: 0.00000205, Time: 0.00623 seconds
Mask: 0x00f120a2, Bias: 0.00004186, Time: 0.00666 seconds
Mask: 0x00f120a3, Bias: 0.00000526, Time: 0.00558 seconds
Mask: 0x00f120a4, Bias: 0.00006235, Time: 0.00527 seconds
Mask: 0x00f120a5, Bias: 0.00000609, Time: 0.00506 seconds
Mask: 0x00f120a6, Bias: 0.00000080, Time: 0.00614 seconds
Mask: 0x00f120a7, Bias: 0.00000477, Time: 0.00573 seconds
Mask: 0x00f120a8, Bias: 0.00004051, Time: 0.00516 seconds
Mask: 0x00f120a9, Bias: 0.00000266, Time: 0.00660 seconds
Mask: 0x00f120aa, Bias: 0.00000989, Time: 0.00577 seconds
Mask: 0x00f120ab, Bias: 0.00000653, Time: 0.00699 seconds
Mask: 0x00f120ac, Bias: 0.00004063, Time: 0.00650 seconds
Mask: 0x00f120ad, Bias: 0.00009082, Time: 0.00540 seconds
Mask: 0x00f120ae, Bias: 0.00006154, Time: 0.00583 seconds
Mask: 0x00f120af, Bias: 0.00008186, Time: 0.00697 seconds
Mask: 0x00f120b0, Bias: 0.00004464, Time: 0.00569 seconds
Mask: 0x00f120b1, Bias: 0.00005665, Time: 0.00600 seconds
Mask: 0x00f120b2, Bias: 0.00002005, Time: 0.00538 seconds
Mask: 0x00f120b3, Bias: 0.00008012, Time: 0.00645 seconds
Mask: 0x00f120b4, Bias: 0.00000938, Time: 0.00653 seconds
Mask: 0x00f120b5, Bias: 0.00002299, Time: 0.00673 seconds
Mask: 0x00f120b6, Bias: 0.00000993, Time: 0.00660 seconds
Mask: 0x00f120b7, Bias: 0.00000276, Time: 0.00545 seconds
Mask: 0x00f120b8, Bias: 0.00000053, Time: 0.00587 seconds
Mask: 0x00f120b9, Bias: 0.00000029, Time: 0.00560 seconds
Mask: 0x00f120ba, Bias: 0.00000575, Time: 0.00672 seconds
Mask: 0x00f120bb, Bias: 0.00000826, Time: 0.00669 seconds
Mask: 0x00f120bc, Bias: 0.00000524, Time: 0.00629 seconds
Mask: 0x00f120bd, Bias: 0.00000437, Time: 0.00569 seconds
Mask: 0x00f120be, Bias: 0.00000197, Time: 0.00611 seconds
Mask: 0x00f120bf, Bias: 0.00007255, Time: 0.00519 seconds
Mask: 0x00f120c0, Bias: 0.00001653, Time: 0.00646 seconds
Mask: 0x00f120c1, Bias: 0.00000299, Time: 0.00625 seconds
Mask: 0x00f120c2, Bias: 0.00000976, Time: 0.00608 seconds
Mask: 0x00f120c3, Bias: 0.00000932, Time: 0.00694 seconds
Mask: 0x00f120c4, Bias: 0.00002185, Time: 0.00536 seconds
```

```
Mask: 0x03e7006e, Bias: 0.00006360, Time: 0.00629 seconds
Mask: 0x03e7006f, Bias: 0.00000145, Time: 0.00686 seconds
Mask: 0x03e70070, Bias: 0.00000977, Time: 0.00518 seconds
Mask: 0x03e70071, Bias: 0.00000083, Time: 0.00642 seconds
Mask: 0x03e70072, Bias: 0.00006863, Time: 0.00625 seconds
Mask: 0x03e70073, Bias: 0.00000146, Time: 0.00634 seconds
Mask: 0x03e70074, Bias: 0.00002514, Time: 0.00615 seconds
Mask: 0x03e70075, Bias: 0.00000321, Time: 0.00504 seconds
Mask: 0x03e70076, Bias: 0.00003534, Time: 0.00664 seconds
Mask: 0x03e70077, Bias: 0.00003321, Time: 0.00700 seconds
Mask: 0x03e70078, Bias: 0.00005031, Time: 0.00510 seconds
Mask: 0x03e70079, Bias: 0.00000838, Time: 0.00647 seconds
Mask: 0x03e7007a, Bias: 0.00000141, Time: 0.00567 seconds
Mask: 0x03e7007b, Bias: 0.00002213, Time: 0.00650 seconds
Mask: 0x03e7007c, Bias: 0.00000750, Time: 0.00524 seconds
Mask: 0x03e7007d, Bias: 0.00000546, Time: 0.00523 seconds
Mask: 0x03e7007e, Bias: 0.00000145, Time: 0.00642 seconds
Mask: 0x03e7007f, Bias: 0.00005772, Time: 0.00524 seconds
Mask: 0x03e70080, Bias: 0.00003951, Time: 0.00700 seconds
Mask: 0x03e70081, Bias: 0.00005628, Time: 0.00644 seconds
Mask: 0x03e70082, Bias: 0.00006179, Time: 0.00559 seconds
Mask: 0x03e70083, Bias: 0.00004391, Time: 0.00699 seconds
Mask: 0x03e70084, Bias: 0.00002318, Time: 0.00530 seconds
Mask: 0x03e70085, Bias: 0.00006852, Time: 0.00523 seconds
Mask: 0x03e70086, Bias: 0.00001580, Time: 0.00550 seconds
Mask: 0x03e70087, Bias: 0.00000442, Time: 0.00562 seconds
Mask: 0x03e70088, Bias: 0.00003044, Time: 0.00597 seconds
Mask: 0x03e70089, Bias: 0.00003435, Time: 0.00510 seconds
Mask: 0x03e7008a, Bias: 0.00003301, Time: 0.00503 seconds
Mask: 0x03e7008b, Bias: 0.00000686, Time: 0.00653 seconds
Mask: 0x03e7008c, Bias: 0.00006644, Time: 0.00513 seconds
Mask: 0x03e7008d, Bias: 0.00008639, Time: 0.00590 seconds
Mask: 0x03e7008e, Bias: 0.00000972, Time: 0.00583 seconds
Mask: 0x03e7008f, Bias: 0.00003196, Time: 0.00646 seconds
Mask: 0x03e70090, Bias: 0.00000467, Time: 0.00637 seconds
Mask: 0x03e70091, Bias: 0.00002518, Time: 0.00682 seconds
Mask: 0x03e70092, Bias: 0.00000461, Time: 0.00589 seconds
Mask: 0x03e70093, Bias: 0.00006252, Time: 0.00586 seconds
Mask: 0x03e70094, Bias: 0.00007955, Time: 0.00653 seconds
Mask: 0x03e70095, Bias: 0.00009230, Time: 0.00542 seconds
Mask: 0x03e70096, Bias: 0.00000948, Time: 0.00669 seconds
Mask: 0x03e70097, Bias: 0.00000191, Time: 0.00634 seconds
Mask: 0x03e70098, Bias: 0.00000170, Time: 0.00698 seconds
Mask: 0x03e70099, Bias: 0.00000621, Time: 0.00651 seconds
Mask: 0x03e7009a, Bias: 0.00005979, Time: 0.00507 seconds
Mask: 0x03e7009b, Bias: 0.00000641, Time: 0.00579 seconds
Mask: 0x03e7009c, Bias: 0.00000958, Time: 0.00560 seconds
Mask: 0x03e7009d, Bias: 0.00002897, Time: 0.00594 seconds
Mask: 0x03e7009e, Bias: 0.00003493, Time: 0.00649 seconds
Mask: 0x03e7009f, Bias: 0.00009423, Time: 0.00619 seconds
Mask: 0x03e700a0, Bias: 0.00003910, Time: 0.00506 seconds
Mask: 0x03e700a1, Bias: 0.00009893, Time: 0.00531 seconds
Mask: 0x03e700a2, Bias: 0.00000811, Time: 0.00561 seconds
Mask: 0x03e700a3, Bias: 0.00000617, Time: 0.00663 seconds
Mask: 0x03e700a4, Bias: 0.00000303, Time: 0.00649 seconds
Mask: 0x03e700a5, Bias: 0.00001260, Time: 0.00561 seconds
Mask: 0x03e700a6, Bias: 0.00000626, Time: 0.00697 seconds
Mask: 0x03e700a7, Bias: 0.00000703, Time: 0.00510 seconds
Mask: 0x03e700a8, Bias: 0.00000791, Time: 0.00626 seconds
Mask: 0x03e700a9, Bias: 0.00000667, Time: 0.00655 seconds
```



خروجی تابع به ازای ماسکهای ورودی از کاربر:

```
Enter a 32-bit hexadecimal mask (e.g., 1A3F22BE): FFFFFCD0
Mask: 0xfffffcd0, Bias: 0.00412432, Time: 0.00578 seconds
```

```
Enter a 32-bit hexadecimal mask (e.g., 1A3F22BE): FFFFFB2F
Mask: 0xfffffb2f, Bias: 0.00002060, Time: 0.00509 seconds
```