به نام خدا

گزارش تمرین کامپیوتری
شماره سه
برنامه نویسی موازی

سید علیرضا میرشفیعی - ۸۱۰۱۰۱۵۳۲
محمد صدرا عباسی - ۸۱۰۱۰۱۴۶۹

محاسبه واریانس

پیاده‌سازی بخش سریال و موازی شامل دو پیمایش کامل روی آرایه داده‌ها است اول برای محاسبه میانگین کل آرایه و دوم برای محاسبه مجموع مربعات تفاضل از میانگین
در نتیجه، بهینه‌سازی اصلی باید در حلقه‌های for این دو پیمایش انجام شود. در واقع bottleneck اصلی دسترسی به حافظه است و هر چقدر که بتوانیم دسترسی به حافظه را بهینه کنیم به نسخه بهتری خواهیم رسید. مزیت اصلی SIMD دربرابر نسخه سریال استفاده از دستورات کمتر برای پردازش یک iteration از حلقه است.

پیاده‌سازی سریال

در این بخش، برای بهینه‌سازی از چهار متغیر جمع‌کننده مستقل استفاده شد. این کار دو دلیل اصلی داشت:

1. افزایش کارایی (ILP): استفاده از یک متغیر جمع‌کننده واحد، باعث ایجاد وابستگی داده‌ای (Data Dependency) در هر تکرار حلقه می‌شود. این وابستگی منجر به توقف (Stall) در خط لوله (Pipeline) پردازنده می‌گردد. با استفاده از ۴ متغیر مستقل، پردازنده می‌تواند از تمام ظرفیت پایپ‌لاین و واحدهای اجرایی (ALU) خود برای اجرای موازی دستورالعمل‌ها (ILP) استفاده کند.
2. مقایسه عادلانه : نسخه موازی از رجیسترهای float ۳۲ بیتی برای جمع استفاده می‌کند. برای اینکه نسخه سریال دقیقاً مشابه نسخه موازی عمل کند، از ۴ جمع‌کننده float استفاده شده است.

پیاده‌سازی SIMD (sse3)

در این پیاده‌سازی، هر دو پیمایش با استفاده از دستورات SSE موازی‌سازی شدند.

- انتخاب عدد ۴: از آنجایی که از دستورات SSE (مانند _mm_add_ps_) روی رجیسترهای ۱۲۸ بیتی (__m128) استفاده می‌شود و نوع داده ما 32 float بیتی است، در هر دستور SIMD دقیقاً ۴ داده به صورت همزمان پردازش می‌شوند. (۱۲۸ / ۳۲ = ۴).
- جمع عمودی: در هر تکرار حلقه، ۴ داده از حافظه (با _mm_loadu_ps_) بارگذاری شده و با دستور _mm_add_ps_ به صورت موازی به یک رجیستر __m128 (که نقش ۴ جمع‌کننده مستقل را ایفا می‌کند) اضافه می‌شوند.

پیاده‌سازی OpenMp

در این بخش برای افزایش کارایی از ترد ها برای موازی سازی استفاده شده. به این ترتیب که هر بخش یا چانک از آرایه را یک ترد به صورت موازی محاسبه می کند و برای اینکار از حداکثر تعداد ترد ها استفاده شده. استراتژی‌های به کار رفته برای این بخش:

- تقسیم‌بندی دستی داده‌ها

  برای تقسیم بار پردازشی بین تردها، بازه آرایه ورودی (ARRAY_SIZE) بر اساس تعداد تردها (omp_get_num_threads) به صورت دستی تقسیم شده است. هر ترد با استفاده از شناسه خود (Thread ID)، محدوده اختصاصی شروع (start) و پایان (end) خود را محاسبه می‌کند. همچنین چانک‌سایزها به گونه‌ای Align شده‌اند که مضرب ۴ باشند تا پردازش در حلقه‌های داخلی بهینه شود

- Data Locality

در این پیاده‌سازی، چون یک ترد واحد مسئولیت پردازش همان بازه داده‌ها را در هر دو فاز بر عهده دارد، داده‌هایی که در فاز اول (محاسبه میانگین) به حافظه نهان (L1/L2 Cache) آورده شده‌اند، در فاز دوم (محاسبه واریانس) همچنان در دسترس هستند.

علاوه بر اینها تمام بهینه سازی هایی که در بخش سریال انجام شده بود در این بخش هم انجام شده

OMP initialization:

```
float total_sum = 0.0f;
float total_sq_diff = 0.0f;
float mean = 0.0f;

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    size_t raw_chunk = ARRAY_SIZE / nthreads;
    size_t chunk_aligned = (raw_chunk / 4) * 4; // each chunk must be a multiple of 4

    size_t start = tid * chunk_aligned;
    size_t end = (tid == nthreads - 1) ? ARRAY_SIZE : start + chunk_aligned;

    float partial_sum = 0.0f;
    float partial_sq = 0.0f;
    size_t i;

    size_t limit = start + ((end - start) / 4) * 4;
```

در این پیاده‌سازی موازی، متغیرهای tid و nthreads برای مدیریت تقسیم کار بین تردها تعریف شده‌اند. tid مشخص می‌کند هر ترد کدام بخش از داده‌ها را پردازش کند و nthreads برای محاسبه اندازه هر بخش استفاده می‌شود. متغیر raw_chunk سهم ساده هر ترد از کل داده‌ها را محاسبه می‌کند، اما برای افزایش کارایی، متغیر chunk_aligned این مقدار را به نزدیک‌ترین مضرب ۴ گرد می‌کند تا با تکنیک‌های SIMD و باز کردن حلقه Unrolling سازگار باشد.

متغیرهای start و end محدوده کاری اختصاصی هر ترد را تعیین می‌کنند، به طوری که ترد آخر مسئول پردازش تمام داده‌های باقیمانده تا انتهای آرایه است . متغیرهای partial_sum و partial_sq به صورت خصوصی برای هر ترد تعریف شده‌اند تا جمع‌های میانی را بدون تداخل حافظه ذخیره کنند. در نهایت، limit حد بالای اجرای حلقه را مشخص می‌کند که مضرب ۴ است، و داده‌های باقی‌مانده بین limit و end در یک حلقه جداگانه و ساده پردازش می‌شوند.

تحلیل منطق محاسباتی و همگام‌سازی تردها:

```
    for (i = start; i < limit; i += 4) {
        partial_sum += data[i];
        partial_sum += data[i+1];
        partial_sum += data[i+2];
        partial_sum += data[i+3];
    }

    for (; i < end; ++i) {
        partial_sum += data[i];
    }

    #pragma omp atomic
    total_sum += partial_sum;

    #pragma omp barrier

    #pragma omp master
    mean = total_sum / ARRAY_SIZE;

    #pragma omp barrier

    for (i = start; i < limit; i += 4) {
        float d0 = data[i] - mean;
        float d1 = data[i+1] - mean;
        float d2 = data[i+2] - mean;
        float d3 = data[i+3] - mean;

        partial_sq += d0 * d0;
        partial_sq += d1 * d1;
        partial_sq += d2 * d2;
        partial_sq += d3 * d3;
    }

    for (; i < end; ++i) {
        float d = data[i] - mean;
        partial_sq += d * d;
    }

    #pragma omp atomic
    total_sq_diff += partial_sq;
}

return total_sq_diff / ARRAY_SIZE;
```

در این بخش، حلقه‌های اصلی محاسبات با استفاده از تکنیک **Loop Unrolling** با پیمانه ۴ پیاده‌سازی شده‌اند. هر ترد جمع جزئی مربوط به چانک خود را محاسبه کرده و در پایان، داده‌های باقیمانده را در یک حلقه پردازش می‌کند. پس از اتمام محاسبات محلی، هر ترد نتیجه خود را با دستور #pragma omp atomic به صورت ایمن به متغیر سراسری total_sum اضافه می‌کند. استفاده از barrier تضمین می‌کند که تمام تردها پیش از محاسبه میانگین توسط ترد اصلی متوقف شوند. سپس همین الگوی محاسباتی برای واریانس تکرار شده و نتایج نهایی تجمیع می‌گردند

پیاده‌سازی OpenMp + SIMD:

در این بخش قابلیت‌های پردازش موازی در سطح ترد (OpenMP) با قابلیت‌های پردازش برداری در سطح دستورالعمل (SIMD) ترکیب شده‌اند. استراتژی کلی تقسیم‌بندی داده‌ها مشابه بخش قبل است، به طوری که تقسیم دستی چانک‌ها و هم‌ترازی حافظه حفظ شده است.

OMP + SIMD initialization:

```
float total_sum = 0.0f;
float total_sq_diff = 0.0f;
float mean = 0.0f;

#pragma omp parallel
{
    __m128 sum_vec;
    __m128 mean_vec;

    float partial_sum = 0.0f;
    float partial_sq = 0.0f;

    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    size_t raw_chunk = ARRAY_SIZE / nthreads;
    size_t chunk_aligned = (raw_chunk / 4) * 4;

    size_t start = tid * chunk_aligned;
    size_t end = (tid == nthreads - 1) ? ARRAY_SIZE : start + chunk_aligned;

    size_t limit = start + ((end - start) / 4) * 4;
    size_t i;
```

متغیرهای sum_vec و mean_vec از نوع برداری (__m128) تعریف شده‌اند تا ۴ داده اعشاری را به صورت همزمان در ثبات‌های پردازنده ذخیره و پردازش کنند. تقسیم‌بندی داده‌ها با متغیرهای raw_chunk و chunk_aligned مشابه روش قبل انجام شده، متغیر sum_vec با صفر مقداردهی اولیه می‌شود تا به عنوان جمع ها جزئی برداری عمل کند، و limit حد نهایی حلقه برداری را مشخص می‌کند.

تحلیل محاسبات SIMD و همگام‌سازی تردها:

```
    sum_vec = _mm_setzero_ps();
    for (i = start; i < limit; i += 4) {
        __m128 data_vec = _mm_load_ps(data + i);
        sum_vec = _mm_add_ps(sum_vec, data_vec);
    }

    __m128 sums = _mm_hadd_ps(sum_vec, sum_vec);
    sums = _mm_hadd_ps(sums, sums);
    partial_sum = _mm_cvtss_f32(sums);

    for (; i < end; ++i) {
        partial_sum += data[i];
    }

    #pragma omp atomic
    total_sum += partial_sum;

    #pragma omp barrier

    #pragma omp master
    mean = total_sum / ARRAY_SIZE;

    #pragma omp barrier

    mean_vec = _mm_set1_ps(mean);
    sum_vec = _mm_setzero_ps();

    for(i = start; i < limit; i += 4){
        __m128 data_vec = _mm_load_ps(data + i);
        __m128 diff_vec = _mm_sub_ps(data_vec, mean_vec);
        __m128 sq_diff_vec = _mm_mul_ps(diff_vec, diff_vec);
        sum_vec = _mm_add_ps(sum_vec, sq_diff_vec);
    }

    sums = _mm_hadd_ps(sum_vec, sum_vec);
    sums = _mm_hadd_ps(sums, sums);
    partial_sq = _mm_cvtss_f32(sums);

    for (; i < end; ++i) {
        float diff = data[i] - mean;
        partial_sq += diff * diff;
    }

    #pragma omp atomic
    total_sq_diff += partial_sq;
}

return total_sq_diff / ARRAY_SIZE;
```

در این بخش، هسته محاسباتی با تلفیق دستورات برداری SSE پیاده‌سازی شده است تا در هر سیکل، ۴ عملیات ریاضی همزمان روی داده‌ها انجام شود. متغیرهای برداری sum_vec برای جمع سریع مقادیر استفاده می‌شوند که در نهایت با عملیات جمع افقی به مقدار اسکالر تبدیل و با دستور atomic در متغیرهای مشترک تجمیع می‌گردند. حلقه‌های اسکالر پایانی وظیفه پردازش داده‌های باقیمانده از

بلوک‌های برداری را دارند. همچنین استفاده از barrier بین دو فاز، تضمین می‌کند که تمامی تردها پیش از شروع محاسبه واریانس منتظر تکمیل محاسبه میانگین بمانند، در حالی که داده‌های مورد نیاز همچنان در کش پردازنده برای دسترسی سریع در دسترس هستند.

تحلیل خروجی و Speed Up

```
--- Variance Calculation ---
Array Size: 16000000 floats

Serial Variance: 0.0831116438 clock cycles: 97677352
SIMD Variance: 0.0831116438 clock cycles: 79454074
OpenMP Variance: 0.0832563266 clock cycles: 26928730
OpenMP SIMD Variance: 0.0833294839 clock cycles: 19959732
Speedup SIMD: 1.2294x
Speedup OpenMP: 3.6273x
Speedup OpenMP + SIMD: 4.8937x
```

نتایج پیاده‌سازی نشان می‌دهد که استفاده از OpenMP با ۸ ترد منجر به افزایش سرعت حدود ۳.۶ برابر شده است، در حالی که انتظار می‌رفت این میزان به ۸ برابر نزدیک باشد. عامل اصلی این است که سرعت پردازش تردها از سرعت انتقال داده‌ها از حافظه اصلی بیشتر است و تردها برای دریافت داده منتظر می‌مانند. علاوه بر این، عواملی مانند سربار مدیریت تردها توسط سیستم‌عامل و هزینه‌های ناشی از همگام‌سازی (Barrier و Atomic) نیز در جلوگیری از رشد خطی عملکرد نقش دارند. با این حال، ترکیب این روش با SIMD توانست با کاهش تعداد دستورات و بهینه‌سازی محاسبات در سطح رجیستر, کارایی را افزایش داده و به ۴.۹ برابر برساند.

افزودن واترمارک

```cpp
int main() {
    Ipp64u start, end;
    Ipp64u time1 , time2;

    Mat base_img = imread("./assets/base.jpg", IMREAD_COLOR);
    Mat water_img = imread("./assets/watermark.png", IMREAD_COLOR);

    if (base_img.empty() || water_img.empty()) {
        cout << "Error: Could not load images." << endl;
        return 1;
    }

    resize(water_img, water_img, base_img.size());

    vector<Mat> base_channels(3), water_channels(3);
    vector<Mat> out_parallel_channels(3);
    vector<Mat> out_serial_channels(3);

    split(base_img, base_channels);
    split(water_img, water_channels);
    for(int i=0; i<3; ++i) {
        out_parallel_channels[i] = Mat::zeros(base_img.size(), CV_8UC1);
        out_serial_channels[i] = Mat::zeros(base_img.size(), CV_8UC1);
    }
    const float total_dim = 1.0f / (float)(base_img.cols + base_img.rows);
```

با استفاده از openCv تصاویر base و watermark خوانده شده و در سه چنل رنگی RGB جدا می شوند همچنین ۳ چنل برای ذخیره خروجی هر پیکسل با ابعاد base از حافظه allocate می شود تا نتیجه محاسبات هر پیکسل برای چنل متناظرش رو در آن ذخیره کنیم و نهایتا با merge کردن سه چنل خروجی تصویر نهایی تولید می شود در ادامه با استفاده از یک روش سریال بهبود یافته و یک روش SIMD پردازش مربوط به هر چنل خروجی را انجام می دهیم

پیاده‌سازی سریال

برای پیاده‌سازی این بخش بهینه‌سازی اصلی در سه سطح انجام شد:

- بهینه‌سازی محاسبه آلفا : به جای محاسبه پرهزینه در داخل حلقه داخلی، از منطق تکراری (Iterative) مشابه نسخه موازی استفاده شد. مقادیر آلفا برای ۸ پیکسل اول محاسبه و در تکرارهای بعدی حلقه، فقط با یک مقدار ثابت $(8/c)$ جمع می‌شوند.
- با هر بار دسترسی به memory یک cache line از حافظه fetch می شود و درون cache قرار می گیرد بنابراین خواندن سطری باعث می شود که کش خراب نشود و حداکثر استفاده از داده های کش شده صورت بگیرد البته خود ذخیره سازی opencv هم به صورت سطر به سطر در حافظه انجام می شود
- بهینه‌سازی حلقه : برای هر سطر، یک ردیف ۸ تایی از پیکسل‌ها پردازش شد و محاسبات در ۸ متغیر مستقل ذخیره گردید. دلایل این کار عبارتند از:

  ۱- استفاده حداکثری از ILP: جلوگیری از وابستگی داده‌ای (Data Dependency) و توقف (Stall) در پایپ‌لاین پردازنده.

  ۲- کاهش سربار حلقه (Loop Overhead): کاهش قابل توجه تعداد تکرارهای حلقه x از N به N/8

  ۳- مقایسه عادلانه با SIMD: این ساختار ۸تایی دقیقاً با نسخه موازی AVX (که ۸ پیکسل را همزمان پردازش می‌کند) مطابقت دارد

پیاده سازی SIMD (avx)

ایده اصلی در بهینه سازی نسخه SIMD (علاوه بر تمام بهینه سازی هایی که در بخش سریال انجام شده) پردازش و محاسبه همزمان ۸ عدد float است(رجیستر ها 256 بیتی هستند) البته محتوای هر پیکسل برای هر کارنال رنگی ۸ بیتی است اما از آنجایی که برای هرکدام باید یک محاسبه از جنس float انجام شود در هر iteration از حلقه داخلی ۸ پیکسل با روش های SIMD محاسبه می شوند.

SIMD initialization:

```
void parallel_blend_avx_channel(const Mat& base_img, const Mat& water_img, Mat& out_img) {
    const int height = base_img.rows;
    const int width = base_img.cols;
    const float total_dim_inv = 1.0f / (float)(width + height);

    // Start parallel region
    __m256 constant_vec = _mm256_set1_ps(total_dim_inv);
    __m256 one_vec = _mm256_set1_ps(1.0f);
    __m256 x_index = _mm256_set_ps(7.0 , 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0); // 0.0 to smallest index (i)
    __m256 x_step = _mm256_set1_ps(8.0f * total_dim_inv);
```

برای بهینه‌سازی حداکثری و کاهش حجم محاسبات، بردارهای ثابت (Vector Constants) که در تمام حلقه‌ها یکسان هستند، یک بار در ابتدای برنامه تعریف شدند. این مقادیر شامل بردار اندیس‌های اولیه (x_index) (برای محاسبه آلفای ۸ پیکسل اول)، بردار گام (x_step) (که آلفای فعلی را برای ۸ پیکسل بعدی آپدیت می‌کند)، بردار one_vec (برای محاسبه alpha-1) و بردار constant_vec (حاصل تقسیم 1.0 / (w+h)) می‌باشند. این کار تضمین می‌کند که دستورات پرهزینه _mm256_set_ps و _mm256_set1_ps به جای اجرای مکرر، تنها یک بار اجرا شوند.

SIMD outer loop:

```
for (int y = 0; y < height; ++y) {
    // access to current pixel
    const unsigned char* base_ptr = base_img.ptr<unsigned char>(y);
    unsigned char* out_ptr = out_img.ptr<unsigned char>(y);
    const unsigned char* water_ptr = water_img.ptr<unsigned char>(y);

    __m256 alpha_by_y = _mm256_set1_ps((float)y * total_dim_inv); // y/(height + width)
    __m256 x_index_mul_by_constant = _mm256_mul_ps(x_index, constant_vec); // i/(height + width)
    __m256 alpha_by_x_y = _mm256_add_ps(alpha_by_y, x_index_mul_by_constant); // i/(height + width) + y/(height + width)

    int x = 0;
    const int limit = (width / 8) * 8;
```

از آنجایی که مقدار آلفا شامل جمع دو متغیر وابسته x و y است (a = x/C + y/C)، مقدار ثابتِ وابسته به y در حلقه بیرونی محاسبه شده و با دستور _mm256_set1_ps در بردار alpha_by_y ذخیره می‌شود. سپس، با استفاده از x_index_mul_by_constant (که از قبل محاسبه شده)، آلفای اولیه (alpha_by_x_y) برای ۸ پیکسل اول سطر، با یک دستور _mm256_add_ps آماده می‌شود. همچنین، برای دسترسی به کانال‌ها، از اشاره‌گرهای خام (base_ptr, water_ptr, out_ptr) استفاده شده تا داده‌ها برای بارگذاری موازی در حلقه داخلی آماده باشند.

SIMD inner loop:

```
for(; x < limit; x += 8){
    __m256 one_minus_alpha_vec = _mm256_sub_ps(one_vec, alpha_by_x_y); // 1 - alpha
    __m128i v_base_u8  = _mm_loadl_epi64((__m128i const*)(base_ptr + x));
    __m128i v_water_u8 = _mm_loadl_epi64((__m128i const*)(water_ptr + x));

    __m256i v_base_i32  = _mm256_cvtepu8_epi32(v_base_u8); // it can be done with a single instruction
    __m256i v_water_i32 = _mm256_cvtepu8_epi32(v_water_u8);
    __m256 v_base_f  = _mm256_cvtepi32_ps(v_base_i32);
    __m256 v_water_f = _mm256_cvtepi32_ps(v_water_i32);

    // calculate result for 8 pixels
    __m256 term1 = _mm256_mul_ps(v_water_f, alpha_by_x_y);
    __m256 term2 = _mm256_mul_ps(v_base_f, one_minus_alpha_vec);
    __m256 v_result_f = _mm256_add_ps(term1, term2);

    // convert result to unsigned char
    __m256i v_result_i32 = _mm256_cvtps_epi32(v_result_f); // convert to 32-bit integer
    __m128i v_low_i32  = _mm256_castsi256_si128(v_result_i32); // extract lower 128 bits
    __m128i v_high_i32 = _mm256_extractf128_si256(v_result_i32, 1); // extract upper 128 bits
    __m128i v_result_i16 = _mm_packus_epi32(v_low_i32, v_high_i32); // pack unsigned 16-bit integers
    __m128i v_result_u8  = _mm_packus_epi16(v_result_i16, _mm_setzero_si128()); // pack unsigned 8-bit integers
    _mm_storel_epi64((__m128i*)(out_ptr + x), v_result_u8); // store packed unsigned 8-bit integers

    // update alpha for next 8 pixels
    alpha_by_x_y = _mm256_add_ps(alpha_by_x_y, x_step); // new alpha = i/(height + width) + y/(height + width) + 8/(height + width)
}
```

در حلقه داخلی، ۸ پیکسل با دستور _mm_loadl_epi64_ بارگذاری شده و سپس با دستورات AVX2 (مانند _mm256_cvtepu8_epi32_ و _mm256_cvtepi32_ps_) به ۸ عدد ۳۲ بیتی float تبدیل می‌شوند. سپس تمام محاسبات ترکیب (Blend) (شامل _mm256_mul_ps_ و _mm256_add_ps_) به صورت همزمان روی این ۸ پیکسل انجام می‌گیرد. نتایج float نهایی با زنجیره‌ای از دستورات تبدیل و فشرده‌سازی به ۸ پیکسل uchar تبدیل شده و با _mm_storel_epi64_ در حافظه خروجی ذخیره می‌شوند. در انتهای تکرار، بردار آلفا (alpha_by_x_y) با یک دستور _mm256_add_ps_ (جمع با x_step)، به صورت بهینه برای ۸ پیکسل بعدی آپدیت می‌شود.

فشرده سازی داده های float 32 Bit:

```
// convert result to unsigned char
__m256i v_result_i32 = _mm256_cvtps_epi32(v_result_f); // convert to 32-bit integer
__m128i v_low_i32  = _mm256_castsi256_si128(v_result_i32); // extract lower 128 bits
__m128i v_high_i32 = _mm256_extractf128_si256(v_result_i32, 1); // extract upper 128 bits
__m128i v_result_i16 = _mm_packus_epi32(v_low_i32, v_high_i32); // pack unsigned 16-bit integers
__m128i v_result_u8  = _mm_packus_epi16(v_result_i16, _mm_setzero_si128()); // pack unsigned 8-bit integers
_mm_storel_epi64((__m128i*)(out_ptr + x), v_result_u8); // store packed unsigned 8-bit integers
```

ابتدا با دستور _mm256_cvtps_epi32 بخش اعشار فلوت های ۳۲ بیتی حذف شده و حاصل ۸ عدد اینت ۳۲ بیتی می شود. سپس با دو دستور بعدی ۱۲۸ بیت رتبه پایین و رتبه بالا در دو رجیستر ۱۲۸ بیتی ذخیره می شوند. دستور _mm_packus_epi32 این دو رجیستر ۱۲۸ بیتی را که در کل حاوی ۸ مقدار خروجی هستند در یک رجیستر ۱۲۸ بیتی حاوی داده های ۱۶ بیتی بدون علامت پک می کند و این کار را با روش اشباع بدون علامت انجام می دهد. دستور آخر نیز ۸ عدد ۱۶ بیتی بدون علامت را به ۸ عدد ۸ بیتی بدون علامت فشرده می کند به روش اشباع بدون علامت . در نهایت نتیجه ۸ پیکسلی که باید در کانال خروجی بگیرد در ۸ بایت سمت راست این رجیستر ۱۲۸ بیتی قرار می گیرد.

پیاده سازی OpenMp

```
#pragma omp parallel for schedule(static)
for (int y = 0; y < height; y++) {

    const unsigned char* base_ptr = base_img.ptr<unsigned char>(y);
    const unsigned char* water_ptr = water_img.ptr<unsigned char>(y);
    unsigned char* out_ptr = out_img.ptr<unsigned char>(y);

    float a0, a1, a2, a3, a4, a5, a6, a7;
    float out_f0, out_f1, out_f2, out_f3, out_f4, out_f5, out_f6, out_f7;

    const float alpha_y = (float)y * c;

    a0 = alpha_y + (x_indices[0] * c);
    a1 = alpha_y + (x_indices[1] * c);
    a2 = alpha_y + (x_indices[2] * c);
    a3 = alpha_y + (x_indices[3] * c);
    a4 = alpha_y + (x_indices[4] * c);
    a5 = alpha_y + (x_indices[5] * c);
    a6 = alpha_y + (x_indices[6] * c);
    a7 = alpha_y + (x_indices[7] * c);

    int x = 0;
    const int limit = (width / 8) * 8;

    for (; x < limit; x += 8) {

        out_f0 = (float)water_ptr[x+0] * a0 + (float)base_ptr[x+0] * (one - a0);
        out_f1 = (float)water_ptr[x+1] * a1 + (float)base_ptr[x+1] * (one - a1);
        out_f2 = (float)water_ptr[x+2] * a2 + (float)base_ptr[x+2] * (one - a2);
        out_f3 = (float)water_ptr[x+3] * a3 + (float)base_ptr[x+3] * (one - a3);
        out_f4 = (float)water_ptr[x+4] * a4 + (float)base_ptr[x+4] * (one - a4);
        out_f5 = (float)water_ptr[x+5] * a5 + (float)base_ptr[x+5] * (one - a5);
        out_f6 = (float)water_ptr[x+6] * a6 + (float)base_ptr[x+6] * (one - a6);
        out_f7 = (float)water_ptr[x+7] * a7 + (float)base_ptr[x+7] * (one - a7);

        out_ptr[x+0] = (unsigned char)out_f0;
        out_ptr[x+1] = (unsigned char)out_f1;
        out_ptr[x+2] = (unsigned char)out_f2;
        out_ptr[x+3] = (unsigned char)out_f3;
        out_ptr[x+4] = (unsigned char)out_f4;
        out_ptr[x+5] = (unsigned char)out_f5;
        out_ptr[x+6] = (unsigned char)out_f6;
        out_ptr[x+7] = (unsigned char)out_f7;

        a0 += x_step; a1 += x_step; a2 += x_step; a3 += x_step;
        a4 += x_step; a5 += x_step; a6 += x_step; a7 += x_step;
    }

    for (; x < width; ++x) {
}
```

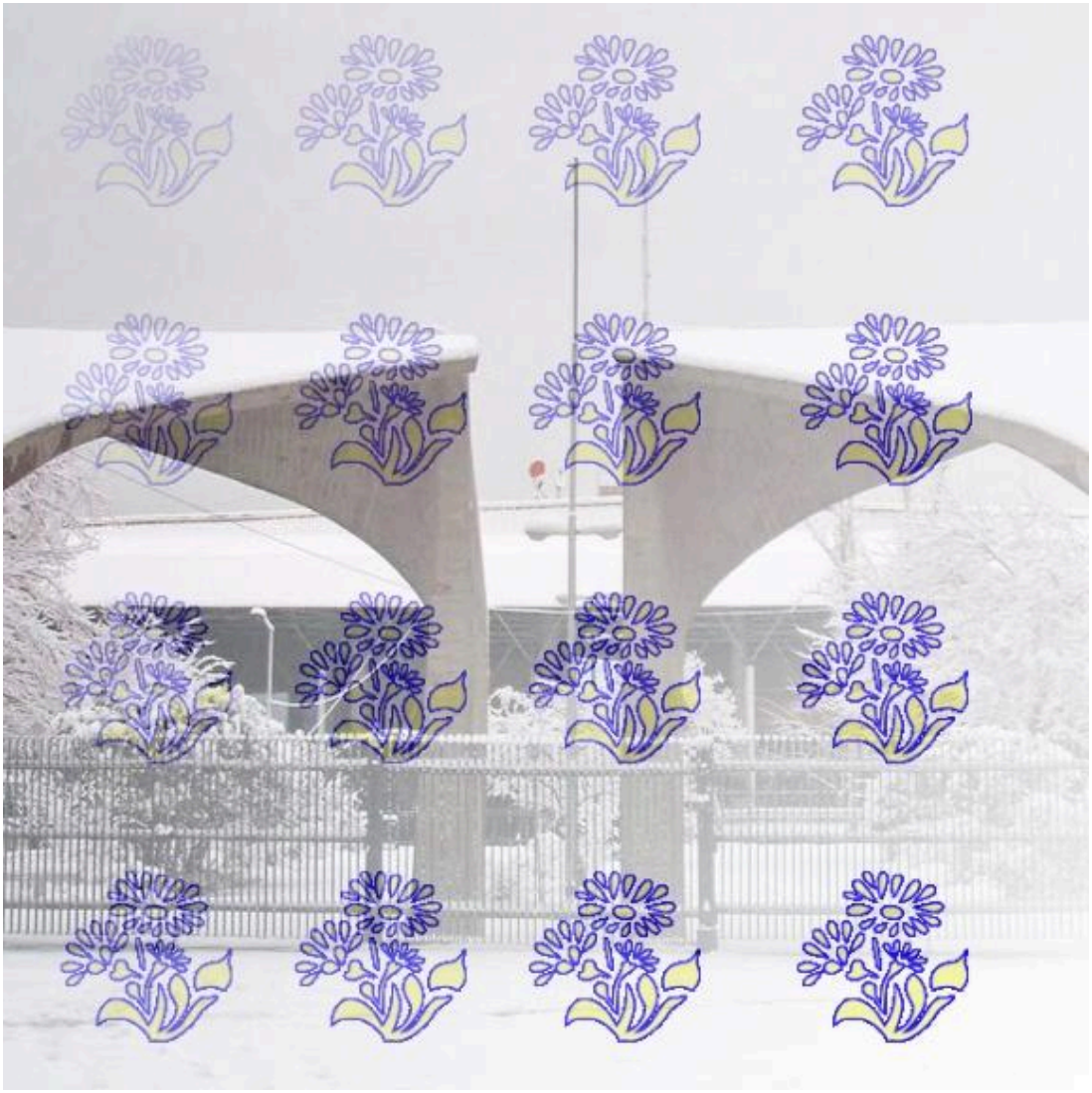
در این پیاده‌سازی، جهت دستیابی به حداکثر کارایی، عملیات موازی‌سازی بر روی حلقه بیرونی (سطرها) اعمال شده است. انتخاب این استراتژی به جای تقسیم‌بندی‌های کوچکتر (مانند ستون‌ها)، باعث می‌شود تا یک بلوک بزرگ و پیوسته از حافظه به هر هسته اختصاص یابد که دو مزیت دارد: اول اینکه سربار زمانی ناشی از مدیریت و ایجاد مکرر تردها را حذف می‌کند و دوم اینکه با دور نگه داشتن محدوده کاری تردها از یکدیگر، مانع از تداخل حافظه می‌شود. همچنین از Static Scheduling به دلیل یکنواخت بودن محاسبات ریاضی برای تمام پیکسل‌ها استفاده شده این روش کارها را در ابتدای اجرا به صورت مساوی تقسیم کرده و هزینه هماهنگی بین هسته‌ها در حین اجرا را به صفر می‌رساند. در نهایت، تعریف تمامی متغیرهای موقت و پوینترها در داخل حلقه موازی، فضای حافظه هر ترد کاملاً مجزا شده تا از هرگونه تداخل در دسترسی به متغیرهای مشترک جلوگیری شود. دیگر مواردی که در نسخه سریال به آن اشاره شده بود در این نسخه نیز به طور مشابه وجود دارد

پیاده سازی OpenMp + SIMD

```
#pragma omp parallel for schedule(static)
for (int y = 0; y < height; ++y) {

    const unsigned char* base_ptr = base_img.ptr<unsigned char>(y);
    const unsigned char* water_ptr = water_img.ptr<unsigned char>(y);
    unsigned char* out_ptr = out_img.ptr<unsigned char>(y);

    __m256 alpha_by_y = _mm256_set1_ps((float)y * total_dim_inv);
    __m256 x_index_mul_by_constant = _mm256_mul_ps(x_index_base, constant_vec);
    __m256 alpha_by_x_y = _mm256_add_ps(alpha_by_y, x_index_mul_by_constant);

    int x = 0;
    const int limit = (width / 8) * 8;

    for(; x < limit; x += 8){
        __m256 one_minus_alpha_vec = _mm256_sub_ps(one_vec, alpha_by_x_y);

        __m128i v_base_u8  = _mm_loadl_epi64((__m128i const*)(base_ptr + x));
        __m128i v_water_u8 = _mm_loadl_epi64((__m128i const*)(water_ptr + x));

        __m256i v_base_i32  = _mm256_cvtepu8_epi32(v_base_u8);
        __m256i v_water_i32 = _mm256_cvtepu8_epi32(v_water_u8);
        __m256 v_base_f  = _mm256_cvtepi32_ps(v_base_i32);
        __m256 v_water_f = _mm256_cvtepi32_ps(v_water_i32);

        // Calculate: (water * alpha) + (base * (1-alpha))
        __m256 term1 = _mm256_mul_ps(v_water_f, alpha_by_x_y);
        __m256 term2 = _mm256_mul_ps(v_base_f, one_minus_alpha_vec);
        __m256 v_result_f = _mm256_add_ps(term1, term2);

        // Compress & Store
        __m256i v_result_i32 = _mm256_cvtps_epi32(v_result_f);
        __m128i v_low_i32  = _mm256_castsi256_si128(v_result_i32);
        __m128i v_high_i32 = _mm256_extractf128_si256(v_result_i32, 1);
        __m128i v_result_i16 = _mm_packus_epi32(v_low_i32, v_high_i32);
        __m128i v_result_u8  = _mm_packus_epi16(v_result_i16, _mm_setzero_si128());

        _mm_storel_epi64((__m128i*)(out_ptr + x), v_result_u8);

        // Update Alpha
        alpha_by_x_y = _mm256_add_ps(alpha_by_x_y, x_step);
    }

    for (; x < width; ++x) {…
}
```

در این بخش استراتژی‌های پیاده‌سازی شده در دو بخش قبل با یکدیگر ترکیب شدند یعنی در لایه بیرونی، از OpenMP با زمان‌بندی static برای تقسیم‌بندی تصویر به بلوک‌های بزرگ سطری و تخصیص آن‌ها به تردها استفاده شده (موازی‌سازی در سطح وظیفه) و در لایه درونی، هر ترد به جای پردازش اسکالر، از SIMD برای پردازش همزمان ۸ پیکسل استفاده می‌کند (موازی‌سازی در سطح داده).
همچنین رجیسترهای AVX (مانند constant_vec و one_vec) در خارج از ناحیه موازی تعریف و مقداردهی شدند تا به صورت Shared و فقط‌خواندنی در اختیار تردها قرار گیرند. در مقابل، تمامی متغیرهای محاسباتی و اشاره‌گرها در محدوده Private قرار گرفتند تا تداخل حافظه بوجود نیاید.

تحلیل خروجی و SpeedUp



```
Serial (Optimized) clock cycles: 2458424
Parallel (AVX) clock cycles:     482342
OpenMP clock cycles:             1046314
OpenMP + SIMD clock cycles:      106896
SIMD speedup: 5.0968x
OpenMP speedup: 2.3496x
OpenMP + SIMD speedup: 22.9983x
```

در نسخه OpenMP، نتیجه ۲.۳۵ برابری نشان می‌دهد که اگرچه توزیع بار بین هسته‌ها انجام شده، اما الگوریتم دارای گلوگاه پهنای باند حافظه است یعنی سرعت درخواست داده‌ها توسط هسته‌های متعدد، از سرعت پاسخگویی RAM بیشتر بوده و هسته‌ها زمان زیادی را در انتظار برای داده بودند اما در نسخه ترکیبی، یک جهش عملکردی با تسریع ۲۳ برابری رخ داده. این نتیجه حاصل دو سطح موازی‌سازی است: OpenMP کار را بین هسته های پردازنده تقسیم کرده و SIMD درون هر ترد،در هر گام ۸ پیکسل را محاسبه می کند . این دو باعث شده تا داده‌های بارگذاری شده در کش با حداکثر کارایی پردازش شوند.

Sobel

1. نسخه سریال

نسخه سریال صرفا پیاده سازی الگوریتم sobel با بهینه سازی های جزئی.

- تبدیل تصویر ورودی به Grayscale

```
    // Using aligned memory
    Mat gray_float = createAlignedMat(gray_img.rows, gray_img.cols, CV_32F);
    gray_img.convertTo(gray_float, CV_32F);
```

- اعمال فیلتر blur با استفاده از gaussian kernel و تابع convolve_serial

```
// --- Serial Functions (unchanged) ---
void convolve_serial(const Mat& src, Mat& dst, const float* kernel, float norm_factor) {
    int rows = src.rows;
    int cols = src.cols;
    dst.create(rows, cols, CV_32F);
    dst.setTo(Scalar(0.0));

    for (int y = 1; y < rows - 1; ++y) {
        const float* p_top = src.ptr<float>(y - 1);
        const float* p_mid = src.ptr<float>(y);
        const float* p_bot = src.ptr<float>(y + 1);
        float* p_dst = dst.ptr<float>(y);

        for (int x = 1; x < cols - 1; ++x) {
            float sum = 0.0f;
            sum += p_top[x - 1] * kernel[0];
            sum += p_top[x]     * kernel[1];
            sum += p_top[x + 1] * kernel[2];
            sum += p_mid[x - 1] * kernel[3];
            sum += p_mid[x]     * kernel[4];
            sum += p_mid[x + 1] * kernel[5];
            sum += p_bot[x - 1] * kernel[6];
            sum += p_bot[x]     * kernel[7];
            sum += p_bot[x + 1] * kernel[8];
            p_dst[x] = sum / norm_factor;
        }
    }
}
```

این تابع کرنل 3*3 ورودی را بر روی تصویر convolve میکند. (بدون بهینه سازی)

- اعمال دو فیلتر sobel x,y بر روی تصویر blur شده برای بدست آوردن گرادیان های افقی و عمودی (edge x,y)
- در نهایت محاسبه اندازه گرادیان هر پیکسل

```cpp
void sobel_magnitude_fused_serial(const Mat& blurred, Mat& magnitude) {
    int rows = blurred.rows;
    int cols = blurred.cols;
    magnitude.create(rows, cols, CV_32F);
    magnitude.setTo(Scalar(0.0));

    for (int y = 1; y < rows - 1; ++y) {
        const float* p_top = blurred.ptr<float>(y - 1);
        const float* p_mid = blurred.ptr<float>(y);
        const float* p_bot = blurred.ptr<float>(y + 1);
        float* p_mag = magnitude.ptr<float>(y);

        for (int x = 1; x < cols - 1; ++x) {
            float gx = (p_top[x - 1] * sobel_x_kernel[0]) + (p_top[x + 1] * sobel_x_kernel[2]) +
                       (p_mid[x - 1] * sobel_x_kernel[3]) + (p_mid[x + 1] * sobel_x_kernel[5]) +
                       (p_bot[x - 1] * sobel_x_kernel[6]) + (p_bot[x + 1] * sobel_x_kernel[8]);

            float gy = (p_top[x - 1] * sobel_y_kernel[0]) + (p_top[x] * sobel_y_kernel[1]) + (p_top[x + 1] * sobel_y_kernel[2]) +
                       (p_bot[x - 1] * sobel_y_kernel[6]) + (p_bot[x] * sobel_y_kernel[7]) + (p_bot[x + 1] * sobel_y_kernel[8]);

            p_mag[x] = sqrt(gx * gx + gy * gy);
        }
    }
}
```

تنها بهینه سازی در این بخش، merge کردن 3 حلقه convolve sobel_x, convolve sobel_y, magnitude با یکدیگر و در یک پیمایش بود، تا کمی memory bound را کاهش دهد.

2. نسخه موازی

در این نسخه، قصد داریم الگوریتم پیاده سازی شده در بخش سریال را با استفاده از دستورالعمل های SIMD و به طور خاط در این بخش AVX، برای پردازش همزمان 8 مقدار float متفاوت.

a. موازی سازی تابع convolve_serial

```
for (int y = 1; y < rows - 1; ++y) {
    const float* p_top = src.ptr<float>(y - 1);
    const float* p_mid = src.ptr<float>(y);
    const float* p_bot = src.ptr<float>(y + 1);
    float* p_dst = dst.ptr<float>(y);

    for (int x = 1; x < cols - 1; ++x) {
        float sum = 0.0f;
        sum += p_top[x - 1] * kernel[0];
        sum += p_top[x]     * kernel[1];
        sum += p_top[x + 1] * kernel[2];

        sum += p_mid[x - 1] * kernel[3];
        sum += p_mid[x]     * kernel[4];
        sum += p_mid[x + 1] * kernel[5];

        sum += p_bot[x - 1] * kernel[6];
        sum += p_bot[x]     * kernel[7];
        sum += p_bot[x + 1] * kernel[8];
        p_dst[x] = sum / norm_factor;
    }
}
```

- Vectorization (برداری سازی):

تلاش برای تغییر پردازش از پیکسل به پیکسل، به بردار به بردار.

الگو:

```
sum[x] = (p_top[x-1] * k[0]) + (p_top[x] * k[1]) + (p_top[x+1] * k[2]) +
         (p_mid[x-1] * k[3]) + (p_mid[x] * k[4]) + (p_mid[x+1] * k[5]) +
         (p_bot[x-1] * k[6]) + (p_bot[x] * k[7]) + (p_bot[x+1] * k[8]);

sum[x + 1] = (p_top[x] * k[0]) + (p_top[x + 1] * k[1]) + (p_top[x + 2] * k[2])
           + (p_mid[x] * k[3]) + (p_mid[x + 1] * k[4]) + (p_mid[x + 2] * k[5])
```

+ (p_bot[x] * k[6]) + (p_bot[x + 1] * k[7]) + (p_bot[x + 2] * k[8]);

محاسبه x+1 همان محاسبه x است، که روی داده ها یک پیکسل به جلو شیفت یافته، که حاوی بخش های تکراریست.

به عنوان مثال، هر دو پیکسل با $k[0]$ ضرب میشوند.

پس ما میتوانیم این عملیات را برای هر عضو کرنل، در کنار 8 پیکسل به صورت موازی انجام دهیم.

به عبارتی هر کرنل را در یک رجیستر mm256 به عدد 8 بار کپی میکنیم.

```
const __m256 k_vec_0 = _mm256_set1_ps(kernel[0]);
const __m256 k_vec_1 = _mm256_set1_ps(kernel[1]);
const __m256 k_vec_2 = _mm256_set1_ps(kernel[2]);
const __m256 k_vec_3 = _mm256_set1_ps(kernel[3]);
const __m256 k_vec_4 = _mm256_set1_ps(kernel[4]);
const __m256 k_vec_5 = _mm256_set1_ps(kernel[5]);
const __m256 k_vec_6 = _mm256_set1_ps(kernel[6]);
const __m256 k_vec_7 = _mm256_set1_ps(kernel[7]);
const __m256 k_vec_8 = _mm256_set1_ps(kernel[8]);
const __m256 k_vec_norm = _mm256_set1_ps(norm_factor);
```

سپس از آنجایی که کرنل ما 3*3 هست، عملیات شیفت دادن را 3 بار تکرار میکنیم و محاسبات موازی را برای 8 پیکسل انجام میدهیم، که شامل load کردن پیکسل ها و سپس محاسبه برداری هر یک میباشد.

```
for (int y = y_start; y < y_end; ++y) {
    const float* p_top = src.ptr<float>(y - 1);
    const float* p_mid = src.ptr<float>(y);
    const float* p_bot = src.ptr<float>(y + 1);
    float* p_dst = dst.ptr<float>(y);

    int x = 1;

    for (; x <= limit; x += 8) {
        const __m256 p0 = _mm256_loadu_ps(&p_top[x - 1]);
        const __m256 p1 = _mm256_loadu_ps(&p_top[x]);
        const __m256 p2 = _mm256_loadu_ps(&p_top[x + 1]);
        const __m256 p3 = _mm256_loadu_ps(&p_mid[x - 1]);
        const __m256 p4 = _mm256_loadu_ps(&p_mid[x]);
        const __m256 p5 = _mm256_loadu_ps(&p_mid[x + 1]);
        const __m256 p6 = _mm256_loadu_ps(&p_bot[x - 1]);
        const __m256 p7 = _mm256_loadu_ps(&p_bot[x]);
        const __m256 p8 = _mm256_loadu_ps(&p_bot[x + 1]);

        __m256 sum_vec = _mm256_setzero_ps();
        sum_vec = _mm256_fmadd_ps(p0, k_vec_0, sum_vec);
        sum_vec = _mm256_fmadd_ps(p1, k_vec_1, sum_vec);
        sum_vec = _mm256_fmadd_ps(p2, k_vec_2, sum_vec);
        sum_vec = _mm256_fmadd_ps(p3, k_vec_3, sum_vec);
        sum_vec = _mm256_fmadd_ps(p4, k_vec_4, sum_vec);
        sum_vec = _mm256_fmadd_ps(p5, k_vec_5, sum_vec);
        sum_vec = _mm256_fmadd_ps(p6, k_vec_6, sum_vec);
        sum_vec = _mm256_fmadd_ps(p7, k_vec_7, sum_vec);
        sum_vec = _mm256_fmadd_ps(p8, k_vec_8, sum_vec);

        sum_vec = _mm256_mul_ps(sum_vec, k_vec_norm);
        _mm256_storeu_ps(&p_dst[x], sum_vec); // Using unaligned store
    }
```

در نتیجه بجای انجام 9 عملیات برای هر پیکسل، ما 9 عملیات برداری انجام می دهیم تا 8 پیکسل را محاسبه کنیم.

- FMA: Fused Multiply-Add

بجای استفاده از ترکیب add , mult برای محاسبه کانولوشن، از fmadd استفاده شده که عملیات ضرب و جمع را با یکدیگر ترکیب کند.

- بهینه سازی نرمالایز:

بجای استفاده از تقسیم در بخش نرمالایز که اورهد زیادی دارد، از ضرب استفاده شده.

- Remainder loop

محاسبه پیکسل های باقی مانده mod 8

```
// Remainder loop
for (; x < cols - 1; ++x) {
    float sum = 0.0f;
    sum += p_top[x - 1] * kernel[0];
    sum += p_top[x]     * kernel[1];
    sum += p_top[x + 1] * kernel[2];
    sum += p_mid[x - 1] * kernel[3];
    sum += p_mid[x]     * kernel[4];
    sum += p_mid[x + 1] * kernel[5];
    sum += p_bot[x - 1] * kernel[6];
    sum += p_bot[x]     * kernel[7];
    sum += p_bot[x + 1] * kernel[8];
    p_dst[x] = sum * norm_factor;
}
```

b. موازی سازی sobel_magnitude_fused_serial

مانند موازی سازی convolve_serial همان ایده در این بخش نیز پیاده سازی شده.

- ست کردن کرنل های sobel x,y

```
const __m256 sx_vec_0 = _mm256_set1_ps(sobel_x_kernel[0]);
const __m256 sx_vec_2 = _mm256_set1_ps(sobel_x_kernel[2]);
const __m256 sx_vec_3 = _mm256_set1_ps(sobel_x_kernel[3]);
const __m256 sx_vec_5 = _mm256_set1_ps(sobel_x_kernel[5]);
const __m256 sx_vec_6 = _mm256_set1_ps(sobel_x_kernel[6]);
const __m256 sx_vec_8 = _mm256_set1_ps(sobel_x_kernel[8]);
const __m256 sy_vec_0 = _mm256_set1_ps(sobel_y_kernel[0]);
const __m256 sy_vec_1 = _mm256_set1_ps(sobel_y_kernel[1]);
const __m256 sy_vec_2 = _mm256_set1_ps(sobel_y_kernel[2]);
const __m256 sy_vec_6 = _mm256_set1_ps(sobel_y_kernel[6]);
const __m256 sy_vec_7 = _mm256_set1_ps(sobel_y_kernel[7]);
const __m256 sy_vec_8 = _mm256_set1_ps(sobel_y_kernel[8]);
```

- FMA ₉ Vectorization

```
// [MODIFIED] y-loop now runs within the tile range
for (int y = y_start; y < y_end; ++y) {
    const float* p_top = blurred.ptr<float>(y - 1);
    const float* p_mid = blurred.ptr<float>(y);
    const float* p_bot = blurred.ptr<float>(y + 1);
    float* p_mag = magnitude.ptr<float>(y);

    int x = 1;

    for (; x <= limit; x += 8) {
        const __m256 p0 = _mm256_loadu_ps(&p_top[x - 1]);
        const __m256 p1 = _mm256_loadu_ps(&p_top[x]);
        const __m256 p2 = _mm256_loadu_ps(&p_top[x + 1]);
        const __m256 p3 = _mm256_loadu_ps(&p_mid[x - 1]);
        // const __m256 p4 = _mm256_loadu_ps(&p_mid[x]);
        const __m256 p5 = _mm256_loadu_ps(&p_mid[x + 1]);
        const __m256 p6 = _mm256_loadu_ps(&p_bot[x - 1]);
        const __m256 p7 = _mm256_loadu_ps(&p_bot[x]);
        const __m256 p8 = _mm256_loadu_ps(&p_bot[x + 1]);

        __m256 gx_vec = _mm256_setzero_ps();
        gx_vec = _mm256_fmadd_ps(p0, sx_vec_0, gx_vec);
        gx_vec = _mm256_fmadd_ps(p2, sx_vec_2, gx_vec);
        gx_vec = _mm256_fmadd_ps(p3, sx_vec_3, gx_vec);
        gx_vec = _mm256_fmadd_ps(p5, sx_vec_5, gx_vec);
        gx_vec = _mm256_fmadd_ps(p6, sx_vec_6, gx_vec);
        gx_vec = _mm256_fmadd_ps(p8, sx_vec_8, gx_vec);

        __m256 gy_vec = _mm256_setzero_ps();
        gy_vec = _mm256_fmadd_ps(p0, sy_vec_0, gy_vec);
        gy_vec = _mm256_fmadd_ps(p1, sy_vec_1, gy_vec);
        gy_vec = _mm256_fmadd_ps(p2, sy_vec_2, gy_vec);
        gy_vec = _mm256_fmadd_ps(p6, sy_vec_6, gy_vec);
        gy_vec = _mm256_fmadd_ps(p7, sy_vec_7, gy_vec);
        gy_vec = _mm256_fmadd_ps(p8, sy_vec_8, gy_vec);

        __m256 gx_sq = _mm256_mul_ps(gx_vec, gx_vec);
        __m256 gy_sq = _mm256_mul_ps(gy_vec, gy_vec);
        __m256 sum_sq = _mm256_add_ps(gx_sq, gy_sq);
        __m256 mag_vec = _mm256_sqrt_ps(sum_sq);

        // [BUG FIX] Using storeu instead of store
        _mm256_storeu_ps(&p_mag[x], mag_vec);
    }
```

- Remainder Loop

```
// Remainder loop
for (; x < cols - 1; ++x) {
    float gx = (p_top[x - 1] * sobel_x_kernel[0]) + (p_top[x + 1] * sobel_x_kernel[2]) +
               (p_mid[x - 1] * sobel_x_kernel[3]) + (p_mid[x + 1] * sobel_x_kernel[5]) +
               (p_bot[x - 1] * sobel_x_kernel[6]) + (p_bot[x + 1] * sobel_x_kernel[8]);

    float gy = (p_top[x - 1] * sobel_y_kernel[0]) + (p_top[x] * sobel_y_kernel[1]) + (p_top[x + 1] * sobel_y_kernel[2]) +
               (p_bot[x - 1] * sobel_y_kernel[6]) + (p_bot[x] * sobel_y_kernel[7]) + (p_bot[x + 1] * sobel_y_kernel[8]);

    p_mag[x] = sqrt(gx * gx + gy * gy);
}
```

Memory Bound .c

مشکل اصلی این بود که تابع convolve کل تصویر را از Ram میخواند و کل تصویر را پس از محاسبات دوباره در Ram می نوشت، و بلافاصله پس از آن تابع sobel خروجی تصویر قبل را از Ram میخواند و دوباره در Ram می نوشت. در حالی که کش بلا استفاده باقی میماند.

- Tiling | Cache Blocking

  ایده کلی این است که بجای پردازش مرحله به مرحله، پردازش به صورت ناحیه به ناحیه انجام شود.

  مزیت:

  زمانی که تابع convolve عملیاتش را برای یک Tile تمام میکند، نتایج آن همچنان درون کش لایه سوم L3 موجود هستند. در نتیجه اجرای تابع sobel بلافاصله روی همان Tile داده ها مستقیم از کش load میشوند که باعث میشود گلوگاه حافظه کنترل شود.

  در نتیجه ساختار دو تابع convolve , sobel به شکلی تغییر کرد تا بتوانند تنها یک نوار را محاسبه کنند.

```
void convolve_parallel_tiled(const Mat& src, Mat& dst, const float* kernel, float norm_factor, int y_start, int y_end) {
```

```
void sobel_magnitude_fused_parallel_tiled(const Mat& blurred, Mat& magnitude, int y_start, int y_end) {
```

پیاده سازی OpenMp

```
const int TILE_H = 128; // based on cache line size

#pragma omp for schedule(static)
for (int y_tile = 0; y_tile < rows; y_tile += TILE_H) {

    int sob_start = max(1, y_tile);
    int sob_end   = min(rows - 1, y_tile + TILE_H);

    if (sob_start >= sob_end) continue;

    int gauss_start = max(1, sob_start - 1);
    int gauss_end   = min(rows - 1, sob_end + 1);

    for (int y = gauss_start; y < gauss_end; ++y) {
        const float* p_top = src.ptr<float>(y - 1);
        const float* p_mid = src.ptr<float>(y);
        const float* p_bot = src.ptr<float>(y + 1);
        float* p_dst = blurred.ptr<float>(y);

        // apply gaussian filter
        for (int x = 1; x < cols - 1; ++x) {…
    }

    for (int y = sob_start; y < sob_end; ++y) {
        const float* p_top = blurred.ptr<float>(y - 1);
        const float* p_mid = blurred.ptr<float>(y);
        const float* p_bot = blurred.ptr<float>(y + 1);
        float* p_mag = magnitude.ptr<float>(y);

        // apply sobel filter
        for (int x = 1; x < cols - 1; ++x) {…
    }
}
```

برای پیاده سازی این بخش کل پردازش با استفاده از OpenMP به نوارهای افقی مستقل تقسیم شد. ارتفاع این نوارها با استفاده از پارامتر TILE_H (تنظیم شده روی ۱۲۸ سطر) متناسب با ظرفیت کش پردازنده تعیین شده. که باعث می شود داده‌های تولید شده در مرحله اول، تا زمان مصرف در مرحله دوم، درون حافظه کش باقی مانده و در مرحله بعد سریعتر مصرف شوند به عبارت دیگر بیشتر cache hit رخ دهد.

در این روش، هر ترد علاوه بر نوار اختصاصی خود، سطرهای مرزی مورد نیاز را نیز به صورت محلی محاسبه می‌کند تا نیازی به ارتباط ترد ها برای بدست آوردن مقدار آنها نباشد اگرچه این کار منجر به اندکی محاسبات تکراری می‌شود، اما با حذف کامل نیاز به مکانیزم‌های همگام‌سازی بین هسته‌ها در نهایت منجر به افزایش کارایی می شود.

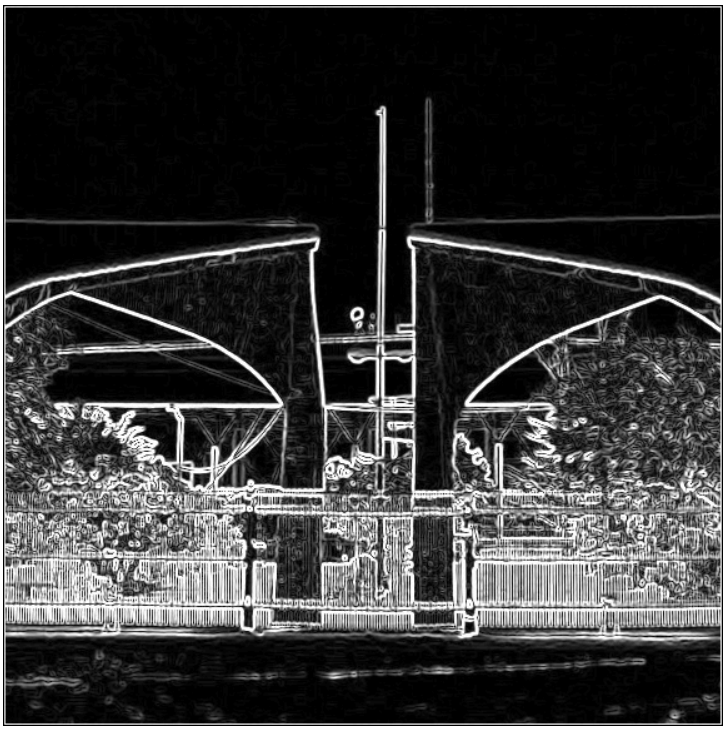
درنهایت برای موازی‌سازی عملیات محاسباتی هر نوار ، از استراتژی زمان‌بندی schedule static استفاده شده زیرا حجم محاسبات ریاضی (تعداد عملیات ضرب و جمع) برای تمامی پیکسل‌ها دقیقاً یکسان و ثابت است. این یکنواختی باعث می‌شود که بار پردازشی تمام نوارها برابر باشد؛ لذا استفاده از روش static علاوه بر توزیع متوازن کار بین هسته‌ها، سربار مدیریتی (Overhead) ناشی از درخواست مکرر کار و هماهنگی بین تردها را حذف می کند. (البته تفاوت قابل توجهی بین دو حالت static و dynamic در عمل برای این مسئله وجود ندارد)

پیاده سازی OpenMp + SIMD

```
void openmp_simd_fused_pipeline(const Mat& src, Mat& blurred, Mat& magnitude,
                                const float* g_kernel, float g_norm) {
    int rows = src.rows;
    int cols = src.cols;

    blurred.create(rows, cols, CV_32F);
    magnitude.create(rows, cols, CV_32F);

    const int TILE_H = 64;

    #pragma omp parallel for schedule(static)
    for (int y_tile = 0; y_tile < rows; y_tile += TILE_H) {

        int sob_start = max(1, y_tile);
        int sob_end   = min(rows - 1, y_tile + TILE_H);

        if (sob_start >= sob_end) continue;

        int gauss_start = max(1, sob_start - 1);
        int gauss_end   = min(rows - 1, sob_end + 1);


        if (gauss_start < gauss_end) {
            convolve_parallel_tiled(src, blurred, g_kernel, g_norm, gauss_start, gauss_end);
        }

        sobel_magnitude_fused_parallel_tiled(blurred, magnitude, sob_start, sob_end);
    }
}
```

در این تابع، تکنیک‌های OpenMP و SIMD با یکدیگر ترکیب شدند. حلقه اصلی با تقسیم تصویر به نوارهای ۶۴ سطری و استفاده از زمان‌بندی `static`، داده‌های میانی را در کش نگه می دارد و ترافیک حافظه اصلی را به حداقل می‌رساند. در لایه درونی، با فراخوانی توابع برداری AVX به جای حلقه‌های اسکالر، پردازش همزمان ۸ پیکسل در هر سیکل انجام می شود که پیاده سازی آن در بخش قبل آمده است.

تحلیل خروجی و SpeedUp



```
--- Results ---
Serial Time:   7432594 clocks
SIMD Time: 2739982 clocks
OpenMP Time:   3183698 clocks
OpenMP+SIMD Time:   768618 clocks
Speedup (SIMD):       2.71x
Speedup (OMP):     2.33x
Speedup (OMP+SIMD):     9.67x
```

در نسخه OpenMP، دستیابی به تسریع ۲.۳۳ برابری نشان می‌دهد که اگرچه توزیع بار میان هسته‌ها انجام شده، اما الگوریتم همچنان با گلوگاه پهنای باند حافظه درگیر است؛ بدین معنا که سرعت فراخوانی داده‌ها از حافظه اصلی کمتر از سرعت پردازش هسته‌ها بوده و بخش قابل‌توجهی از زمان اجرا صرف انتظار شده است.

اما در نسخه ترکیبی یک جهش عملکردی با تسریع ۹.۶۷ برابری رخ داده. این نتیجه حاصل هم‌افزایی دو سطح موازی‌سازی و مدیریت حافظه است: OpenMP با تقسیم تصویر به نوارهای متناسب با کش ، ترافیک حافظه اصلی را حذف کرده و SIMD درون هر ترد در هر گام ۸ پیکسل را پردازش می کند. این ترکیب باعث شده تا داده‌های بارگذاری شده در کش با حداکثر کارایی ممکن پردازش شوند.

Network

استراتژی پیاده سازی کاملا یکسان و بدون تغییر از پروژه قبل باقی مانده.
از آنجایی که محاسبه هر نود مستقل از دیگریست، به راحتی میتوان حلقه ها را با استفاده از openMP موازی کرد.
تنها نکته مهم که در پروژه قبل نیز گفته شد، Memory Bound بودن ذاتی این مسئله است، به این معنی که حتی با موازی سازی به وسیله openMP و استفاده از دستورات بردار SIMD، همچنان گلوگاه اصلی سیستم، پهنای باند حافظه (Memory Bandwidth) باقی می ماند.
چرا که در ای مسئله شدت محاسباتی به شدت پایین است، یعنی پردازنده سریع تر از آنکه مموری داده ها را تامین کند محاسبات را انجام میدهد، در نتیجه با افزایش تعداد هسته ها، Speedup به صورت چشمگیر افزایش پیدا نمیکند.

- Serial + openMP

  از آنجایی که شدت محاسبات همگن و به شکل بالانس توزیع شده اند، میتوان scheduling را به شکل static انجام داد.

```
void forward_layer_openmp(float* output_vector, const float* input_vector,
                          const float* weights_transposed, const float* bias_vector,
                          int num_inputs, int num_outputs) {
    int j;
    #pragma omp parallel for schedule(static) private(j) \
            shared(output_vector, input_vector, weights_transposed, bias_vector, num_inputs, num_outputs)
    for (j = 0; j < num_outputs; ++j) {
        float sum = 0.0f;
        const float* p_weight_row = weights_transposed + (j * num_inputs);
        for (int i = 0; i < num_inputs; ++i) {
            sum += input_vector[i] * p_weight_row[i];
        }
        output_vector[j] = activation_relu(sum + bias_vector[j]);
    }
}
```

- SIMD + openMP

```
void forward_layer_omp_simd(float* output_vector, const float* input_vector,
                            const float* weights_vsimd, const float* bias_vector,
                            int num_inputs, int num_outputs) {
    int j;
    #pragma omp parallel for schedule(static) private(j) \
            shared(output_vector, input_vector, weights_vsimd, bias_vector, num_inputs, num_outputs)
    for (j = 0; j < num_outputs; j += AVX_LANE_COUNT) {
        __m256 v_sum = _mm256_load_ps(bias_vector + j);
        const float* p_weight_chunk = weights_vsimd + (j / AVX_LANE_COUNT) * (num_inputs * AVX_LANE_COUNT);

        for (int i = 0; i < num_inputs; ++i) {
            __m256 v_input = _mm256_set1_ps(input_vector[i]);
            __m256 v_weights = _mm256_load_ps(p_weight_chunk + (i * AVX_LANE_COUNT));
            v_sum = _mm256_fmadd_ps(v_input, v_weights, v_sum);
        }
        v_sum = _mm256_relu_ps(v_sum);
        _mm256_store_ps(output_vector + j, v_sum);
    }
}
```

- SpeedUp

حالت بررسی شده برای تعداد نود های بالاتر استفاده شده، تا اورهد مدیریت ترد ها نادیده گرفته شود

```
$ ./network.out
Network: 4096 -> 8192 -> 4096
Threads: 12

Serial Clocks:       109176112
SIMD Clocks:         41318344
OpenMP Clocks:       37413415
OpenMP+SIMD Clocks:  28024716

--------------------------------
Speedup (SIMD vs Serial):        2.64x
Speedup (OpenMP vs Serial):      2.92x
Speedup (OpenMP+SIMD vs Serial): 3.90x
--------------------------------
Verify SIMD           : PASSED (Max Diff: 3.4e-03)
Verify OpenMP         : PASSED (Max Diff: 0.0e+00)
Verify OpenMP+SIMD    : PASSED (Max Diff: 3.4e-03)
```

Julia Set

هدف این بخش، پیاده سازی چندگانه جولیا ست با استفاده از محاسبات بر روی اعداد مختلط است.
الگوریتم این مسئله شامل محاسبه یک رابطه بازگشتی برای هر پیکسل و تشخیص همگرایی یا واگرایی آن خواهد بود. که نکته اصلی آن مستقل بودن محاسبات برای هر پیکسل از پیکسل دیگر است.

$$Z_{new} = Z^N + C$$

1. نسخه سریال

نسخه سریال شامل پیاده سازی الگوریتم به همراه کمی بهینه سازی اولیه خواهد بود.
برای حفظ کش محاسبات به صورت سطری انجام خواهند شد.

- نگاشت: هر پیکسل در مختصات صحیح (x, y) را به یک مختصات اعشاری (Real, Imag) در فضای اعداد مختلط تبدیل میکنیم.

```
double imag = IMAG_MIN + (double)y / HEIGHT * (IMAG_MAX - IMAG_MIN);
double real = REAL_MIN + (double)x / WIDTH * (REAL_MAX - REAL_MIN);
```

و سپس آنها را درون کلاس complex برای سادگی محاسبات ذخیره میکنیم.

- محاسبات: برای هر پیکسل رابطه بازگشتی را اعمال می کنیم، و در هر مرحله شعاع قابل قبول Z را بررسی میکنیم تا در صورت واگرایی از حلقه تکرار خارج شویم.

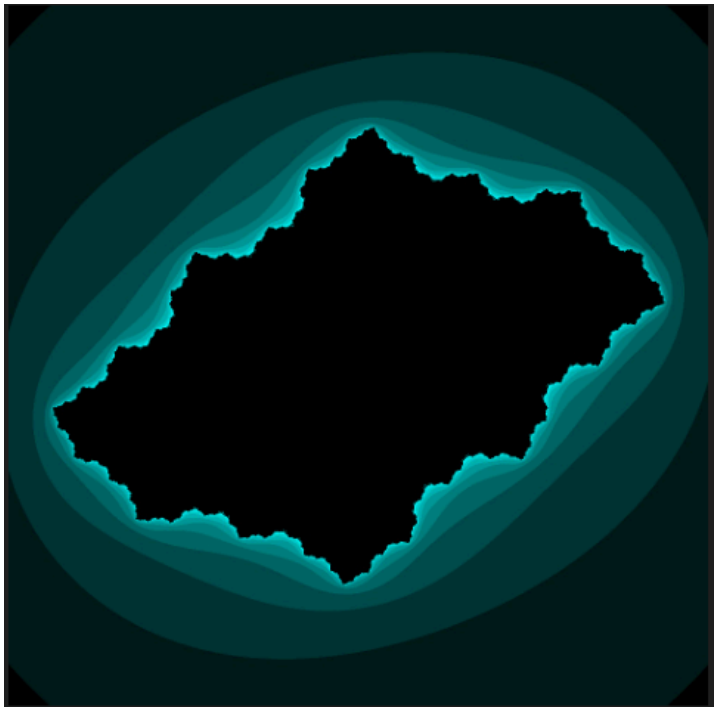
```
// --- Step 2: Iterate the formula Z = Z^n + C ---
int iter = 0;
while (iter < MAX_ITER) {
    // Optimization: Check squared norm to avoid sqrt()
    // std::norm(z) returns real^2 + imag^2
    if (std::norm(z) > ESCAPE_RADIUS_SQ) {
        break; // Point escaped (Diverged)
    }

    // Apply the formula: Z_new = Z^n + C
    z = std::pow(z, POWER_N) + C;

    iter++;
}
```

- رنگ آمیزی: پیکسل های همگرا سیاه، و پیکسل های واگرا بر اساس میزان واگرایی به یک طیف رنگی مپ میشوند.

```
// --- Step 3: Color the pixel based on iterations ---
if (iter == MAX_ITER) {
    // Point is convergent (part of the set) -> Color Black
    row_ptr[x] = Vec3b(0, 0, 0);
} else {
    // I just changed the constants for a better visual effect :)
    unsigned char color_val = (unsigned char)(255.0 * iter / 10);
    // BGR format in OpenCV
    // Just a simple coloring scheme (Blueish to White)
    row_ptr[x] = Vec3b(color_val, color_val, 0);
}
```

- خروجی: برای مقدار $C(-0.355,\ 0.355)$

```
Running Serial...
Serial Time:    3431736684 clocks.
Serial Time:    1.314 seconds.
```



2. نسخه openMP

هدف این بخش، کاهش زمان اجرا با استفاده از تمام هسته های پردازنده میباشد.

- پیاده سازی:

از آنجایی که در این الگوریتم هیچگونه وابستگی بین دیتا ها وجود ندارد هر ترد میتواند بصورت کاملا مستقل پیکس مربوط به خودش را محاسبه کند.

ایده اصلی این موازی سازی این بخش، موازی سازی بر اساس هر سطر است. به شکلی که هر ترد وظیفه محاسبه یک الی چند سطر را بر عهده بگیرد.

```
#pragma omp parallel for schedule(dynamic, CHUNK_SIZE) shared(image, dx, dy) private(y, imag, real)
for (y = 0; y < HEIGHT; ++y) {
```

چالش مهم این بخش همگن نبودن محاسبات برای هر سر میباشد، به عنوان مثال در حالت نمونه ما برای سریال برخی سطر ها حاوی نقاط همگرایی بیشتری هستند که در نتیجه آن تکرار رابطه بازگشتی تا MAX_ITER برای آن پیکسل ادامه داشته است. در حالی که برخی سطر های دیگر حتی دارای یک نقطه همگرایی نیز نیستند و رابطه بازگشتی در ایتریشن به عنوان مثال 10 ام به اتمام میرسد.

به همین جهت برای تعادل میان بار ترد ها، اسکجولینگ به شکل داینامیک انجام میشود تا هر ترد پس از اتمام چانک خود یک چانک جدید بگیرد تا عدم تعادل بار مدیریت شود.

- نتیجه:

```
Running Parallel (OpenMP)...
Parallel Time: 447011449 clocks.
Parallel Time: 0.171 seconds.
```

3. نسخه openMP + SIMD

در این بخش بدون تغییر در ایده بخش openMP قصد دارم تا با استفاده از پردازش برداری SIMD برنامه را بهینه کنیم.

ایده اصلی این است که هر ترد در لحظه به جای محاسبه یک پیکسل بتواند همزمان 8 پیکسل را بطور همزمان محاسبه کند. اما از آنجایی که میدانیم محاسبات برای هر پیکسل لزوما یکسان نیستند و ممکن است برخی در تکرار های کم واگرا شوند و برخی تا انتها MAX_ITER همگرا بمانند. اما نمی توان محاسبات را متوقف کرد تا زمانی که هر 8 پیکسل به نتیجه نرسیده باشند.

- تکنیک Masking

برای حل این مشکل از این تکنیک استفاده شده، به شکلی که میتوان گفت هر پیکسل یک فلگ برای خودش دارد که نشان میدهد محاسبات آن هنوز در حال جریان است یا خیر. که در صورت صفر بودن تمامی مقادیر masking، حلقه محاسبه بخش بازگشتی break میشود.

همچنین نیاز است که برای هر پیکسل یک شمارنده نگه داریم، تا پس از اتمام ایتریشن آن مقدار آن ایتریشن را بدانیم.

- ذخیره سازی:

برای آنکه بتوانیم از ابزار SIMD استفاده کنیم مختصات حقیقی را پس از تبدیل به مختصات مختلط درون دو رجیستر جدا برای بخش حقیقی و موهومی ذخیره میکنیم.

- محاسبه ایتریشن بعدی:

$$Z_{new} = Z^N + C$$

با استفاده از ابزار SIMD این رابطه را پیاده سازی میکنیم.

```
// Loop to multiply Z by itself (POWER_N - 1) times
// Example: If N=2, loop runs once: Z * Z
// Example: If N=3, loop runs twice: (Z * Z) * Z
for (int p = 1; p < POWER_N; ++p) {
    // Complex Multiplication: (res * base)
    // Real: (res_re * base_re) - (res_im * base_im)
    // Imag: (res_re * base_im) + (res_im * base_re)

    __m256 v_ac = _mm256_mul_ps(v_res_re, v_base_re);
    __m256 v_bd = _mm256_mul_ps(v_res_im, v_base_im);
    __m256 v_ad = _mm256_mul_ps(v_res_re, v_base_im);
    __m256 v_bc = _mm256_mul_ps(v_res_im, v_base_re);

    v_res_re = _mm256_sub_ps(v_ac, v_bd);
    v_res_im = _mm256_add_ps(v_ad, v_bc);
}

// Finally add constant C
v_z_re = _mm256_add_ps(v_res_re, v_c_re);
v_z_im = _mm256_add_ps(v_res_im, v_c_im);
```

- بررسی شرط همگرایی:

برای هر 8 پیکسل مقدار نورم z را پیدا میکنیم.

```
// Calculate |Z|^2 = re^2 + im^2
__m256 v_re2 = _mm256_mul_ps(v_z_re, v_z_re);
__m256 v_im2 = _mm256_mul_ps(v_z_im, v_z_im);
__m256 v_norm2 = _mm256_add_ps(v_re2, v_im2);
```

سپس نورم هر یک را با حد قابل قبول برای واگرایی مقایسه میکنیم.

```
// Check escape: norm2 < 4.0 ?
// Returns 0xFFFFFFFF (-1) if True (inside), 0 if False (escaped)
__m256 v_inside = _mm256_cmp_ps(v_norm2, v_escape, _CMP_LT_OQ);
```

و در نهایت از تکنیک masking برای بررسی پایان ایتریشن برای هر 8 پیکسل استفاده میکنیم. (مقدار اولیه v_mask: یک میباشد)

```
v_mask = _mm256_and_ps(v_mask, v_inside);

// If mask is all zeros, everyone has escaped. Break early!
int mask_bits = _mm256_movemask_ps(v_mask);
if (mask_bits == 0) {
    break;
}
```

- آپدیت تعداد تکرار ها:

همانطور که در بخش Masking گفته شد، ما نیاز داریم تا تعداد تکرار هار قبل از خروج ذخیره کنیم، پس از یک counter استفاده میکنیم که تنها زمانی مقدار آن اضافه می شود که مقدار mask برای آن پیکسل True باشد.

از آنجایی که مقدار باینری True برابر -1 است از تفریق mask در iters قبل برای بروزرسانی تعداد تکرار ها استفاده میکنیم.

$$iters = iters - mask$$

```
v_iters = _mm256_sub_epi32(v_iters, _mm256_castps_si256(v_mask));
```

- رنگ آمیزی:

پس از اتمام حلقه و استخراج تعداد ایتریشن ها درون یک آرایه، مانند نسخه های قبل رنگ آمیزی انجام میشود.

- نتیجه:

```
Running Parallel (OpenMP + AVX)...
AVX Time: 43923291 clocks.
AVX Time: 0.016 seconds.
```

4. Speedup

اسپید آپ نهایی برای هر سه حالت نسبت به یکدیگر بر اساس تعداد clock ها محاسبه شده و به شرح زیر است.

```
Speedup openMP (vs Serial): 6.05x
Speedup openMP + SIMD (vs Serial): 79.60x
Speedup openMP + SIMD (vs openMP): 13.17x
```