Национальный исследовательский университет ИТМО
Мегафакультет компьютерных технологий и управления
Факультет программной инженерии и компьютерной техники

Отчет по лабораторной работе №2

по курсу «Программирование на C++»

Тема: «Использование SIMD – инструкций»

Выполнил студент группы P4119 _____ Д.Л. Симоновский
(подпись)

Руководитель _____ А. Д. Жданов
(подпись)

24 ноября 2025 г.

Санкт-Петербург
2025

Оглавление

1. Цель работы

Получение практических навыков проектирования, реализации и исследования алгоритмов, использующих векторизацию вычислений с применением SIMD-инструкций. В ходе выполнения работы предусматривается разработка наивного (последовательного) и векторизованного решений задач поиска K ближайших соседей в наборе данных и применения размытия по Гауссу к изображению. Необходимо провести сравнительный анализ производительности реализованных подходов с использованием выбранных метрик эффективности.

2. План работы

1. Выполнить реализацию алгоритма поиска K ближайших соседей.
2. Модифицировать реализацию алгоритма поиска K ближайших соседей с использованием SIMD-инструкций.
3. Выполнить реализацию алгоритма размытия по Гауссу изображения.
4. Модифицировать реализацию алгоритма размытия по Гауссу изображения с использованием SIMD-инструкций.
5. Произвести сравнительное тестирование разработанных программных компонентов.

3. Ход работы

3.1. Алгоритма поиска K ближайших соседей

Алгоритм k-Nearest Neighbors (k-NN) — это метрический алгоритм, используемый для задач классификации, регрессии и поиска похожих объектов. Основная идея метода заключается в «гипотезе компактности»: если метрическое расстояние между примерами мало, то и сами объекты похожи.

В задаче поиска принцип работы алгоритма состоит из следующих шагов:

1. Хранение данных: Система хранит набор векторов-примеров (dataset) в N-мерном пространстве.

2. Получение запроса: на вход поступает вектор-запрос (query) той же размерности.
3. Вычисление расстояний: для каждого вектора из набора данных вычисляется расстояние до вектора-запроса. Наиболее часто используемой метрикой является евклидово расстояние. Формула для двух векторов A и B размерности n:

$$d(A,B) = \sqrt{\sum_{i=0}^{n-1} (A_i - B_i)^2}$$

4. Сортировка и выбор: Все вычисленные расстояния сортируются по возрастанию. Алгоритм отбирает k первых элементов (с наименьшим расстоянием), индексы которых и являются результатом поиска.

В рамках проекта была реализована базовая или "наивная" версия алгоритма на языке C++. Она выполняет линейный проход по всем векторам набора данных без использования сложных структур пространственного индексирования (таких как k-d деревья)

Функция get_euclidean_distance_naive принимает два вектора и последовательно вычисляет сумму квадратов разностей их компонент. Реализация использует стандартный цикл:

Листинг 3.1. Калькулятор евклидового расстояния

```
float KnnSearcher::get_euclidean_distance_naive(const std::vector<float>& a,
const std::vector<float>& b) {
    float sum = 0.0f;
    for (size_t i = 0; i < a.size(); ++i) {
        float diff = a[i] - b[i];
        sum += diff * diff;
    }
    return std::sqrt(sum);
}
```

Основная функция find_naive организует полный перебор всего датасета. Для каждого элемента сохраняется пара «расстояние — индекс». После вычисления расстояний массив сортируется, и возвращаются индексы первых k элементов.

Листинг 3.2. Наивный поиск

```
std::vector<int> KnnSearcher::find_naive(
    const std::vector<std::vector<float>>& dataset,
    const std::vector<float>& query,
    int k
) {
    std::vector<std::pair<float, int>> distances;
    distances.reserve(dataset.size());

    for (size_t i = 0; i < dataset.size(); ++i) {
        float dist = get_euclidean_distance_naive(dataset[i], query);
        distances.push_back({dist, (int)i});
    }

    std::sort(distances.begin(), distances.end());

    std::vector<int> result;
    result.reserve(k);
    for (int i = 0; i < k && i < (int)distances.size(); ++i) {
        result.push_back(distances[i].second);
    }
    return result;
}
```

Данная реализация является эталонной для проверки корректности более сложных оптимизаций, однако она чувствительна к размеру данных, так как имеет сложность O(N * D + N logN) на один запрос, где N — количество векторов, а D — их размерность.

Для ускорения вычислений была реализована векторизованная версия алгоритма, использующая расширение набора команд AVX-512. Технология SIMD (Single Instruction, Multiple Data) позволяет процессору обрабатывать несколько элементов данных одной инструкцией. В случае AVX-512 мы можем выполнять операции сразу над 16 числами типа *float* (так как регистры имеют ширину 512 бит, а *float* занимает 32 бита: 512 / 32 = 16).

В реализации применяются следующие интринсики (встроенные функции компилятора), обеспечивающие доступ к аппаратным возможностям процессора:

_mm512_setzero_ps – инициализирует 512-битный регистр (вектор) нулями.

_mm512_loadu_ps – загружает 16 значений *float* из памяти в регистр. Суффикс *u* (unaligned) допускает невыровненную память.

_mm512_sub_ps – выполняет почленное вычитание двух векторов.

_mm512_fmadd_ps – Fused Multiply-Add: выполняет операцию a * b + c для каждого элемента за одну инструкцию. Используется для накопления суммы квадратов.

_mm512_reduce_add_ps – горизонтальное сложение: суммирует все 16 элементов внутри одного регистра и возвращает скалярный результат.

_mm512_maskz_loadu_ps – загрузка с маской: загружает только те элементы, для которых соответствующий бит маски равен 1, остальные обнуляет. Критично для обработки "хвостов" массивов.

Функция get_euclidean_distance_avx512 разбивает входные векторы на блоки по 16 элементов. Основной цикл обрабатывает полные блоки, используя FMA (Fused Multiply-Add) для одновременного возведения разности в квадрат и прибавления к текущей сумме.

Особенностью AVX-512 является удобная работа с остатками данных (когда размерность вектора не кратна 16). Вместо написания отдельного скалярного цикла используется маскирование: создается битовая маска для оставшихся элементов, и операции загрузки выполняются только для валидных данных, исключая выход за границу массива.

Листинг 3.3. Калькулятор евклидового расстояния на SIMD

```
float KnnSearcher::get_euclidean_distance_avx512(const float* a, const float*
b, size_t size) {
    __m512 sum_vec = _mm512_setzero_ps();
    size_t i = 0;
    for (; i + 16 <= size; i += 16) {
        __m512 va = _mm512_loadu_ps(a + i);
        __m512 vb = _mm512_loadu_ps(b + i);
        __m512 diff = _mm512_sub_ps(va, vb);
        sum_vec = _mm512_fmadd_ps(diff, diff, sum_vec);
    }

    float total_sum = _mm512_reduce_add_ps(sum_vec);
    if (i < size) {
        unsigned int remaining = size - i;
        __mmask16 mask = (__mmask16)((1U << remaining) - 1);

        __m512 va = _mm512_maskz_loadu_ps(mask, a + i);
        __m512 vb = _mm512_maskz_loadu_ps(mask, b + i);

        __m512 diff = _mm512_sub_ps(va, vb);
        __m512 sq_diff = _mm512_mul_ps(diff, diff);
```

```
        total_sum += _mm512_reduce_add_ps(sq_diff);
    }

    return std::sqrt(total_sum);
}
```

Метод find_simd аналогичен наивной реализации по структуре, но вызывает оптимизированную функцию расчета расстояния. Важно отметить, что для передачи данных в интринсики используются сырые указатели, полученные из std::vector.

Листинг 3.4. Основной поиск с использованием SIMD

```
std::vector<int> KnnSearcher::find_simd(
    const std::vector<std::vector<float>>& dataset,
    const std::vector<float>& query,
    int k
) {
    std::vector<std::pair<float, int>> distances;
    distances.reserve(dataset.size());

    const float* query_ptr = query.data();
    size_t dim = query.size();

    for (size_t i = 0; i < dataset.size(); ++i) {
        float dist = get_euclidean_distance_avx512(dataset[i].data(),
query_ptr, dim);
        distances.push_back({dist, (int)i});
    }
    std::sort(distances.begin(), distances.end());

    std::vector<int> result;
    result.reserve(k);
    for (int i = 0; i < k && i < (int)distances.size(); ++i) {
        result.push_back(distances[i].second);
    }
    return result;
}
```

3.2. Алгоритма размытия изображения по Гауссу

Размытие по Гауссу — это метод обработки изображений, используемый для уменьшения шума и детализации изображения. Математически операция представляет собой свертки исходного изображения с фильтром (ядром), значения которого описываются функцией нормального распределения (Гауссианой).

В дискретном виде для цифровых изображений это реализуется через матрицу (ядро свертки), которая "скользит" по изображению. Значение каждого пикселя результирующего изображения вычисляется как взвешенная сумма значений соседних пикселей, где веса определяются ядром Гаусса.

В проекте реализован класс *ImageConvolver*, который выполняет свертку RGB-изображения с произвольным ядром. "Наивная" реализация *process_default* обрабатывает каждый пиксель последовательно, используя вложенные циклы для прохода по ядру свертки.

Особенности реализации:

1. Формат данных: Изображение хранится как одномерный массив байтов (unsigned char), где каждые 4 байта кодируют один пиксель (RGBA).
2. Обработка границ: Для упрощения алгоритма область свертки ограничена так, чтобы ядро всегда полностью находилось внутри изображения. Граничные пиксели, для которых ядро выходит за пределы, просто копируются из исходного изображения без изменений.
3. Арифметика с плавающей точкой: Вычисления взвешенной суммы производятся в *float* для точности, а затем результат приводится к диапазону [0, 255] с помощью std::clamp.

Листинг 3.5. Наивная реализация операции размытия по Гауссу

```
std::vector<unsigned char> ImageConvolver::process_default(const unsigned
char* img_in, int w, int h) {
    if (!img_in) return {};

    std::vector<unsigned char> img_out(w * h * 4);

    int kHalfW = m_kW / 2;
    int kHalfH = m_kH / 2;

    for (int y = kHalfH; y < h - kHalfH; ++y) {
        for (int x = kHalfW; x < w - kHalfW; ++x) {

            float sumR = 0.f, sumG = 0.f, sumB = 0.f;

            for (int ky = -kHalfH; ky <= kHalfH; ++ky) {
                for (int kx = -kHalfW; kx <= kHalfW; ++kx) {
                    int sx = x + kx;
                    int sy = y + ky;
                    int srcIdx = (sy * w + sx) * 4;

                    int kkx = kx + kHalfW;
                    int kky = ky + kHalfH;
                    float wgt = m_kernel[kky * m_kW + kkx];

                    sumR += wgt * img_in[srcIdx + 0];
                    sumG += wgt * img_in[srcIdx + 1];
                    sumB += wgt * img_in[srcIdx + 2];
                }
            }
```

```
            int dstIdx = (y * w + x) * 4;
            img_out[dstIdx + 0] = (unsigned char)std::clamp(sumR, 0.f, 255.f);
            img_out[dstIdx + 1] = (unsigned char)std::clamp(sumG, 0.f, 255.f);
            img_out[dstIdx + 2] = (unsigned char)std::clamp(sumB, 0.f, 255.f);
            img_out[dstIdx + 3] = img_in[dstIdx + 3];
        }
    }

    for (int y = 0; y < h; ++y) {
        for (int x = 0; x < w; ++x) {
            if (y < kHalfH || y >= h - kHalfH ||
                x < kHalfW || x >= w - kHalfW) {
                int idx = (y * w + x) * 4;
                img_out[idx + 0] = img_in[idx + 0];
                img_out[idx + 1] = img_in[idx + 1];
                img_out[idx + 2] = img_in[idx + 2];
                img_out[idx + 3] = img_in[idx + 3];
            }
        }
    }

    return img_out;
}
```

Этот метод является понятным и надежным, но страдает от низкой производительности из-за большого количества ветвлений во внутренних циклах и отсутствия параллелизма, что делает его идеальным кандидатом для оптимизации.

Векторизация процесса свертки изображений позволяет обрабатывать несколько пикселей одновременно, что значительно ускоряет работу фильтра. В данной реализации используется стратегия обработки групп по 4 пикселя (16 байт данных) за одну итерацию, так как каждый пиксель состоит из 4 компонентов (RGBA), и 4 пикселя 4 канала = 16 значений *float*, что идеально укладывается в 512-битный регистр AVX-512.

Для реализации свертки используются специфические инструкции для работы с целыми числами (для загрузки пикселей) и числами с плавающей точкой (для вычислений):

_mm_loadu_si128 – загружает 128 бит (16 байт) данных. В данном контексте — это ровно 4 пикселя формата RGBA.

_mm512_cvtepu8_epi32 – расширяет 8-битные беззнаковые целые до 32-битных целых. Позволяет перевести упакованные пиксели в формат, пригодный для конвертации во *float*.

_mm512_cvtepi32_ps – конвертирует 32-битные целые числа в числа с плавающей точкой для выполнения арифметических операций.

_mm512_set1_ps – создает вектор, в котором все 16 элементов равны одному заданному значению. Используется для размножения текущего веса ядра на все каналы всех 4 пикселей.

_mm512_fmadd_ps – Fused Multiply-Add: выполняет a * b + c. Ключевая инструкция свертки: умножает значения пикселей на вес ядра и прибавляет к аккумулятору.

_mm512_cvtps_epi32 – обратная конвертация результата из *float* в 32-битные целые (с округлением).

_mm512_cvtusepi32_epi8 – мощная инструкция сжатия и насыщения. Конвертирует 32-битные целые обратно в 8-битные unsigned char. Она автоматически выполняет clamp, избавляя от необходимости писать ручные проверки.

_mm_storeu_si128 – сохраняет результат (128 бит / 4 пикселя) обратно в память.

_mm256_zeroupper – служебная инструкция для очистки верхних частей YMM регистров при переходе от AVX кода к SSE (необязательна на чистом AVX-512, но полезна для совместимости).

Функция *process_simd* обрабатывает изображение блоками по 4 пикселя по горизонтали.

1. Загрузка и распаковка: 4 пикселя (16 байт) загружаются в __m128i, затем расширяются до __m512i и конвертируются в __m512 (*float*).

2. Параллельная свертка: Внутренний цикл проходит по ядру свертки. Вес ядра "размножается" на весь вектор (_mm512_set1_ps), и происходит одновременное умножение и сложение для 16 каналов.

4. Упаковка и сохранение: Результат (сумма) конвертируется обратно в байты с автоматическим насыщением, что заменяет ручной std::clamp.

Листинг 3.6. Реализация операции размытия по Гауссу с использованием SIMD

```
std::vector<unsigned char> ImageConvolver::process_SIMD(const unsigned char*
img_in, int w, int h) {
    if (!img_in) return {};

    std::vector<unsigned char> img_out(w * h * 4);

    int kHalfW = m_kW / 2;
    int kHalfH = m_kH / 2;

    int xEnd = w - kHalfW;
    int xSimdEnd = kHalfW + ((xEnd - kHalfW) / 4) * 4;

    for (int y = kHalfH; y < h - kHalfH; ++y) {

        int x = kHalfW;

        for (; x < xSimdEnd; x += 4) {
            __m512 vSum = _mm512_setzero_ps();

            for (int ky = -kHalfH; ky <= kHalfH; ++ky) {
                for (int kx = -kHalfW; kx <= kHalfW; ++kx) {
                    float wgt = m_kernel[(ky + kHalfH) * m_kW + (kx +
kHalfW)];
                    __m512 vWgt = _mm512_set1_ps(wgt);
                    int srcIdx = ((y + ky) * w + (x + kx)) * 4;
                    __m128i vPx8 = _mm_loadu_si128(reinterpret_cast<const
__m128i*>(&img_in[srcIdx]));
                    __m512i vPx32 = _mm512_cvtepu8_epi32(vPx8);
                    __m512 vPxFloat = _mm512_cvtepi32_ps(vPx32);
                    vSum = _mm512_fmadd_ps(vPxFloat, vWgt, vSum);
                }
            }

            __m512i vRes32 = _mm512_cvtps_epi32(vSum);

            __m128i vRes8 = _mm512_cvtusepi32_epi8(vRes32);

            int dstIdx = (y * w + x) * 4;
            _mm_storeu_si128(reinterpret_cast<__m128i*>(&img_out[dstIdx]),
vRes8);

            img_out[dstIdx + 3]  = img_in[dstIdx + 3];
            img_out[dstIdx + 7]  = img_in[dstIdx + 7];
            img_out[dstIdx + 11] = img_in[dstIdx + 11];
            img_out[dstIdx + 15] = img_in[dstIdx + 15];
        }

        for (; x < xEnd; ++x) {
            float sumR = 0.f, sumG = 0.f, sumB = 0.f;
            for (int ky = -kHalfH; ky <= kHalfH; ++ky) {
                for (int kx = -kHalfW; kx <= kHalfW; ++kx) {
                    int srcIdx = ((y + ky) * w + (x + kx)) * 4;
                    float wgt = m_kernel[(ky + kHalfH) * m_kW + (kx +
kHalfW)];
                    sumR += wgt * img_in[srcIdx + 0];
                    sumG += wgt * img_in[srcIdx + 1];
                    sumB += wgt * img_in[srcIdx + 2];
                }
            }
            int dstIdx = (y * w + x) * 4;
            img_out[dstIdx + 0] = (unsigned char)std::clamp(sumR, 0.f, 255.f);
```

```
            img_out[dstIdx + 1] = (unsigned char)std::clamp(sumG, 0.f, 255.f);
            img_out[dstIdx + 2] = (unsigned char)std::clamp(sumB, 0.f, 255.f);
            img_out[dstIdx + 3] = img_in[dstIdx + 3];
        }
    }

    _mm256_zeroupper();

    for (int y = 0; y < h; ++y) {
        for (int x = 0; x < w; ++x) {
            if (y < kHalfH || y >= h - kHalfH || x < kHalfW || x >= w -
kHalfW) {
                int idx = (y * w + x) * 4;
                img_out[idx + 0] = img_in[idx + 0];
                img_out[idx + 1] = img_in[idx + 1];
                img_out[idx + 2] = img_in[idx + 2];
                img_out[idx + 3] = img_in[idx + 3];
            }
        }
    }

    return img_out;
}
```

Такой подход позволяет радикально сократить количество инструкций, выполняя тяжелую арифметику с плавающей точкой сразу для 16 значений за такт, а также эффективно используя пропускную способность памяти за счет 128-битных чтений.

4. Тестирование

Тестирование будет происходить с использованием библиотеки Google Benchmark, рассмотренной в прошлой лабораторной, поэтому код подробно приводиться не будет.

Также для тестирования используются аналогичные с первой лабораторной скрипты для подготовки окружения, которые также не будут приводиться.

Для алгоритма поиска K-ближайших в ходе тестирования меняются параметры длины вектора, размер датасета, а также число K. Итоговые графики приведены ниже:

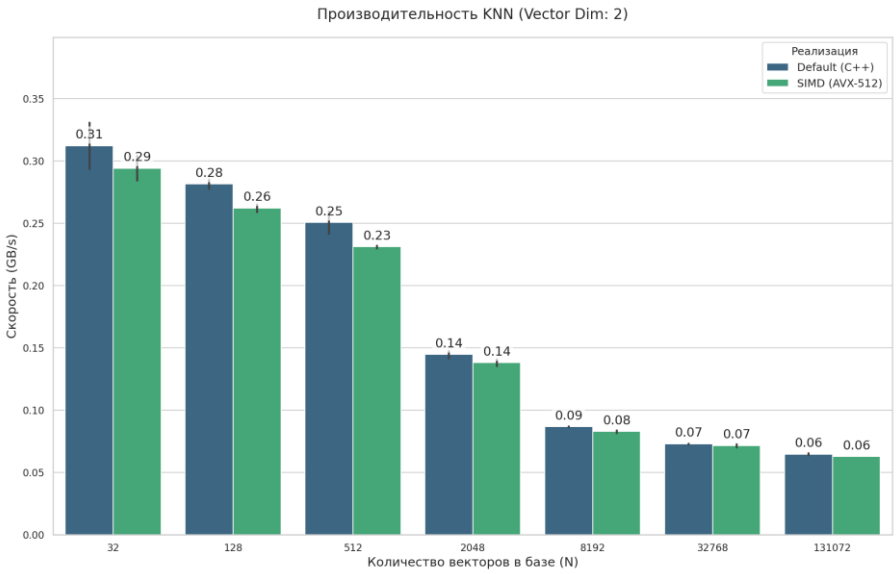

Рис. 4.1. Алгоритм К-ближайших, вектор длинной 2.

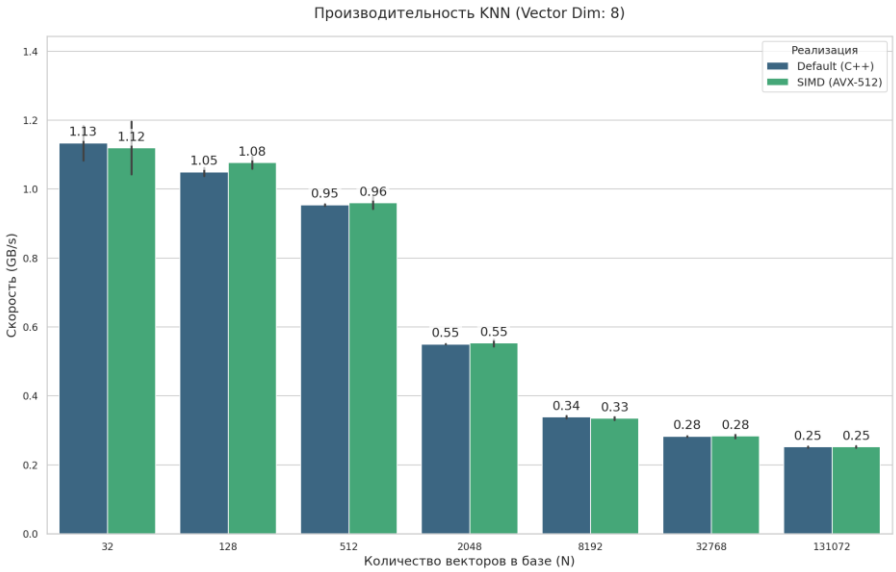

Рис. 4.2. Алгоритм К-ближайших, вектор длинной 8.

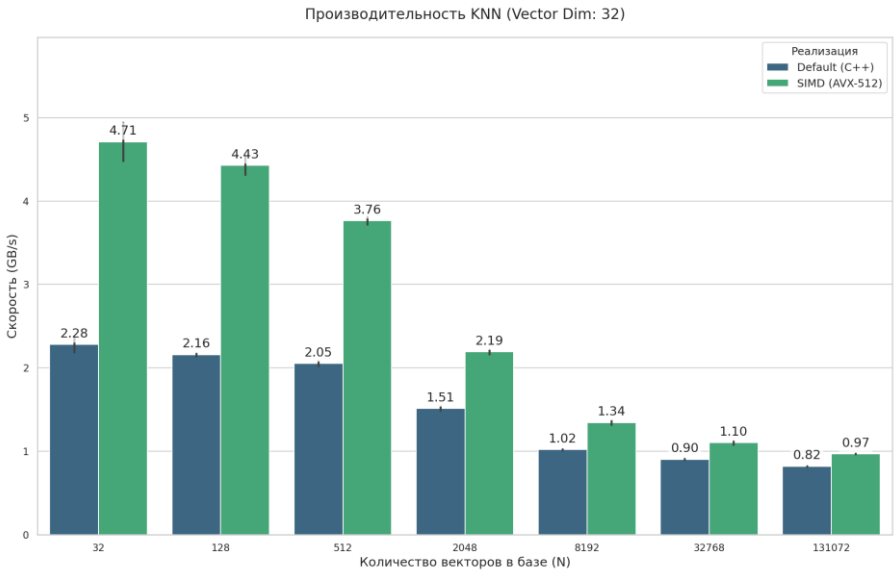

Рис. 4.3. Алгоритм К-ближайших, вектор длинной 32.

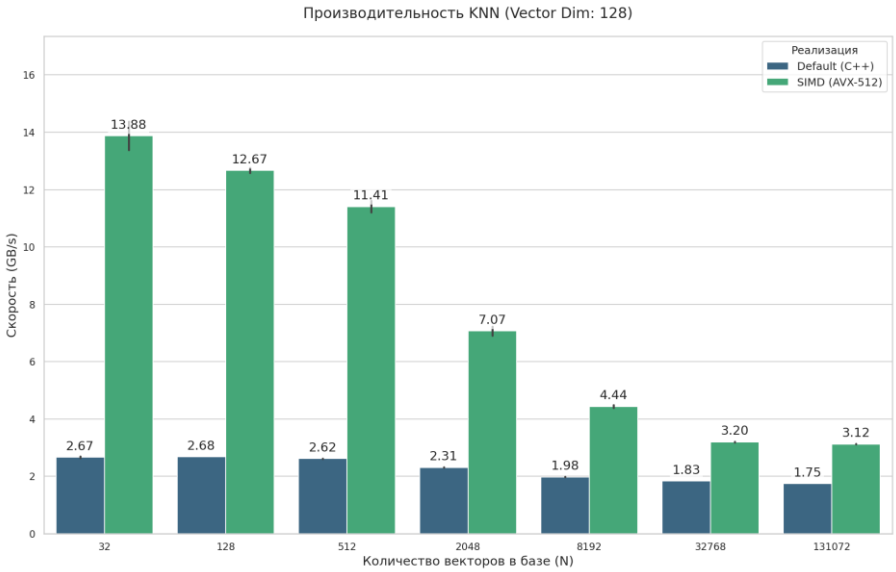

Рис. 4.4. Алгоритм К-ближайших, вектор длинной 128.

При малом значении размерности вектора мы практически не получаем никакого прироста, однако при увеличении мы видим значительный рост. Это связано с тем, что количество операций вычисления расстояния (именно то, что мы оптимизировали, используя SIMD инструкции) при малом значении размерности вектора сопоставимы, однако при увеличении размера вектора SIMD реализация несомненно выигрывает. Падение же скорости обусловлено кеш-промахами.

Для алгоритма размытия по Гауссу изменяются два параметра, а именно размер окна и размер фотографии:

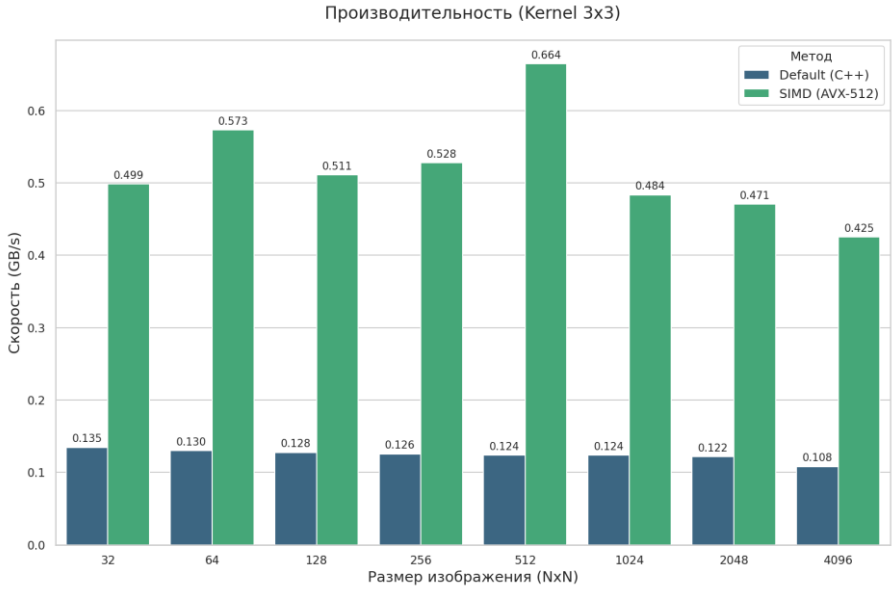

Рис. 4.5. Алгоритм размытия по Гауссу, окно размером 3 на 3.

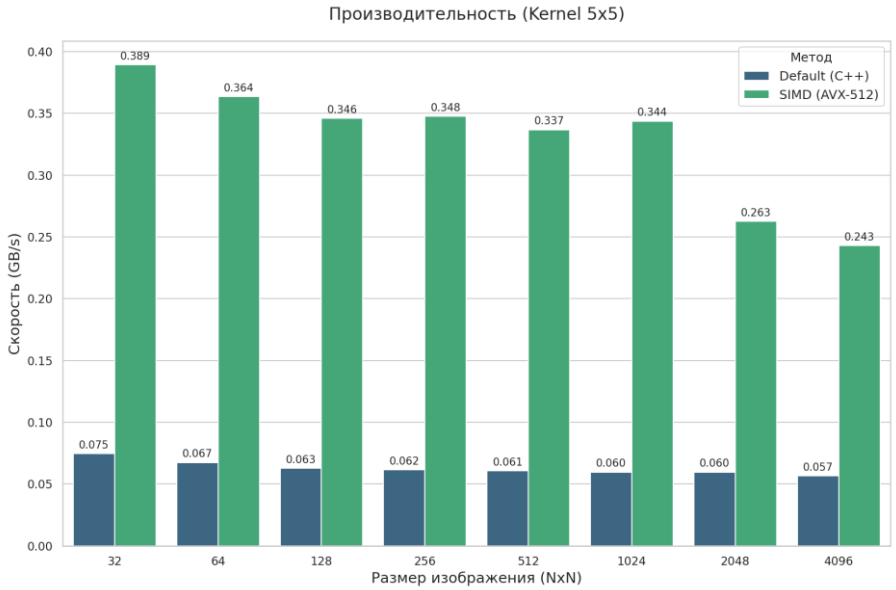

Рис. 4.6. Алгоритм размытия по Гауссу, окно размером 5 на 5.

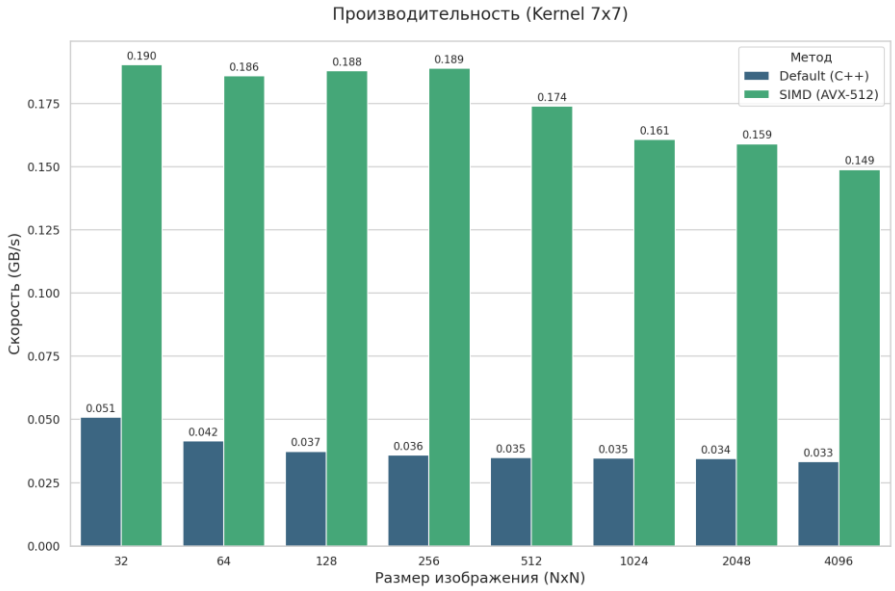

Рис. 4.7. Алгоритм размытия по Гауссу, окно размером 7 на 7.

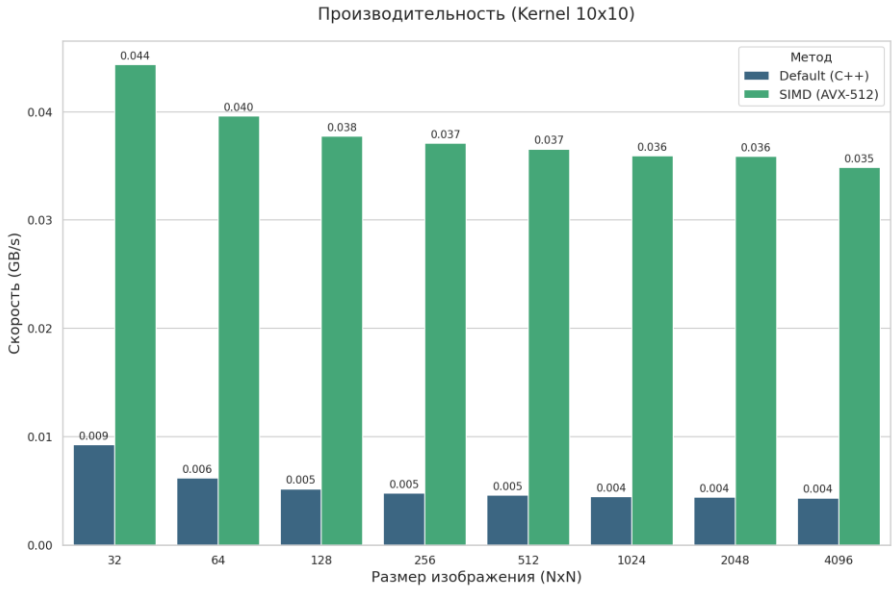

Рис. 4.8. Алгоритм размытия по Гауссу, окно размером 10 на 10.

Как видно по графикам, независимо от того, какого размера было окно и размер фотографии, мы получили прирост более чем в 4 раз, это связано с тем, что независимо от размера картинки, число операций практически одинаковое, однако реализация с SIMD инструкциями одновременно считает значения для 4 пикселей, а обычный алгоритм лишь для 1.

5. Вывод

В ходе выполнения лабораторной работы были успешно реализованы и исследованы алгоритмы поиска k ближайших соседей и размытия изображения по Гауссу. Для каждого из них были разработаны последовательная версия и оптимизированная версия с использованием инструкций AVX-512, после чего проведен сравнительный анализ их производительности.

Анализ результатов для алгоритма k-NN показал, что эффективность векторизации напрямую зависит от размерности обрабатываемых векторов. При малой длине вектора накладные расходы на подготовку данных практически нивелируют выигрыш, однако с ростом размерности (до 128 и выше) наблюдается существенное ускорение за счет параллельной обработки 16 элементов. При этом на больших объемах данных ограничивающим фактором становится подсистема памяти и промахи кэша.

Алгоритм размытия по Гауссу, напротив, продемонстрировал стабильный прирост производительности более чем в 4 раза независимо от параметров теста. Этого удалось достичь благодаря обработке групп по 4 пикселя, что позволяет полностью утилизировать 512-битные регистры для сложных вычислений с плавающей точкой. Дополнительную эффективность обеспечило использование инструкций насыщения, заменивших медленные операции ветвления при проверке границ цвета.

Таким образом, работа подтвердила высокую эффективность применения SIMD-инструкций для задач, поддающихся параллелизации на уровне данных, и позволила закрепить практические навыки низкоуровневой оптимизации программного обеспечения.

6. Ссылки

1. DafterT. CPP_Lang_lab_2 [Электронный ресурс]. URL: https://github.com/DafterT/CPP_lang_lab_2 (дата обращения: 24.11.2025).