به نام خدا

گزارش تمرین کامپیوتری
شماره دو
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.

Serial mean calculation:

```
// --- Serial Pass 1: Calculate Mean ---
sum_parts[0] = 0.0f; sum_parts[1] = 0.0f; sum_parts[2] = 0.0f; sum_parts[3] = 0.0f;
for (i = 0; i < limit ; i+=4){
    sum_parts[0] += data[i];
    sum_parts[1] += data[i+1];
    sum_parts[2] += data[i+2];
    sum_parts[3] += data[i+3];
}
total_sum = sum_parts[0] + sum_parts[1] + sum_parts[2] + sum_parts[3];

for (i = limit; i < ARRAY_SIZE; ++i) { total_sum += data[i]; } // add remaining elements
serial_mean = total_sum / ARRAY_SIZE;
```

Serial variance calculation:

```
// --- Serial Pass 2: Calculate Sum of Squared Differences ---
sum_parts[0] = 0.0f; sum_parts[1] = 0.0f; sum_parts[2] = 0.0f; sum_parts[3] = 0.0f;
for(i = 0 ; i < limit ; i+=4){
    float diff0 = data[i] - serial_mean;
    float diff1 = data[i+1] - serial_mean;
    float diff2 = data[i+2] - serial_mean;
    float diff3 = data[i+3] - serial_mean;
    sum_parts[0] += diff0 * diff0;
    sum_parts[1] += diff1 * diff1;
    sum_parts[2] += diff2 * diff2;
    sum_parts[3] += diff3 * diff3;
}
total_sq_diff = sum_parts[0] + sum_parts[1] + sum_parts[2] + sum_parts[3];

// Start remainder loop from 'limit'
for (i = limit; i < ARRAY_SIZE; ++i) {
    float diff = data[i] - serial_mean;
    total_sq_diff += diff * diff;
}
serial_variance = total_sq_diff / ARRAY_SIZE;
```

در هر دو حلقه پردازشی از متغیر های جمع کننده مستقل استفاده شده و همچنین محاسبه آنها به صورت پیمانه ای(مانند نسخه موازی) انجام شده تا حداکثر بهینگی و شباهت را به نسخه موازی داشته باشد

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.

Parallel mean calculation:

```
// --- SIMD Pass 1: Calculate Mean ---
sum_vec = _mm_setzero_ps(); // make a pack of 4 single-precision values by setting each element to 0

for (i = 0; i < limit; i += 4) {
    __m128 data_vec = _mm_load_ps(data + i); // load 4 single-precision values from ALIGNED memory
    sum_vec = _mm_add_ps(sum_vec, data_vec); // add 4 single-precision values
}
sums = _mm_hadd_ps(sum_vec, sum_vec); // add pairs of single-precision values
sums = _mm_hadd_ps(sums, sums);
total_sum_simd = (double)_mm_cvtss_f32(sums); // (Cast to double for precision)

for (i = limit; i < ARRAY_SIZE; ++i) { total_sum_simd += data[i]; } // add remaining elements
mean_simd = static_cast<float>(total_sum_simd / ARRAY_SIZE);
```

در این نسخه جمع داده های پیمانه ای با یک دستور انجام شده و لود کردن آنها نیز با یک دستور در حالی که در نسخه سریال به ازای هر جمع پیمانه ای دو دستور استفاده می شد یک دستور برای لود کردن آن مقدار در رجیستر و یک دستور برای محاسبه جمع برای آن که مجموع می شود ۸ دستور برای یک iteration از حلقه. می توان نتیجه گرفت که ۲ دستور در حالت SIMD بهینه تر از ۸ دستور در حالت سریال خواهد بود

Parallel variance calculation:

```
// --- SIMD Pass 2: Calculate Sum of Squared Differences ---
mean_vec = _mm_set1_ps(mean_simd); // Use the mean calculated by SIMD
sum_vec = _mm_setzero_ps();

for(i = 0 ; i < limit; i += 4){
    __m128 data_vec = _mm_load_ps(data + i); // Use ALIGNED load
    __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);
total_sq_diff_simd = (double)_mm_cvtss_f32(sums); // (Cast to double for precision)

for (i = limit; i < ARRAY_SIZE; ++i) { // add remaining elements
    float diff = data[i] - mean_simd;
    total_sq_diff_simd += diff * diff;
}
parallel_variance = static_cast<float>(total_sq_diff_simd / ARRAY_SIZE);
```

برای محاسبه واریانس در حالت SIMD برای هر iteration چهار دستور استفاده شده یک دستور برای لود داده ها, یک دستور برای تفریق ۴ داده و میانگین محاسبه شده در بخش قبل, یک دستور برای ضرب اختلاف ها و یک دستور برای جمع با مقادیر قبلی یعنی در مجموع ۴ دستور برای یک iteration اما در حالت سریال ۴ بارگذاری داده, ۴ تفریق, ۴ ضرب و ۴ جمع انجام شده که در مجموع می شود ۱۶ دستور برای یک iteration باز هم می توان بهینگی ۴ دستور SIMD را در برابر ۱۶ دستور سریال نتیجه گرفت

تحلیل خروجی و Speed Up

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

Serial Variance: 0.0830927789 clock cycles: 47775216
Parallel Variance: 0.0830927789 clock cycles: 36486994
Speedup: 1.3094x
```

توقع می رفت که مقدار speedUp به دلیل نسبت دستورات حالت سریال و موازی که برابر ۴ بود در حدود ۴ برابر باشد اما خروجی نشان می دهد که تفاوت زیادی بین زمان دو نسخه وجود ندارد و علت اصلی آن همانطور که اشاره شد این است که bottleneck اصلی برنامه دسترسی به حافظه است و هر دو نسخه باید دوبار منتظر رسیدن داده ها از رم (به cache) بمانند و بخش بزرگی از زمان اجرا در

هر دو مشترک خواهد بود با این حال بهینه سازی در سطح دستورات, نسخه SIMD را حدود 30 درصد کارآمد تر کرده است

افزودن واترمارک

```
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

Serial initialization:

```
const int height = base_img.rows;
const int width = base_img.cols;
const float total_dim_inv = 1.0f / (float)(width + height);

const float c = total_dim_inv;
const float x_indices[8] = {0.0f, 1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f};
const float x_step = 8.0f * c;
const float one = 1.0f;

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;
```

برای بهینه‌سازی حداکثری و کاهش حجم محاسبات، مقادیر ثابت از حلقه بیرونی (y) خارج شدند. این مقادیر شامل بردار اندیس‌های اولیه (x_indices) (برای محاسبه آلفای ۸ پیکسل اول هر سطر) و بردار گام (x_step) (که آلفای فعلی را برای ۸ پیکسل بعدی آپدیت می‌کند) و متغیر های دیگر می‌باشد همچنین محاسبه تقسیم float که یک عملیات پر هزینه است یکبار در این بخش انجام می شود

Serial outer loop:

```
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);

    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;
```

از آنجایی که مقدار آلفا شامل جمع دو متغیر وابسته x و y است (a=x/C+y/C)، مقدار ثابتِ وابسته به y (یعنی y/C) به حلقه بیرونی منتقل شده که باعث می‌شود محاسبات تکراری از از حلقه داخلی حذف شوند. همچنین، برای دسترسی به کانال‌های تخصیص‌یافته در RAM، از اشاره‌گرهای خام (مانند base_ptr) استفاده شده که دسترسی به داده‌هایی که از قبل در کش (Cache) قرار دارند را بسیار سریع می‌کند

Serial inner loop:

```
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;
}
```

مقادیر کانال خروجی برای هم ۸ پیکسل انجام می شود و سپس به unsigned char هشت بیتی کست می شود تا آن را در کانال خروجی ذخیره شود. سپس مقدار آلفا برای ۸ پیکسل بعدی را با جمع کردن آنها با مقدار ثابت (step = 8/w+h) , آپدیت می شود.

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)
}
```

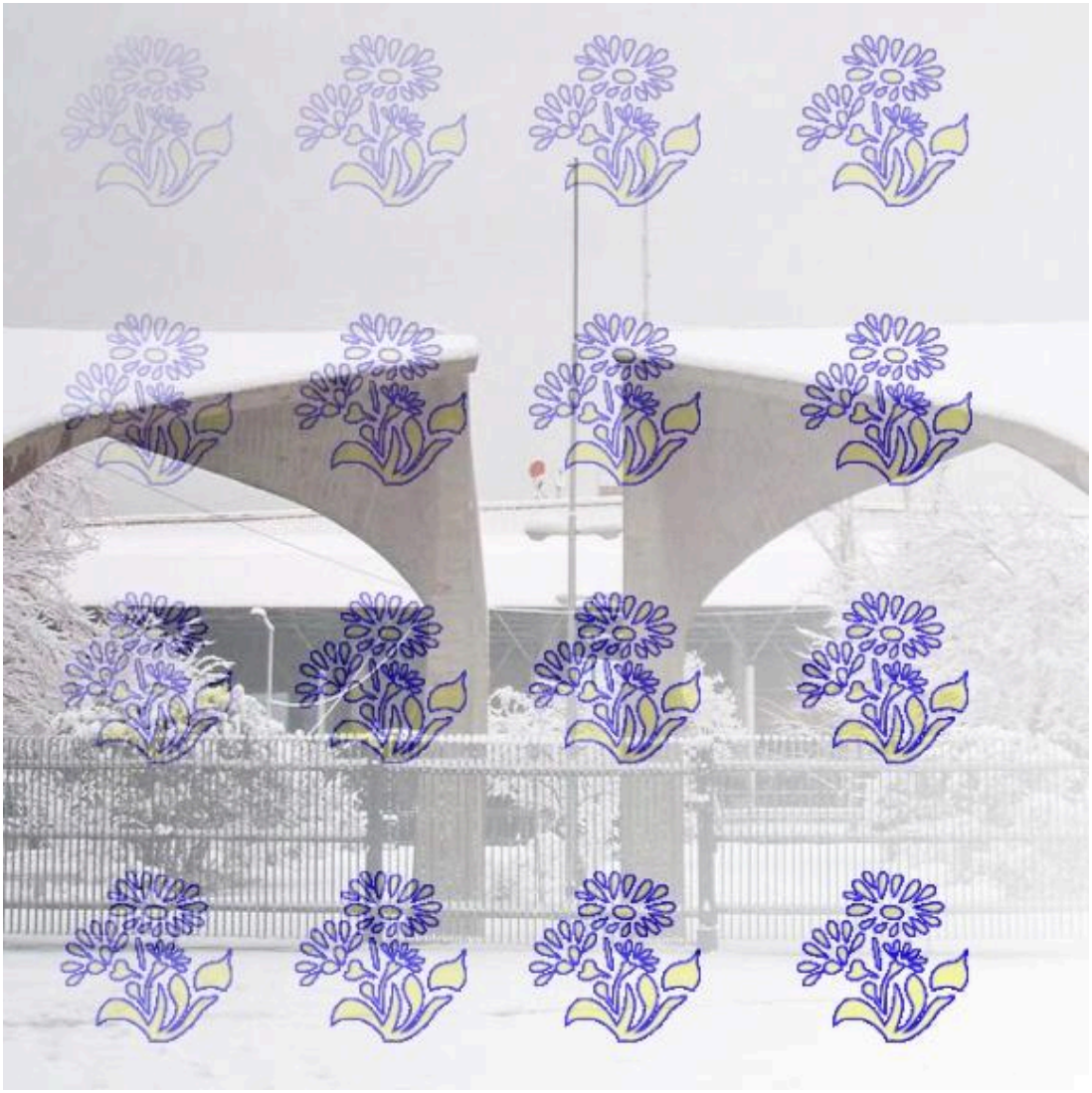
در حلقه داخلی، ۸ پیکسل با دستور _mm_loadl_epi64_ بارگذاری شده و سپس با دستورات AVX2 (مانند _mm256_cvtepu8_epi32_ و _mm256_cvtepi32_ps_) به ۸ عدد ۳۲ بیتی float تبدیل می‌شوند. سپس تمام محاسبات ترکیب (Blend) (شامل _mm256_mul_ps_ و _mm256_add_ps_) به صورت همزمان روی این ۸ پیکسل انجام می‌گیرد. نتایج float 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.

تحلیل خروجی و SpeedUp



```
--- Watermark Blending (Optimized Serial vs AVX) ---
Serial (Optimized) clock cycles: 3168764
Parallel (AVX) clock cycles:     561786
Speedup: 5.6405x

Output images saved.
```

گلوگاه اصلی (Bottleneck) در این بخش حجم محاسبات (Compute-Bound) است. در نسخه سریال بهینه، به ازای هر ۸ پیکسل ، پردازنده باید زنجیره بلندی از محاسبات سنگین ممیز شناور (FP) و تبدیل نوع داده را اجرا کند. در مقابل، نسخه AVX بسیاری از این عملیات را در دستورالعمل‌های واحد تجمیع می‌کند.

۱. نسخه سریال

- فرمول: Out = α*W + (1-α)*B
- به ازای هر پیکسل (۱ پیکسل):
  - عملیات FP (ممیز شناور): ۵ دستور
    عملیات حافظه و تبدیل: ۳ دستور
- مجموع برای ۸ پیکسل (در یک تکرار حلقه x):
  - عملیات FP: 8 × 5 = 40
  - عملیات حافظه و تبدیل: 8 × 3 = 24
  - مجموع تخمینی دستورات اسکالر: ~۶۴ دستور

۲. نسخه موازی

- به ازای هر ۸ پیکسل (در یک تکرار حلقه x):
  - عملیات FP (برداری): ۵ دستور
    - 2 × _mm256_mul_ps (ضرب‌ها)
    - 1 × _mm256_add_ps (جمع blend)
    - 1 × _mm256_sub_ps (محاسبه 1-α)
    - 1 × _mm256_add_ps (آپدیت α با x_step)
  - عملیات تبدیل/بسته‌بندی: ۷ دستور
    - 2 × _mm256_cvtepu8_epi32 (تبدیل uchar به int32)
    - 2 × _mm256_cvtepi32_ps (تبدیل int32 به float32)
    - 1 × _mm256_cvtps_epi32 (تبدیل float32 به int32)
    - 2 × _mm_packus_... (فشرده‌سازی int32 به uchar با اشباع)
  - عملیات حافظه: ۳ دستور
    - 2 × _mm_loadl_epi64 (بارگذاری base و watermark)
    - 1 × _mm_storel_epi64 (ذخیره خروجی)
  - مجموع تخمینی دستورات وکتور: ~۱۵ دستور

۳. نتیجه‌گیری

- نسبت توان عملیاتی (خام): ~64 دستور (سریال) / ~15 دستور (موازی) ≈ 4.3x
- نسبت تئوریک (فقط FP): 40 دستور (سریال) / 5 دستور (موازی) ≈ 8.0x

سقف تئوریک 8x در عمل دست‌یافتنی نیست، زیرا بخش بزرگی از گلوگاه محاسباتی، مربوط به سربار (Overhead) سنگین تبدیل نوع داده (Conv/Pack) و دسترسی به حافظه است. همانطور که محاسبه شد، نسبت واقعی‌تر دستورات در حلقه داخلی (با احتساب تبدیل و حافظه) در حدود ~4.3x است. این حساب سرانگشتی، Speedup مشاهده‌شده 5.64x را کاملاً توجیه می‌کند و نشان می‌دهد که این الگوریتم به شدت وابسته به محاسبات (Compute-Bound) بوده است.

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)
- درنهایت محاسبه اندازه گرادیان هر پیکسل

```
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 بدار.

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 عداد 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 شیف 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 میدهیم تا 8 پیکسل را محاسبه کنیم.

- FMA: Fused Multiply-Add

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

- بینه 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]);
```

- Vectorization و FMA

```
// [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 مینوشت، و بلافاصله پس از آن تابع sobel خروجی تصویر قبل را از Ram میخواند و دوباره در Ram مینوشت. 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 گلوکاه حافظه کنترل شود.

در نتیجه ساختار دو تابع sobel , convolve 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) {
```

در نتیجه در تابع main و تعریف یک ثابت به عنوان بازه ارتفاع نوار ها (برای استفاده از کش L3 برابر 64) خواهیم داشت:

```
for (int y_tile = 0; y_tile < rows; y_tile += TILE_HEIGHT) {

        // --- Step 1: Define the Sobel rows this tile is responsible for ---
        int y_sobel_start = max(1, y_tile);
        int y_sobel_end   = min(rows - 1, y_tile + TILE_HEIGHT);

        // If this tile has no valid rows to process (e.g., it's a tile for row 0), skip.
        if (y_sobel_start >= y_sobel_end) {
            continue;
        }

        // --- Step 2: Define the Gaussian rows needed for this Sobel tile ---
        // To run Sobel on [y_sobel_start, y_sobel_end),
        // we need blurred data from [y_sobel_start-1] up to [y_sobel_end].
        // (Sobel at y_sobel_end-1 reads blurred[y_sobel_end])

        int y_gauss_start = y_sobel_start - 1;
        int y_gauss_end   = y_sobel_end + 1;

        y_gauss_start = max(1, y_gauss_start);
        y_gauss_end   = min(rows - 1, y_gauss_end);

        // Run Gaussian on the required (and valid) range
        if (y_gauss_start < y_gauss_end) {
            convolve_parallel_tiled(gray_float, blurred_parallel, gaussian_kernel, GAUSSIAN_RECIPROCAL, y_gauss_start, y_gauss_end);
        }

        // --- Step 3: Run Sobel on the Tile ---
        sobel_magnitude_fused_parallel_tiled(blurred_parallel, magnitude_parallel, y_sobel_start, y_sobel_end);
    }
```

Speedup .3

```
alireza@DESKTOP-7OORIV9 MSYS /e/UT/PP/CA/Parallel-Programming/ca2/sobel
$ ./test.out
Image Size: 576x576
--- Running *Optimized* Serial Sobel Pipeline ---
Serial Time (clocks): 4172301
--- Running Parallel Sobel Pipeline (Tiled + SIMD) ---
Parallel Time (clocks): 1366835
--- Results ---
Serial Time:   4172301 clocks
Parallel Time: 1366835 clocks
Speedup:       3.05x
```

Network

1. نسخه سریال

هدف: پیاده سازی شبکه عصبی سه لایه با ساختار (8,16,8)

- محاسبه هر نورون

محاسبه برای هر نورون شامل یک جمع وزن دار از ورودی لایه قبل + bias و درنهایت اعمال خروجی در یک Activation Function که در این برنامه ReLU استفاده شده میباشد.

```
// === Activation Function ===
inline float activation_relu(float x) {
    return (x > 0.0f) ? x : 0.0f;
}
```

- ورودی

یک برنامه پایتون مقادیر ورودی، بایاس و وزن های بین نورون هارا بصورت رندوم مقدار دهی و ذخیره میکند

- چیدمان وزن ها

اگر چیدمان ماتریس وزن ها بصورت $W[Inputs][Outputs]$ باشد، از آنجایی که محاسبه هر نورون نیازمند دسترسی بصورت عمودی به حافظه میشود، باعث از بین بردن محلی بود کش میشود و کش خراب میشود.

در نتیجه برای بهینه سازی این بخش، وزن ها بصورت $W[Outputs][Inputs]$ ذخیره میشوند تا نسخه سریال بهینه باشد.

- پیاده سازی

تابع `forward_layer_serial` محاسبات را برای یک لایه انجم میدهد.

```
void forward_layer_serial(
    float* output_vector,
    const float* input_vector,
    const float* weights_transposed, // Reads 'weights_xx_t.bin'
    const float* bias_vector,
    int num_inputs,
    int num_outputs
) {
    for (int 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]);
    }
}
```

سپس به صورت static برای ترنزیشن ها Input to hidden و hidden to output استفاده میشود، با این حال در صورت افزایش لایه ها و نورون ها نیز قابلیت استفاده دارد.

```
unsigned long long start_serial, end_serial;
start_serial = __rdtsc();    identifier "__rdtsc" is undefined

forward_layer_serial(hidden_serial, input, weights_ih_t, bias_h, INPUT_SIZE, HIDDEN_SIZE);
forward_layer_serial(output_serial, hidden_serial, weights_ho_t, bias_o, HIDDEN_SIZE, OUTPUT_SIZE);

end_serial = __rdtsc();
unsigned long long time_serial = end_serial - start_serial;
```

2. نسخه موازی

برای موازی سازی نسخه سریال، دو ایده وجود دارد که هردو آن ها برای مقایسه پیاده سازی شدند.
ایده اول: موازی سازی ضرب داخلی برا هر نود (درون فولدر Network)
ایده دوم: همگام سازی محاسبات برای تمام لایه (درون فولدر Network2)

a. Idea#1

هدف اصلی این است که حلقه داخلی درونی تابع `forward_layer_serial` تابع را موازی سازی کنیم.

```
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]);
```

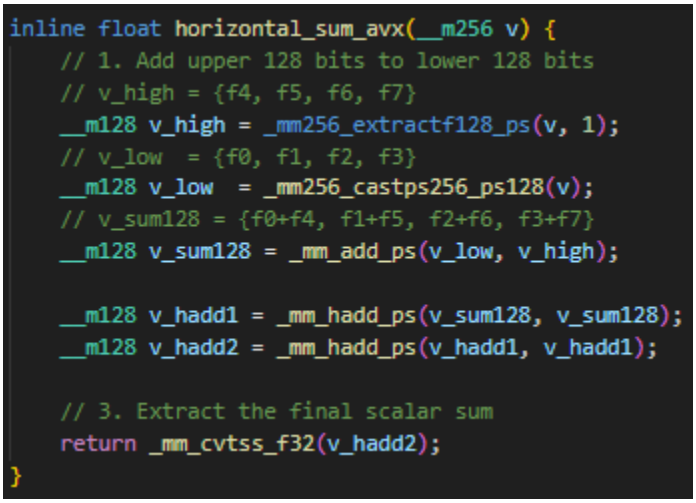
- پیاده سازی:
به جای محاسبه تک به تک ورودی ها و وزن ها، آن ها را با استفاده از رجیستر ها __m256 در بسته های 8 تایی بارگذاری کرده و عمل ضرب و جمع را بصورت همزمان با استفاده از fmadd انجام میدهیم.

```
// --- Vectorized Inner Loop ---
for (int i = 0; i < num_inputs; i += 8) {
    // Load 8 inputs
    __m256 v_input = _mm256_load_ps(input_vector + i);
    // Load 8 weights
    __m256 v_weight = _mm256_load_ps(p_weight_row + i);

    // Fused Multiply-Add: v_sum = v_sum + (v_input * v_weight)
    v_sum = _mm256_fmadd_ps(v_input, v_weight, v_sum);
}
```

- جمع افقی:
خروجی fmadd درون v_sum ذخیره میشود، اما همچنان نیاز است تا جمع این اعضا نیز محاسبه شود، که گلوگاه اصلی این نسخه است چرا که باید sum تمامی اعضا v_sum را محاسبه کند.

برای این منظور از تابع `horizontal_sum_avx` استفاده شده



```
inline float horizontal_sum_avx(__m256 v) {
    // 1. Add upper 128 bits to lower 128 bits
    // v_high = {f4, f5, f6, f7}
    __m128 v_high = _mm256_extractf128_ps(v, 1);
    // v_low  = {f0, f1, f2, f3}
    __m128 v_low  = _mm256_castps256_ps128(v);
    // v_sum128 = {f0+f4, f1+f5, f2+f6, f3+f7}
    __m128 v_sum128 = _mm_add_ps(v_low, v_high);

    __m128 v_hadd1 = _mm_hadd_ps(v_sum128, v_sum128);
    __m128 v_hadd2 = _mm_hadd_ps(v_hadd1, v_hadd1);

    // 3. Extract the final scalar sum
    return _mm_cvtss_f32(v_hadd2);
}
```

که به نسبت پر هزینه است، چرا که مجموعه ای از دستورالعمل های add , shifting در کنار یکدیگر است.

- speedup:

به دلیل سربار موجود در جمع افقی در بهترین اسپیدآپ 2 دارد.

b. Idea#2

در اصل در اینجا ما قصد داریم که در حلقه خارجی تابع `forward_layer_serial` موازی سازی انجام دهیم. یعنی محاسبه 8 نورون خروجی به صورت همزمان به طوری که از عضو از رجیستر m256 یک نورون مجزا را محاسبه کند

- چیدمان وزن ها

بر خلاف بخش قبل، در اینجا ما نیاز داریم که به همان صورت $W[Inputs][Outputs]$ وزن هارا ذخیره کنیم، چرا که قصد داریم برای هر عضو input به صورت موازی ضرب در 8 وزن را محاسبه کنیم.

- پیاده سازی:

درون رجیستر v_sum از همان مرحله اول bias مربوط به همان نورون را بارگذاری میکنیم.
یک حلقه داخلی تا بر روی ورودی ها پیمایش کند.
بارگذاری یک ورودی بر روی رجیستر به شکل کپی.
بارگذاری پیوسته وزن ها، که بدلیل ساختار ذخیره سازی وزن ها کش سالم میماند.
و در نهایت محاسبه جمع وزن دار برای نورون به کمک fmadd

```
// Loop over output neurons in chunks of 8
for (int j = 0; j < num_outputs; j += AVX_LANE_COUNT) {

    // Load the 8 biases for neurons j, j+1, ..., j+7
    __m256 v_sum = _mm256_load_ps(bias_vector + j);

    // This pointer advances through the V-SIMD weight matrix
    const float* p_weight_chunk = weights_vsimd + (j / AVX_LANE_COUNT) * (num_inputs * AVX_LANE_COUNT);

    // --- Serial Inner Loop (over inputs) ---
    for (int i = 0; i < num_inputs; ++i) {

        // 1. Load *one* input and broadcast it to all 8 lanes
        __m256 v_input = _mm256_set1_ps(input_vector[i]);

        // 2. Load 8 contiguous weights from the V-SIMD layout
        //    (w[j,i], w[j+1,i], ..., w[j+7,i])
        __m256 v_weights = _mm256_load_ps(p_weight_chunk + (i * AVX_LANE_COUNT));

        // 3. Fused Multiply-Add
        // v_sum[0] = v_sum[0] + (v_input[0] * v_weights[0])
        // v_sum[1] = v_sum[1] + (v_input[1] * v_weights[1])
        // ... (and all 8 lanes are identical: in[i])
        v_sum = _mm256_fmadd_ps(v_input, v_weights, v_sum);
    }

    // --- Apply Activation (ReLU) on all 8 sums at once ---
    v_sum = _mm256_relu_ps(v_sum);

    // --- Store 8 final results ---
    _mm256_store_ps(output_vector + j, v_sum);
}
```

- Activation Function:

در نهایت ما نیاز داریم تا تابع فعالساز را برای هر یک از lane های خروجی v_sum محاسبه کنیم.

```
inline __m256 _mm256_relu_ps(__m256 v) {
    const __m256 v_zero = _mm256_setzero_ps();
    return _mm256_max_ps(v, v_zero);
}
```

- Speedup:

در این ایده، بدلیل حذف کامل گلوگاه جمع افقی در ایده 1 و دسترسی مناسب به حافظه بدون خراب کردن کش محلی، خروجی speedup بهتری داشته.

```
--- Verification ---
SUCCESS: Serial and Parallel outputs match within tolerance!

--- Performance Results (Idea 3) ---
Serial Time (clocks):   4870
Parallel Time (clocks): 1148
Speedup:                4.24x
```