گزارش پروژه پایانی ساختار زبان و کامپیوتر

طراحی برنامه Convolution-2D

علی فتحی

فهرست مطالب

- **مقدمه:**

در این گزارش به بررسی برنامه Conv2D می‌پردازیم، به ساختار برنامه، نحوه اجرا، امکانات و محدودیت‌های آن اشاره می‌کنیم و در آخر چند نوت‌بوک تکمیلی جهت بررسی بیشتر نتایج حاصل شده و استفاده از این برنامه در زبان‌های سطح بالا، معرفی می‌کنیم.

- **ساختار کلی برنامه:**

این برنامه، می‌تواند به صورت تعاملی و یا از طریق آرگومان‌های رابط خط فرمان، از کاربر ورودی‌های مورد نیاز، از جمله آدرس عکس ورودی، فیلتر انتخابی، نوع پردازش رنگ تصویر (grayscale و یا rgb) و ... را گرفته و عمل convolution را به کمک یکی از ۳ موتور (engine) اصلی خود انجام می‌دهد. همچنین در صورت درخواست کاربر، خروجی را در آدرس تعیین‌شده ذخیره می‌کند.

موتورهای موجود در برنامه عبارتند از:

- Baseline: کد ساده C++ که به صورت scalar عمل می‌کند
- SSE: استفاده از رجیسترهای SSE (۱۲۸ بیتی) پردازنده x86
- AVX: استفاده از رجیسترهای AVX (۲۵۶ بیتی) پردازنده x86

از آنجایی که این برنامه با داده‌هایی از نوع float32 کار می‌کند، رجیسترهای SSE می‌توانند ۴ پردازش، و رجیسترهای AVX، ۸ پردازش را به طور همزمان انجام دهند.

- **نحوه اجرای برنامه:**

در این بخش توابعی را مشاهده می‌کنید که در ابتدای اجرای برنامه، نوع اجرای برنامه را (Interactive و یا CLI) مشخص کرده و طبق آن توابع مربوطه بعدی را صدا می‌زنند.

سپس اگر برنامه به صورت تعاملی اجرا شده باشد، از کاربر درخواست می‌شود مقادیر ورودی‌های موردنظر را وارد کند، و در غیر این صورت از آرگومان‌های رابط خط فرمان استفاده می‌شود.

همانطور که مشاهده می‌کنید، آرگومان‌های CLI باید به صورت زیر به برنامه داده شوند:

```cpp
void print_help() {
    std::cout <<
    "Usage: conv2d [OPTIONS]\n\n"
    "Modes:\n"
    "  -m --mode functional | speed\n\n"
    "Options:\n"
    "  -e, --engine   baseline | sse | avx\n"
    "  -k, --ktype    sharpen | box_blur | gaussian_blur | sobel_x | sobel_y\n"
    "  -s, --ksize    kernel size (e.g. 3)\n"
    "  -i, --input    input file or directory\n"
    "  -o, --output   output file or directory (optional)\n"
    "  -c, --color    grayscale | rgb (optional) (default = rgb)\n"
    "  -h, --help     show this help\n";
}
```

```cpp
static int read_input_and_run_functional_test() {

    int res = CODE_SUCCESS;

    FunctionalTestParams params;

    res = read_functional_test_input(params);
    if (res != CODE_SUCCESS)
        return res;

    res = run_functional_test(params);

    return res;
}

static int read_input_and_run_speed_test() {

    int res = CODE_SUCCESS;

    SpeedTestParams params;

    res = read_speed_test_params(params);
    if (res != CODE_SUCCESS)
        return res;

    res = run_speed_test(params);

    return res;
}

static int run_interactive() {

    int res = CODE_SUCCESS;

    int option_def = RUN_MODE_SPEED_TEST;

    int option;

    OptionEntry options[2];
    options[0].option_number = RUN_MODE_FUNCTIONAL_TEST;
    options[0].option_name   = RUN_MODE_FUNCTIONAL_TEST_STR;
    options[1].option_number = RUN_MODE_SPEED_TEST;
    options[1].option_name   = RUN_MODE_SPEED_TEST_STR;

    res = read_option("Option", options, 2, stdin, &option_def, &option);
    if (res != CODE_SUCCESS) {
        print_err("Failed to read option", res);
        return res;
    }

    if (option == RUN_MODE_FUNCTIONAL_TEST)
        res = read_input_and_run_functional_test();
    else if (option == RUN_MODE_SPEED_TEST)
        res = read_input_and_run_speed_test();

    return res;
}

static int run_cli(int argc, char **argv) {

    int res = CODE_SUCCESS;

    CLIArgs args;

    res = parse_cli(argc, argv, args);
    if (res != CODE_SUCCESS || args.help) {
        print_help();
        return res;
    }

    res = validate_cli(args);
    if (res != CODE_VALIDATION_OK) {
        return res;
    }

    if (args.run_mode == RUN_MODE_FUNCTIONAL_TEST) {

        FunctionalTestParams params = {
            args.engine_mode,
            args.kernel_type,
            args.kernel_size,
            args.color_mode,
            args.input,
            args.output,
            args.save_output
        };

        res = run_functional_test(params);
    }
    else if (args.run_mode == RUN_MODE_SPEED_TEST) {

        SpeedTestParams params = {
            args.engine_mode,
            args.kernel_type,
            args.kernel_size,
            args.color_mode,
            args.input,
            args.output,
            args.save_output
        };

        res = run_speed_test(params);

    }

    return res;
}

int main(int argc, char **argv) {

    int res = CODE_SUCCESS;

    if (argc == 1) res = run_interactive();
    else           res = run_cli(argc, argv);

    return res;
}
```

- **ورودی‌های برنامه:**

- **Mode:** در حالت functional تنها یک عکس به عنوان ورودی داده می‌شود. هدف این مود تست عملکرد برنامه و مشاهده تاثیر فیلتر انتخاب شده روی عکس ورودی می‌باشد.
  مود speed برای تست سرعت برنامه است.
- **Engine:** موتور اصلی برنامه که نحوه پردازش عمل convolution را مشخص می‌کند. می‌تواند به صورت baseline، و یا با استفاده از رجیسترهای SSE و AVX پردازنده x86 باشد.
- **Kernel Type:** این آرگومان نوع کرنل (فیلتر) را انتخاب می‌کند. برای مثال می‌توان از فیلترهای sharpen، blur و ... استفاده کرد.
- **Kernel Size:** این آرگومان سایز کرنل را انتخاب می‌کند. در حال حاضر این مقدار می‌تواند ۳، ۵ و یا ۷ باشد (به نوع کرنل نیز وابسته است)
- **Input/Output:** آدرس فایل یا پوشه ورودی/خروجی. در مود functional آدرس فایل ورودی و آدرس فایل خروجی برای ذخیره عکس داده می‌شود و در مود speed، یک پوشه حاوی عکس به عنوان ورودی داده می‌شود و یک پوشه برای ذخیره عکس‌های خروجی.
- **Color:** تعداد کانال‌های پردازش را مشخص می‌کند، در حالت grayscale، عکس‌ها به صورت تک‌کاناله بارگذاری و پردازش می‌شوند و در حالت rgb، به صورت ۳ کاناله.

در اجرای Interactive نیز همین ورودی‌ها درخواست می‌شوند.

- **Kernel Factory:**

در این تابع، با توجه به نوع و اندازه کرنل، یک کرنل پیش‌فرض انتخاب شده و یا یک کرنل مناسب ساخته می‌شود. اگر در آینده قصد اضافه کردن فیلترهای بیشتری به برنامه داشته باشیم، می‌توانیم از طریق همین تابع، فیلترهای دلخواه خود را اضافه کنیم.

```
int get_kernel(int kernel_type, int kernel_size, Kernel& kernel) {

    if (!validate_odd_kernel(kernel_size)) {
        print_err("Kernel size must be odd and >= 3", CODE_FAILURE_NOT_SUPPORTED);
        return CODE_FAILURE_NOT_SUPPORTED;
    }

    kernel.size = kernel_size;
    kernel.type = kernel_type;
    kernel.data = new float[kernel_size * kernel_size];

    int status = CODE_SUCCESS;

    switch (kernel_type) {

        case KERNEL_TYPE_SHARPEN:
            status = create_sharpen(kernel);
            break;

        case KERNEL_TYPE_SOBEL_X:
            status = create_sobel_x(kernel);
            break;

        case KERNEL_TYPE_SOBEL_Y:
            status = create_sobel_y(kernel);
            break;

        case KERNEL_TYPE_BOX_BLUR:
            status = create_box_blur(kernel);
            break;

        case KERNEL_TYPE_GAUSSIAN_BLUR:
            status = create_gaussian_blur(kernel);
            break;

        default:
            print_err("Unsupported kernel type", CODE_FAILURE_NOT_SUPPORTED);
            status = CODE_FAILURE_NOT_SUPPORTED;
            break;
    }

    if (status != CODE_SUCCESS) {
        delete[] kernel.data;
        kernel.data = nullptr;
    }

    return status;
}
```

- **خواندن و ذخیره عکس (OpenCV2):**

برای خواندن عکس‌ها و تبدیل آن‌ها به فرمت float *، و همچنین تبدیل آرایه float به عکس و ذخیره آن، از کتابخانه OpenCV استفاده شده است. اگر عکس به صورت grayscale باشد، تنها یک آرایه از float (یک کانال) توسط توابع این بخش ایجاد و یا دریافت می‌شود، و اگر به صورت rgb باشد، ۳ آرایه (البته می‌توانیم مقادیر هر ۳ آرایه را در یک آرایه و به صورت scalar پاس بدهیم).

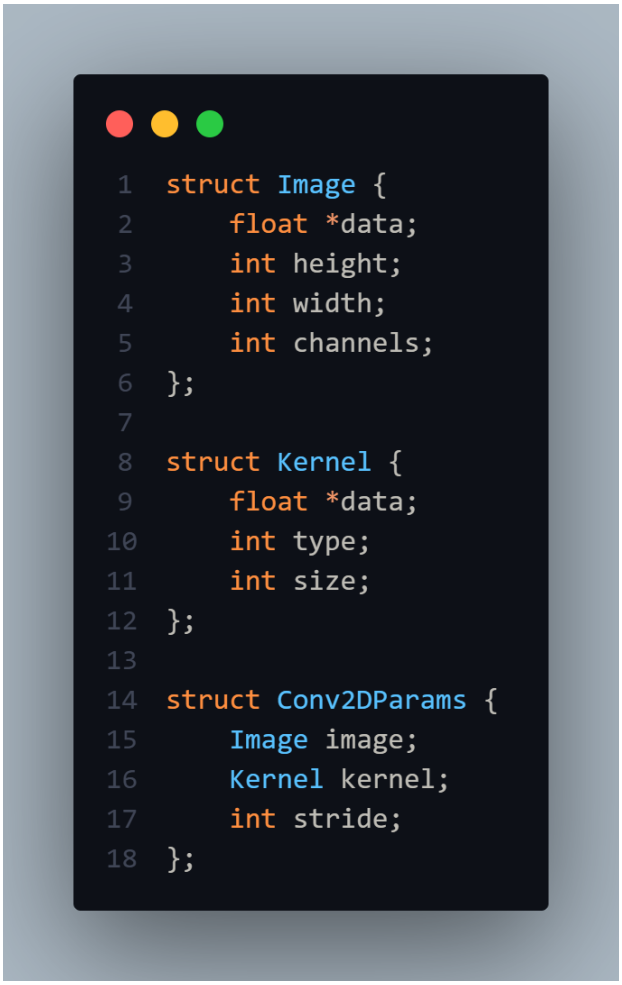
در اینجا قسمتی از توابع مربوط به خواندن و نوشتن عکس‌ها را مشاهده می‌کنید:

```cpp
int load_grayscale_image(
    const char *filename,
    Image& image
) {
    cv::Mat img = cv::imread(filename, cv::IMREAD_GRAYSCALE);

    if (img.empty()) {
        print_err("Failed to load image", CODE_FAILURE_READ_INPUT);
        return CODE_FAILURE;
    }

    cv::Mat img_f;
    img.convertTo(img_f, CV_32F, 1.0  / 255.0);

    image.height   = img_f.rows;
    image.width    = img_f.cols;
    image.channels = CHANNELS_GRAYSCALE;

    image.data = new float[image.height * image.width];

    for (int i = 0; i < image.height; i++) {
        const float *row_ptr = img_f.ptr<float>(i);
        std::memcpy(
            image.data + i * image.width,
            row_ptr,
            image.width * sizeof(float)
        );
    }

    return CODE_SUCCESS;
}
```

```cpp
int save_float_array_as_grayscale_image(
    const char *filename,
    const Image& output
) {
    cv::Mat img_f(output.height, output.width, CV_32F, const_cast<float*>(output.data));

    cv::Mat img_clamped;
    cv::min(img_f, 1.0f, img_clamped);
    cv::max(img_clamped, 0.0f, img_clamped);

    cv::Mat img_u8;
    img_clamped.convertTo(img_u8, CV_8U, 255.0);

    if (!cv::imwrite(filename, img_u8)) {
        print_err("Failed to write output image", CODE_FAILURE_WRITE_OUTPUT);
        return CODE_FAILURE;
    }

    return CODE_SUCCESS;
}
```

- **Struct های به کار رفته در برنامه:**

قبل از ورود به بخش محاسبات، نگاهی به چند استراکت مهم که برای خوانایی بیشتر کد در برنامه به کار برده شده‌اند می‌اندازیم:



```
struct Image {
    float *data;
    int height;
    int width;
    int channels;
};

struct Kernel {
    float *data;
    int type;
    int size;
};

struct Conv2DParams {
    Image image;
    Kernel kernel;
    int stride;
};
```

- **Image:**

حاوی اطلاعات عکس، از جمله طول و عرض، تعداد کانال و اعداد نمایانگر پیکسل‌ها. بدیهی است که تعداد float32 هایی که در data قرار می‌گیرد از رابطه width * height * channels حاصل می‌شود.

- **Kernel:**

مانند Image با این تفاوت که در کاربرد این برنامه از کرنل‌های تک کاناله استفاده می‌کنیم.

- **Conv2DParams:**

حاوی اطلاعاتی که برای عمل convolution به آن نیاز داریم. شایان ذکر است که در برنامه فعلی ما از stride ثابت ۱ استفاده می‌شود.

- **موتورهای Convolution:**

تابع conv2d_channels، همان تابعی است که برای اجرای عملیات کانولوشن آن را صدا می‌زنیم. با توجه به اینکه ممکن است تعداد متغیری کانال داشته باشیم، این تابع، عملیات را به طور جداگانه روی هر کانال اجرا می‌کند:

```
int conv2d_channels(
    int engine_mode,
    const Conv2DParams& params,
    Image& output
) {

    int res = CODE_SUCCESS;

    if (
        params.kernel.size != KERNEL_SIZE_3 &&
        engine_mode != ENGINE_MODE_BASELINE
    ) {
        engine_mode = ENGINE_MODE_BASELINE;
        print_warn("Only 3x3 kernels are supported in SSE/AVX engines. Falling back to baseline engine.");
    }

    Image image   = params.image;
    Kernel kernel = params.kernel;

    int out_height = (image.height - kernel.size) / params.stride + 1;
    int out_width  = (image.width - kernel.size) / params.stride + 1;

    output.height   = out_height;
    output.width    = out_width;
    output.channels = image.channels;
    output.data     = new float[output.height * output.width * output.channels];

    for (int c = 0; c < output.channels; c++) {
        Conv2DParams ch_params = params;
        ch_params.image.data = channel_ptr(image, c);

        Image ch_out;
        res = conv2d(engine_mode, ch_params, ch_out);

        if (res != CODE_SUCCESS) {
            std::string err_msg = "Operation failed on channel " + std::to_string(c);
            print_err(err_msg.c_str(), res);
            return res;
        }

        std::memcpy(
            channel_ptr(output, c),
            ch_out.data,
            output.height * output.width * sizeof(float)
        );

        delete[] ch_out.data;
    }

    return res;
}
```

در شبکه‌های عصبی پیچشی (CNN) معمولا باید تعداد کانال‌های کرنل با تعداد کانال‌های ورودی آن لایه برابر باشد و خروجی هر عمل convolution تک‌کاناله است، در این کاربرد اگر بخواهیم n کانال

خروجی داشته باشیم (یا به عبارتی n ویژگی از عکس استخراج کنیم) باید n بار عمل convolution را با کرنل‌های متفاوت اجرا کنیم. اما از آنجایی که برنامه ما یک برنامه convolution کلاسیک است که تنها می‌خواهیم اثر فیلترهای گوناگون را روی عکس‌ها مشاهده و سرعت engine های مختلف را مقایسه کنیم، یک کرنل مشخص را روی هر n کانال عکس ورودی اعمال کرده و n کانال خروجی دریافت می‌کنیم.

در تابع conv2d، بر اساس ورودی engine_mode انتخاب می‌کنیم که پردازش با کدام موتور صورت گیرد:

```
static int conv2d(
    int engine_mode,
    const Conv2DParams& params,
    Image& output
) {
    int res = CODE_SUCCESS;

    res = conv2d_validation(
            engine_mode,
            params);

    if (res != CODE_VALIDATION_OK) {
        print_err("Invalid arguments to function", res);
        return CODE_FAILURE_INVALID_ARG;
    }

    switch(engine_mode) {
        case ENGINE_MODE_BASELINE:
            res = conv2d_baseline(
                    params,
                    output);
            break;

        case ENGINE_MODE_SSE:
            res = conv2d_sse(
                    params,
                    output);
            break;

        case ENGINE_MODE_AVX:
            res = conv2d_avx(
                    params,
                    output);
            break;

        default: res = CODE_FAILURE;
    }

    return res;
}
```

حال، ۳ موتور مختلف برنامه را بررسی می‌کنیم:

```
static int conv2d_baseline(
    const Conv2DParams& params,
    Image& output
) {

    int res = CODE_SUCCESS;

    Image image = params.image;
    Kernel kernel = params.kernel;

    int out_height = (image.height - kernel.size) / params.stride + 1;
    int out_width  = (image.width - kernel.size) / params.stride + 1;
    int out_size   = out_height * out_width;

    output.height = out_height;
    output.width  = out_width;
    output.data   = new float[out_size];

    for (int i = 0; i < out_height; i++) {
        for (int j = 0; j < out_width; j++) {

            float sum = 0.0f;

            int base_i = i * params.stride;
            int base_j = j * params.stride;

            for (int u = 0; u < kernel.size; u++) {

                int img_row = (base_i + u) * image.width;
                int ker_row = u * kernel.size;

                for (int v = 0; v < kernel.size; v++) {
                    int img_idx = img_row + (base_j + v);
                    int ker_idx = ker_row + v;

                    sum += image.data[img_idx] * kernel.data[ker_idx];
                }
            }

            int out_idx = i * out_width + j;
            output.data[out_idx] = sum;
        }
    }

    return res;
}
```

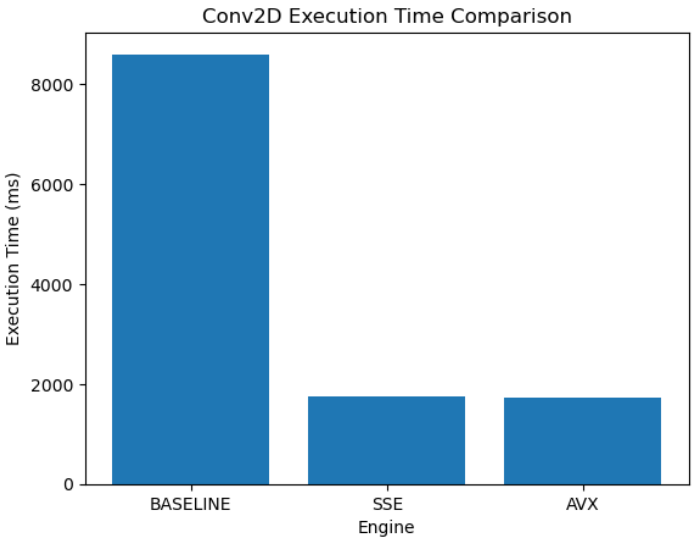
```
static int conv2d_sse(
    const Conv2DParams& params,
    Image& output
) {
    int res = CODE_SUCCESS;

    Image image = params.image;
    Kernel kernel = params.kernel;

    int out_height = (image.height - kernel.size) / params.stride + 1;
    int out_width  = (image.width - kernel.size) / params.stride + 1;
    int out_size   = out_height * out_width;

    output.height = out_height;
    output.width  = out_width;
    output.data   = new float[out_size];

    __m128 k00 = _mm_set1_ps(kernel.data[0]);
    __m128 k01 = _mm_set1_ps(kernel.data[1]);
    __m128 k02 = _mm_set1_ps(kernel.data[2]);

    __m128 k10 = _mm_set1_ps(kernel.data[3]);
    __m128 k11 = _mm_set1_ps(kernel.data[4]);
    __m128 k12 = _mm_set1_ps(kernel.data[5]);

    __m128 k20 = _mm_set1_ps(kernel.data[6]);
    __m128 k21 = _mm_set1_ps(kernel.data[7]);
    __m128 k22 = _mm_set1_ps(kernel.data[8]);

    constexpr int SSE_FLOATS = 4;

    for (int i = 0; i < out_height; i++) {
        int base_i = i * params.stride;

        int j = 0;
        for (; j <= out_width - SSE_FLOATS; j += SSE_FLOATS) {
            int base_j = j * params.stride;

            __m128 sum = _mm_setzero_ps();

            // Row -1
            __m128 r0 = _mm_loadu_ps(&image.data[(base_i + 0) * image.width + base_j + 0]);
            __m128 r1 = _mm_loadu_ps(&image.data[(base_i + 0) * image.width + base_j + 1]);
            __m128 r2 = _mm_loadu_ps(&image.data[(base_i + 0) * image.width + base_j + 2]);

            sum = _mm_add_ps(sum, _mm_mul_ps(r0, k00));
            sum = _mm_add_ps(sum, _mm_mul_ps(r1, k01));
            sum = _mm_add_ps(sum, _mm_mul_ps(r2, k02));

            // Row 1
            r0 = _mm_loadu_ps(&image.data[(base_i + 1) * image.width + base_j + 0]);
            r1 = _mm_loadu_ps(&image.data[(base_i + 1) * image.width + base_j + 1]);
            r2 = _mm_loadu_ps(&image.data[(base_i + 1) * image.width + base_j + 2]);

            sum = _mm_add_ps(sum, _mm_mul_ps(r0, k10));
            sum = _mm_add_ps(sum, _mm_mul_ps(r1, k11));
            sum = _mm_add_ps(sum, _mm_mul_ps(r2, k12));

            // Row 2
            r0 = _mm_loadu_ps(&image.data[(base_i + 2) * image.width + base_j + 0]);
            r1 = _mm_loadu_ps(&image.data[(base_i + 2) * image.width + base_j + 1]);
            r2 = _mm_loadu_ps(&image.data[(base_i + 2) * image.width + base_j + 2]);

            sum = _mm_add_ps(sum, _mm_mul_ps(r0, k20));
            sum = _mm_add_ps(sum, _mm_mul_ps(r1, k21));
            sum = _mm_add_ps(sum, _mm_mul_ps(r2, k22));

            _mm_storeu_ps(&output.data[i * out_width + j], sum);
        }

        for (; j < out_width; j++) {
            int base_j = j * params.stride;
            float s = 0.f;
            for (int ky = 0; ky < 3; ky++)
                for (int kx = 0; kx < 3; kx++)
                    s += image.data[(base_i + ky) * image.width + (base_j + kx)] *
                         kernel.data[ky * 3 + kx];

            output.data[i * out_width + j] = s;
        }
    }

    return res;
}
```

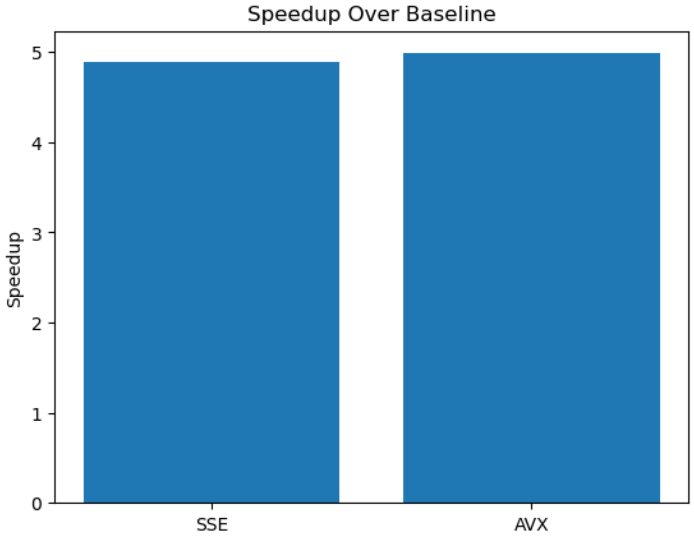
```
static int conv2d_avx(
    const Conv2DParams& params,
    Image& output
) {
    int res = CODE_SUCCESS;

    Image image = params.image;
    Kernel kernel = params.kernel;

    int out_height = (image.height - kernel.size) / params.stride + 1;
    int out_width  = (image.width - kernel.size) / params.stride + 1;
    int out_size   = out_height * out_width;

    output.height = out_height;
    output.width  = out_width;
    output.data   = new float[out_size];

    __m256 k00 = _mm256_set1_ps(kernel.data[0]);
    __m256 k01 = _mm256_set1_ps(kernel.data[1]);
    __m256 k02 = _mm256_set1_ps(kernel.data[2]);

    __m256 k10 = _mm256_set1_ps(kernel.data[3]);
    __m256 k11 = _mm256_set1_ps(kernel.data[4]);
    __m256 k12 = _mm256_set1_ps(kernel.data[5]);

    __m256 k20 = _mm256_set1_ps(kernel.data[6]);
    __m256 k21 = _mm256_set1_ps(kernel.data[7]);
    __m256 k22 = _mm256_set1_ps(kernel.data[8]);

    constexpr int AVX_FLOATS = 8;

    for (int i = 0; i < out_height; i++) {
        int base_i = i * params.stride;

        int j = 0;
        for (; j <= out_width - AVX_FLOATS; j += AVX_FLOATS) {
            int base_j = j * params.stride;

            __m256 sum = _mm256_setzero_ps();

            // Row 0
            __m256 r0 = _mm256_loadu_ps(&image.data[(base_i + 0) * image.width + base_j + 0]);
            __m256 r1 = _mm256_loadu_ps(&image.data[(base_i + 0) * image.width + base_j + 1]);
            __m256 r2 = _mm256_loadu_ps(&image.data[(base_i + 0) * image.width + base_j + 2]);

            sum = _mm256_add_ps(sum, _mm256_mul_ps(r0, k00));
            sum = _mm256_add_ps(sum, _mm256_mul_ps(r1, k01));
            sum = _mm256_add_ps(sum, _mm256_mul_ps(r2, k02));

            // Row 1
            r0 = _mm256_loadu_ps(&image.data[(base_i + 1) * image.width + base_j + 0]);
            r1 = _mm256_loadu_ps(&image.data[(base_i + 1) * image.width + base_j + 1]);
            r2 = _mm256_loadu_ps(&image.data[(base_i + 1) * image.width + base_j + 2]);

            sum = _mm256_add_ps(sum, _mm256_mul_ps(r0, k10));
            sum = _mm256_add_ps(sum, _mm256_mul_ps(r1, k11));
            sum = _mm256_add_ps(sum, _mm256_mul_ps(r2, k12));

            // Row 2
            r0 = _mm256_loadu_ps(&image.data[(base_i + 2) * image.width + base_j + 0]);
            r1 = _mm256_loadu_ps(&image.data[(base_i + 2) * image.width + base_j + 1]);
            r2 = _mm256_loadu_ps(&image.data[(base_i + 2) * image.width + base_j + 2]);

            sum = _mm256_add_ps(sum, _mm256_mul_ps(r0, k20));
            sum = _mm256_add_ps(sum, _mm256_mul_ps(r1, k21));
            sum = _mm256_add_ps(sum, _mm256_mul_ps(r2, k22));

            _mm256_storeu_ps(&output.data[i * out_width + j], sum);
        }

        for (; j < out_width; j++) {
            int base_j = j * params.stride;
            float s = 0.f;
            for (int ky = 0; ky < 3; ky++)
                for (int kx = 0; kx < 3; kx++)
                    s += image.data[(base_i + ky) * image.width + (base_j + kx)] *
                         kernel.data[ky * 3 + kx];

            output.data[i * out_width + j] = s;
        }
    }

    return res;
}
```

همانطور که مشاهده می‌کنید، در تابع baseline، پردازش به صورت خطی است و هیچگونه بهینه‌سازی در آن صورت نگرفته. البته compiler بهینه‌سازی و حتی در مواقعی موازی‌سازی را نیز تا حدی انجام می‌دهد. در دو تابع sse و avx، پردازش به صورت برداری و با کمک رجیسترهای برداری ۱۲۸ بیتی و ۲۵۶ بیتی پردازنده x86 انجام شده است.

در نوت‌بوک speed-comparison درباره میزان افزایش سرعت sse و avx نسبت به حالت baseline صحبت می‌کنیم.

تصاویری از نتیجه بررسی سرعت در نوت‌بوک مذکور:





- ## توضیح پیاده‌سازی SIMD

در این بخش، به جای تمرکز بر نتایج سرعت، نحوه پیاده‌سازی برداری توضیح داده می‌شود.

- ### استفاده از Intrinsics

در موتورهای SSE و AVX از توابع intrinsic مربوط به معماری پردازنده x86 استفاده شده است. این توابع به کامپایلر اجازه می‌دهند مستقیماً دستورهای SIMD متناظر را تولید کند، بدون نیاز به نوشتن کد اسمبلی به صورت جداگانه.

برای مثال:

- بارگذاری داده‌ها به رجیستر برداری
- ضرب برداری مقادیر کرنل و پیکسل‌ها
- جمع برداری نتایج میانی

- **عرض رجیسترها**

- در SSE از رجیسترهای ۱۲۸ بیتی استفاده شده که شامل ٤ مقدار float32 هستند.
- در AVX از رجیسترهای ۲۵٦ بیتی استفاده شده که شامل ۸ مقدار float32 هستند.

در نتیجه در هر سیکل پردازشی، به ترتیب ٤ یا ۸ عملیات ضرب-جمع به صورت همزمان انجام می‌شود.

- **نحوه بردارسازی حلقه‌ها**

در پیاده‌سازی SIMD، حلقه داخلی convolution که روی عرض تصویر حرکت می‌کند، به صورت بلوک‌های ٤ یا ۸ تایی پردازش می‌شود. به گونه‌ای که:

- چند پیکسل مجاور به صورت همزمان load می‌شوند.
- با ضرایب کرنل ضرب می‌شوند.
- نتایج در یک رجیستر جمع می‌شوند.
- در نهایت مقدار خروجی ذخیره می‌شود.

- **مدیریت بخش باقیمانده (Tail Processing)**

  اگر عرض تصویر مضرب ٤ یا ٨ نباشد، بخش انتهایی تصویر که در قالب بردار کامل قرار نمی‌گیرد، به صورت scalar یا خطی پردازش می‌شود.

داده‌های تصویر به صورت آرایه خطی ذخیره شده‌اند، که باعث می‌شود دسترسی حافظه در حلقه داخلی به صورت متوالی باشد. این موضوع نقش مهمی در عملکرد SIMD ایفا می‌کند.

نکته قابل توجه این است که در حالی که عرض رجیسترهای AVX دو برابر SSE است، افزایش سرعت دقیقاً دو برابر مشاهده نمی‌شود. تحلیل کمی و تجربی این موضوع در نوت‌بوک speed-comparison صورت گرفته است.

- **تصاویری از اجرای برنامه:**

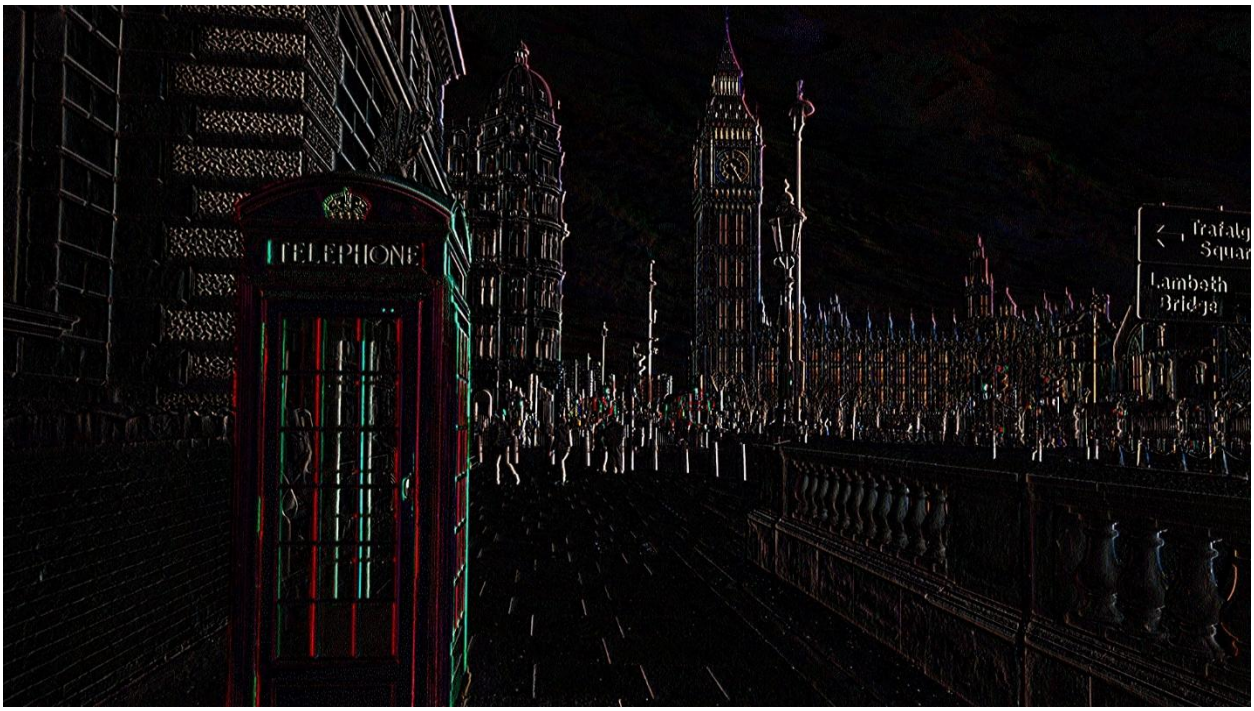
در ادامه تصاویری از اجرای برنامه، عکس‌های ورودی و خروجی متناظر با آن‌ها قرار داده شده است.

تصویری که فیلترهای مختلف را روی آن تست خواهیم کرد:

```
Options:
        (1) Functional Test
        (2) Speed Test
Enter Option [2] > 1
Options:
        (1) Baseline
        (2) SSE
        (3) AVX
Enter Engine Mode [1] >
Options:
        (1) Sharpen
        (2) Box Blur
        (3) Gaussian Blur
        (4) Sobel X
        (5) Sobel Y
Enter Kernel Type [1] >
Options:
        (3) 3 × 3
        (5) 5 × 5
        (7) 7 × 7
Enter Kernel Size [3] >
Enter Image Path [./images/normal-small/01.jpeg] >
Enter Output Filename [./images/output/01.jpeg] >
Options:
        (1) RGB
        (2) Grayscale
Enter Color Mode [1] >
[TIMING] Engine: Baseline took 86.699823 ms
```

```
Options:
        (1) Functional Test
        (2) Speed Test
Enter Option [2] > 1
Options:
        (1) Baseline
        (2) SSE
        (3) AVX
Enter Engine Mode [1] >
Options:
        (1) Sharpen
        (2) Box Blur
        (3) Gaussian Blur
        (4) Sobel X
        (5) Sobel Y
Enter Kernel Type [1] > 2
Options:
        (3) 3 × 3
        (5) 5 × 5
        (7) 7 × 7
Enter Kernel Size [3] > 7
Enter Image Path [./images/normal-small/01.jpeg] >
Enter Output Filename [./images/output/01.jpeg] >
Options:
        (1) RGB
        (2) Grayscale
Enter Color Mode [1] >
[TIMING] Engine: Baseline took 212.983973 ms
```

```
Options:
        (1) Functional Test
        (2) Speed Test
Enter Option [2] > 1
Options:
        (1) Baseline
        (2) SSE
        (3) AVX
Enter Engine Mode [1] > 3
Options:
        (1) Sharpen
        (2) Box Blur
        (3) Gaussian Blur
        (4) Sobel X
        (5) Sobel Y
Enter Kernel Type [1] > 4
Options:
        (3) 3 × 3
        (5) 5 × 5
        (7) 7 × 7
Enter Kernel Size [3] >
Enter Image Path [./images/normal-small/01.jpeg] >
Enter Output Filename [./images/output/01.jpeg] >
Options:
        (1) RGB
        (2) Grayscale
Enter Color Mode [1] >
[TIMING] Engine: AVX took 22.451762 ms
```



لازم به ذکر است که در این تصاویر از مود functional برنامه و در نوت‌بوک speed-comparison برای مقایسه سرعت، از مود speed استفاده شده است.

- **معرفی نوت‌بوک‌های تکمیلی:**
  - **نوت‌بوک speed-comparison:**

    در این نوت‌بوک سرعت ۳ موتور مختلف با هم مقایسه شده.
  - **نوت‌بوک object-detection:**

    در این نوت‌بوک تلاش شده است با استفاده از فیلترهای پیاده‌سازی شده، اَشکال ساده دیتاست‌های ساختگی را شناسایی و دقت تشخیص را بیازماییم.
  - **نوت‌بوک cnn-pneumonia:**

    در این نوت‌بوک با کمک کتابخانه pytorch، یک مدل بسیار ساده روی داده‌های پزشکی آموزش داده شده و سپس وزن‌های آن را به برنامه C++ خود انتقال می‌دهیم و با ۳ موتور اصلی برنامه آن دقت و سرعت پردازش روی داده‌های test را بررسی میکنیم.