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

Отчет по лабораторной работе №3

по курсу «Программирование на C++»

Тема: «Многопоточность в C++»

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

Руководитель _____ А. Д. Жданов
(подпись)

25 декабря 2025 г.

Санкт-Петербург
2025

Оглавление

1. Цель работы

Получение практических навыков проектирования, реализации и исследования алгоритмов, использующих распараллеливание вычислений на уровне потоков. В ходе выполнения работы предусматривается разработка собственной реализации пула потоков с интерфейсом постановки задач, поддерживающего заданное число рабочих потоков, очередь задач и корректное завершение работы с ожиданием выполнения всех поставленных задач, с применением примитивов синхронизации на основе мьютексов и условных переменных. Далее требуется разработать многопоточную версию выбранного варианта второй лабораторной работы (без SIMD) и провести сравнительный анализ производительности последовательной, SIMD- и многопоточной реализаций, включая оценку ускорения и накладных расходов, связанных с распараллеливанием кода.

2. План работы

1. Спроектировать и реализовать собственную реализацию пула потоков с возможностью задания числа рабочих потоков, очередью задач и интерфейсом dispatch_task, возвращающим std::future результата.
2. Реализовать синхронизацию внутри пула потоков с использованием мьютексов и условных переменных, обеспечив корректную обработку постановки задач и пробуждения рабочих потоков.
3. Обеспечить корректное завершение работы пула потоков: корректную остановку потоков и ожидание выполнения всех задач, находящихся в очереди, в деструкторе.
4. Выбрать один из вариантов второй лабораторной работы и разработать его многопоточную версию с использованием реализованного пула потоков (без SIMD).
5. Провести сравнительное тестирование производительности последовательной, многопоточной и SIMD-версий выбранного

алгоритма, оценить ускорение и накладные расходы на распараллеливание.

3. Ход работы

3.1. Реализация пула потоков

Реализованный в ходе задачи пул потоков имеет следующий заголовочный файл:

```
class ThreadPool {
public:
    explicit ThreadPool(size_t num_threads = 0);
    ThreadPool(const ThreadPool&) = delete;
    ThreadPool& operator=(const ThreadPool&) = delete;
    ~ThreadPool();

    template<typename Fn, typename T = typename std::invoke_result_t<Fn>>
    std::future<T> dispatch_task(Fn&& f);
    size_t get_thread_count() const;
    size_t get_queue_size() const;

private:
    struct Task {
        std::function<void()> func;  ///< Функция для выполнения
        Task() = default;
        explicit Task(std::function<void()> f) : func(std::move(f)) {}
    };
    void worker_thread();
    void stop_all_threads();

    std::vector<std::thread> m_workers;        ///< Вектор рабочих потоков
    std::queue<Task> m_tasks;                  ///< Очередь задач
    mutable std::mutex m_queue_mutex; ///< Мьютекс для синхронизации доступа к очереди
    std::condition_variable m_condition; ///< Переменная для уведомления потоков
    std::atomic<bool> m_stop{false};           ///< Флаг остановки пула потоков
};
```

В данной реализации пользователю предоставляется интерфейс для создания пула потоков, а также возможность добавить задачу в пул потоков. Дополнительно реализованы функции для контроля очереди и числа потоков.

В приватной же области созданы создана структура для хранения задачи, а также функция для выполнения потоком. Также имеется функция для остановки рабочих.

Из переменных присутствует вектор рабочих потоков, очередь задач, а также мьютекс для синхронизации доступа к очереди, именно он позволяет потокам корректно распределять между собой задачи. Дополнительно присутствует условная переменная, она уведомляет потоки о том, что произошло какое-то событие, например добавление задачи в очередь или остановка работы.

Дополнительно сделан флаг статуса, запущен пул потоков или нет. Если этот флаг false и вызвать добавление элемента в поток, то будет ошибка.

В детали реализации вдаваться смысла нет. Все достаточно очевидно, при добавлении задачи функция дожидается освобождения очереди, используя мьютекс, и кладет задачу в очередь, уведомляя потоки о событии.

Потоки в свою очередь ждут событие, забирают себе управление очередью, используя мьютекс, после чего запускают задачу, которую забрали.

3.2. Реализация Гауссова размытия с использованием пула потоков

Было решено сделать 2 реализации для алгоритма Гауссова размытия, используя потоки. Первый – создать каждому потоку свою задачу (выделить часть строк полноценной картинки) и послать их в очередь, потом отдельно проделать тоже самое с границами картинки.

Вторя же, реализация проще, в ней каждая строчка посылается в очередь, и любой поток забирает её из очереди и выполняет.

Если в первой реализации не предполагается заполнение очереди, все потоки сразу забирают себе задачу и работают над ней, то вторая, наоборот, забивает очередь и ждет, когда все будет выполнено.

Подробнее с кодом можно ознакомиться в приложении.

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

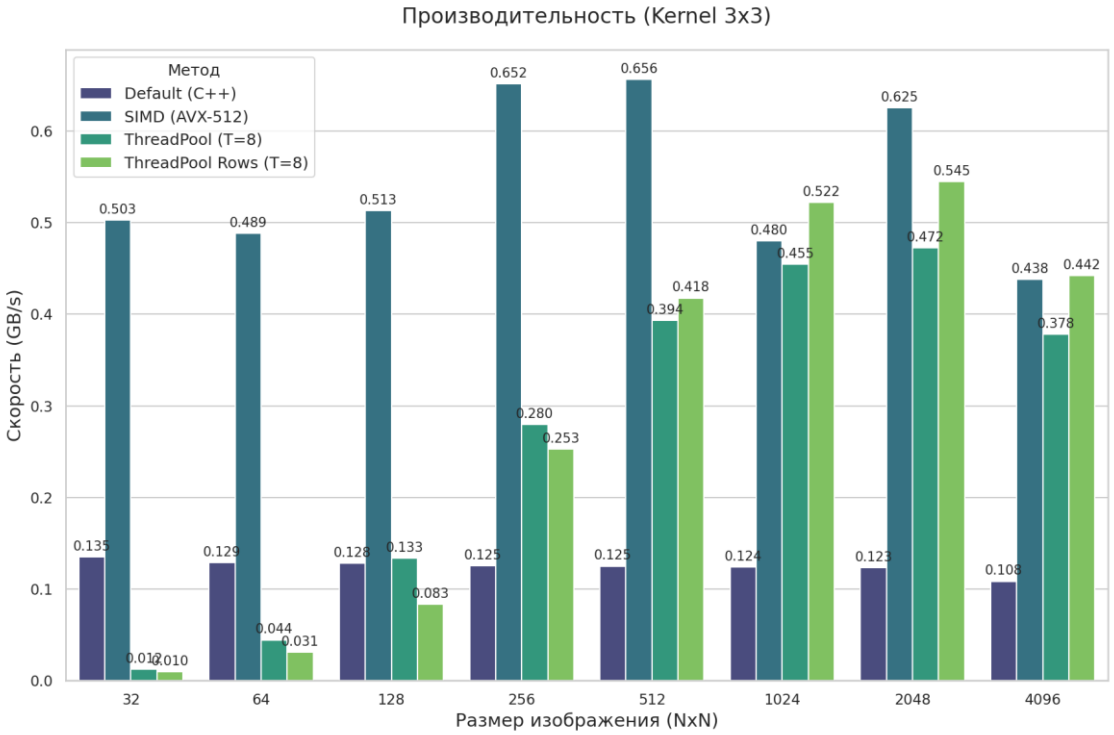

Рис. 4.1. Алгоритм размытия по Гауссу, окно размером 3 на 3.

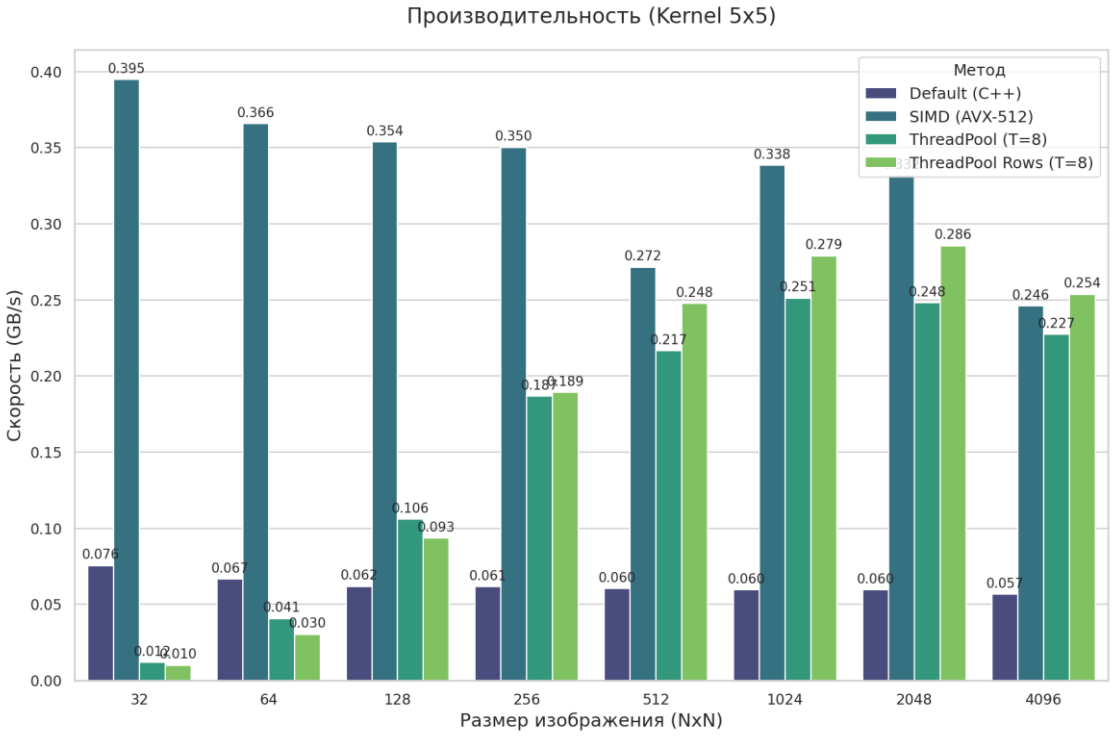

Рис. 4.2. Алгоритм размытия по Гауссу, окно размером 5 на 5.

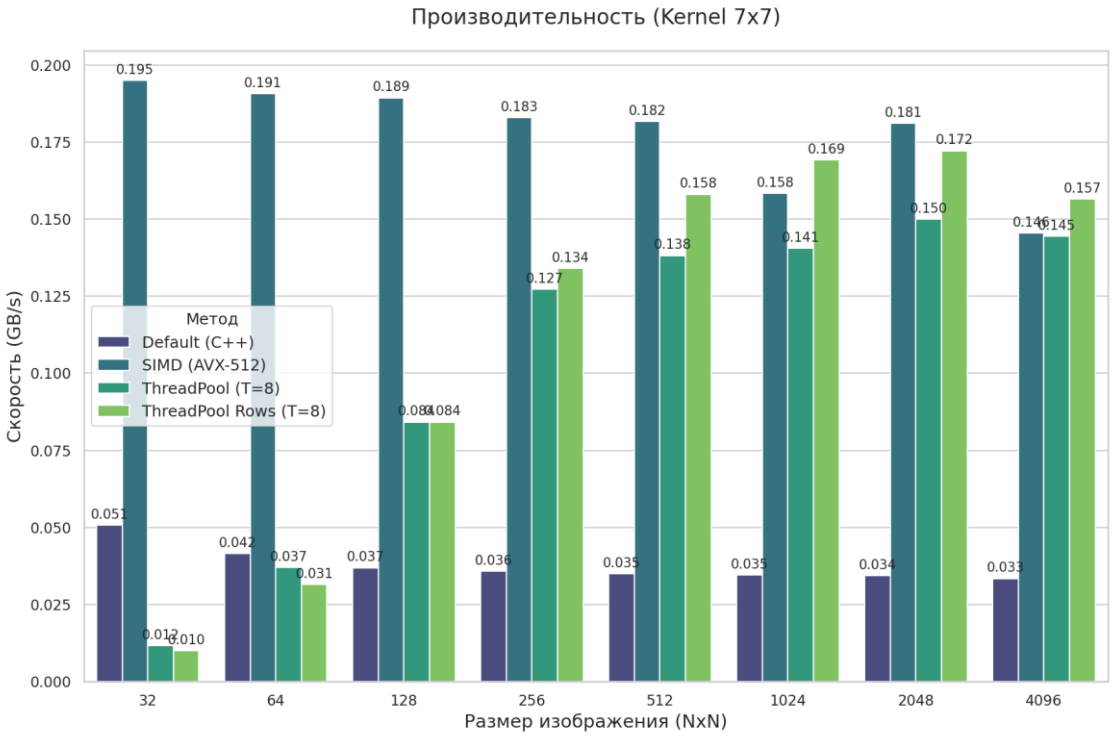

Рис. 4.3. Алгоритм размытия по Гауссу, окно размером 7 на 7.

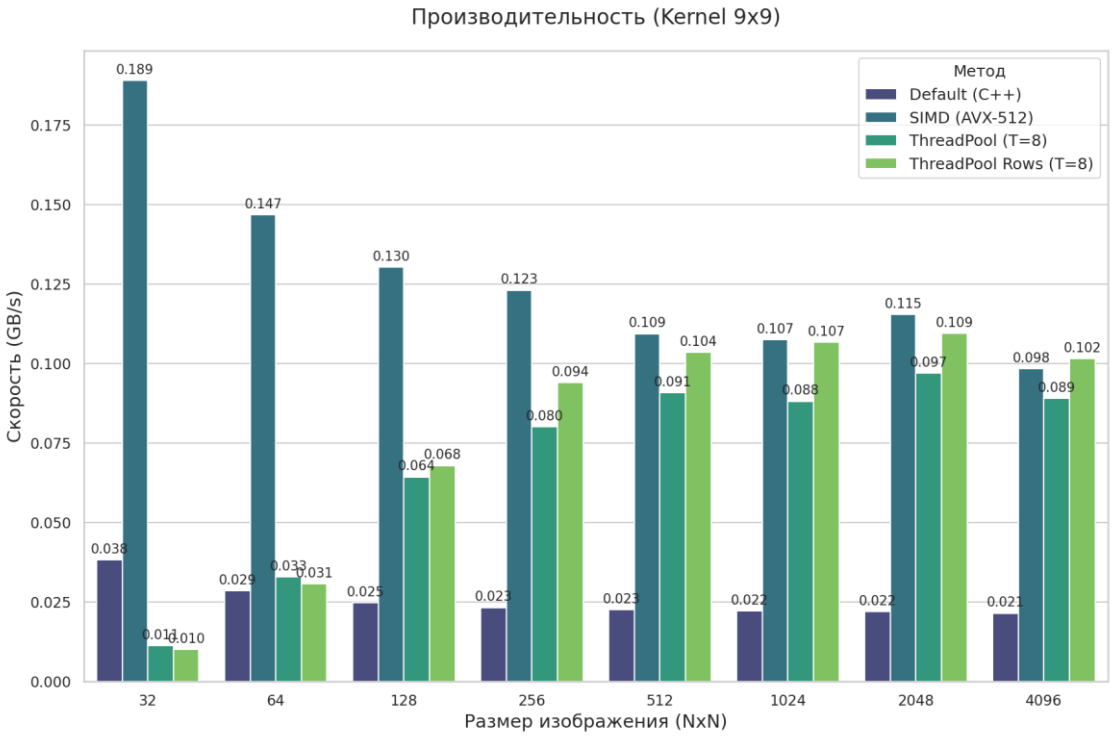

Рис. 4.4. Алгоритм размытия по Гауссу, окно размером 10 на 10.

В ходе тестирование чисто потоков было выбрано равным 8 для сравнения. Это количество потоков, которые поддерживаются процессором на тестируемой системе.

Как можно заметить, SIMD реализация практически везде выигрывает многопоточную, особенно при маленьком размере изображения. При увеличении размера, они примерно сравниваются.

Стоит отметить, что на маленьком изображении многопоточность показывает себя даже хуже, чем стандартная реализация «без ухищрений».

Стоит отдельно сравнить реализацию на разном числе потоков, предполагается, что реализация на 8 потоках (то, сколько аппаратно поддерживается целевой системой) должна показать себя лучше всего, но это необходимо проверить:

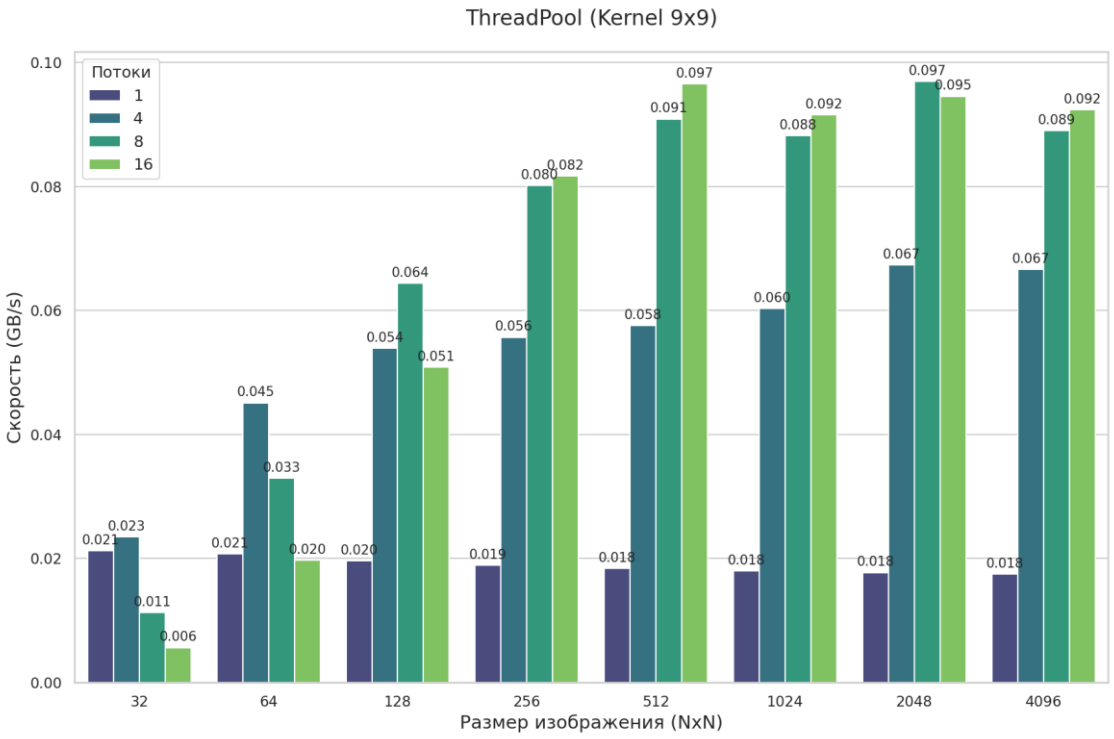

Рис. 4.5. Разное число потоков при ядре 9 на 9 на первой реализации.

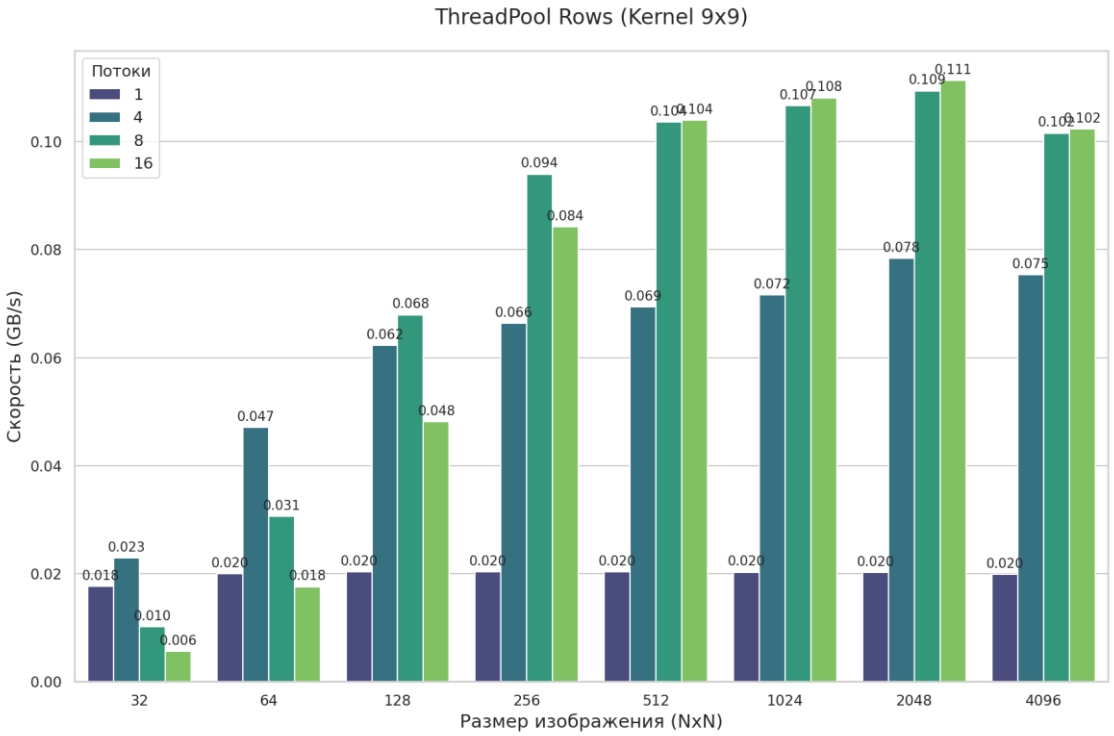

Рис. 4.6. Разное число потоков при ядре 9 на 9 на второй реализации.

Как можно заметить, реализация с 1 потоком, очевидно, показала себя хуже всего, причем даже хуже, чем обычная линейная реализация, но это и очевидно, т.к. требуется время на создание потока, передачу ему функции, также время на синхронизацию. 4 потока смотрятся лучше, однако также отстают от SIMD и 8 поточной реализации. 8 и 16 потоков на больших данных примерно сровнялись, очевидно, что это не дает выигрыша по скорости так как эти процессы просто конфликтуют между собой, равенство — это уже достаточно хороший результат, так как в худшем случае реализация могла стать медленнее.

Важно отметить тот факт, что на маленьких данных 8 и 16 потоков заметно проигрывают 1, проблема в том, что там основное время занимает именно что создание потока и синхронизация потоков, а не расчет значений. Убедимся в этом, обратившись к этому графику:

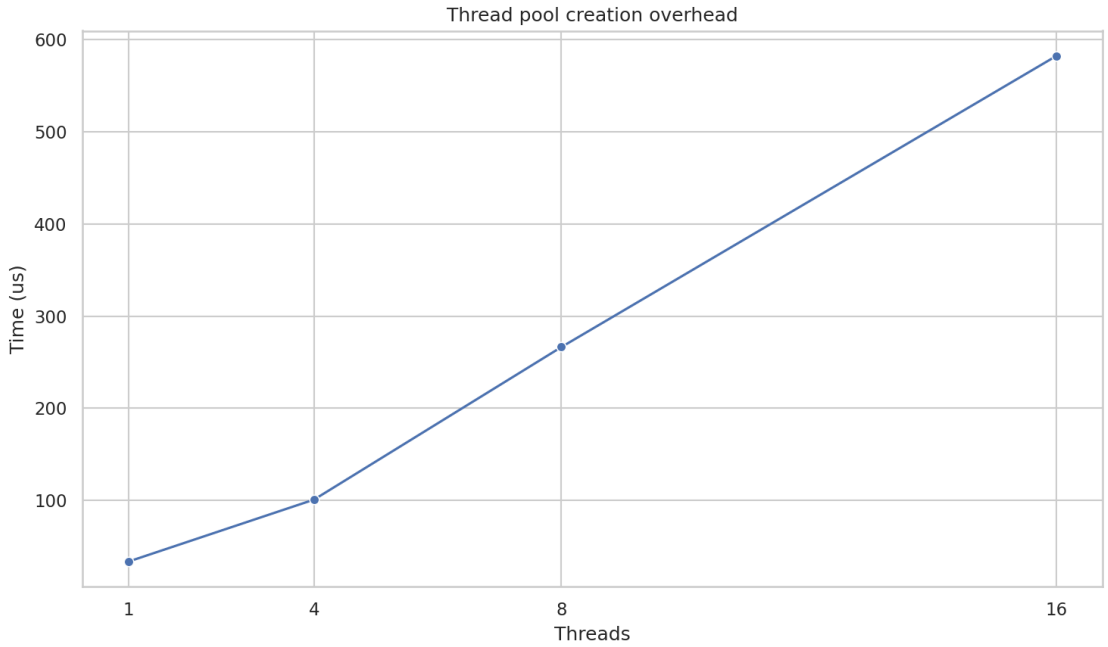

Рис. 4.7. Время на создание потоков.

Как можно заметить, чтоб создать 4 потока требуется 100 мкс, на 8 уже 250 мкс, а на 16 целых 600 мкс. В масштабах картинки 32 на 32, обработка которой занимает время порядка 10 мкс, создание 16 потоков — это невероятно долго, из-за этого и видна значительная просадка по времени, а на больших изображениях, где обработка часто превышает несколько секунд просадка практически незаметна, а вот прирост от многопоточности виден.

Лучше всего это демонстрируется на этом графике:

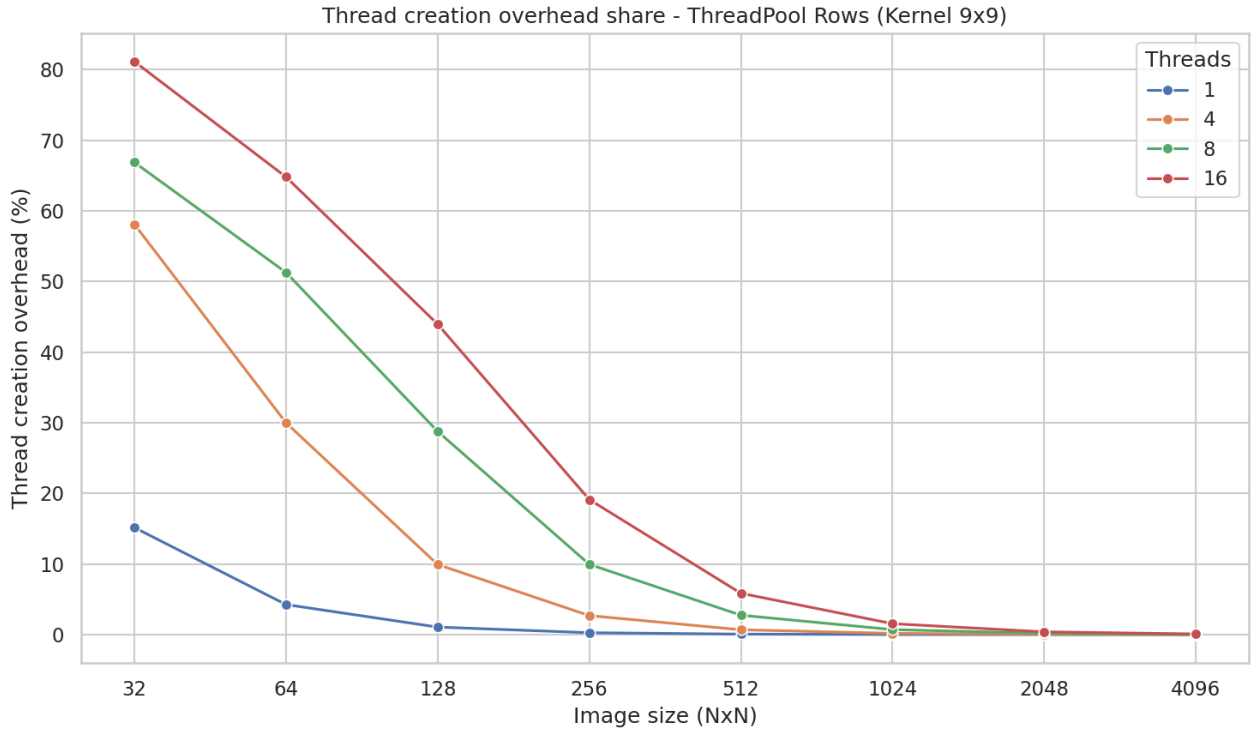

Рис. 4.8. Процент затрачиваемого времени на создание потоков.

Как можно заметить, для картинок до 128 на 128, время на создание потока занимало половину от времени выполнения. Это без учета времени на синхронизацию и тп. Однако для больших картинок это не проблема и процент становится сравнительно мал, относительно получаемого выигрыша.

5. Вывод

В ходе лабораторной работы была реализована собственная реализация пула потоков на языке C++ с использованием мьютексов и условных переменных, обеспечивающая корректную постановку и выполнение задач, а также корректное завершение работы. На его основе была разработана многопоточная версия алгоритма Гауссова размытия изображения. Проведённое тестирование показало, что многопоточность даёт выигрыш производительности на больших объёмах данных, однако на малых размерах изображений накладные расходы на создание и синхронизацию потоков могут превышать пользу от распараллеливания. Также установлено, что SIMD-реализация в большинстве случаев демонстрирует наилучшие результаты.

6. Ссылки

1. DafterT. CPP_Lang_lab_3 [Электронный ресурс]. URL: https://github.com/DafterT/CPP_lang_lab_3 (дата обращения: 25.12.2025).