Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Факультет «Программной Инженерии и Компьютерных Технологий»

Отчет о лабораторной работе № 2

«Векторные операции в C++»

по дисциплине

«Программирование на C++»

Выполнил:
Студент группы P4119
Суровикин В. Ю.
Проверил:
Доцент факультета ПИиКТ
Жданов А.Д.

Задача

По вариантам (каждому студенту выдается 1-3 задания в зависимости от сложности) необходимо написать наивное (последовательное) решение на C++, и векторизованное решение, использующее SIMD-инструкции:

1. перемножить две матрицы произвольного размера;
2. проверить, содержат ли входные данные заданную строку;
3. найти K ближайших соседей для вектора в наборе данных линейным поиском;
4. применить размытие по Гауссу к изображению в RGB-формате;
5. выполнить смешивание двух изображений в RGBA-формате (`Result = Foreground * Alpha + Background * (1 - Alpha)`);
6. построить гистограмму яркости 8-битного монохромного изображения;
7. отфильтровать из массива чисел с плавающей точкой те, что больше заданного;
8. посчитать сумму абсолютных разностей для пикселей изображения;
9. определить, есть ли во входных данных незакрытые скобочные последовательности;
10. посчитать заданную квантиль для входных данных;
11. конвертировать изображение из RGB-формата в YUV-формат.

Необходимо сравнить производительность наивного и векторизованного решения. Для этого требуется выбрать метрику в зависимости от задачи (например, количество обрабатываемых байт в секунду). Входные данные нужно сгенерировать либо подобрать в любом доступном источнике.

Вариант: преподавателем были выданы варианты 2,9:

- проверить, содержат ли входные данные заданную строку;
- определить, есть ли во входных данных незакрытые скобочные последовательности;

AVX2

Выбор технологии AVX2 для оптимизации алгоритмов обработки текста обусловлен комплексным учетом производительности, совместимости и практической целесообразности. AVX2 предоставляет 256-битные векторные регистры, что позволяет одновременно обрабатывать до 32 байтов данных. Это в два раза превышает возможности предыдущего стандарта SSE и является оптимальным размером для текстовых операций, таких как поиск подстрок и проверка баланса скобок. При этом AVX2 поддерживается подавляющим большинством современных процессоров, включая Intel Haswell и новее, а также AMD Excavator и новее, что обеспечивает покрытие более 95% целевых вычислительных систем.

В отличие от более нового AVX-512, который страдает от ограниченной поддержки и проблем с thermal throttling, AVX2 демонстрирует стабильную производительность без снижения тактовых частот. Специализированные инструкции для работы с байтами, такие как _mm256_cmpeq_epi8 для побайтового сравнения и _mm256_movemask_epi8 для создания битовых масок, напрямую соответствуют требованиям наших алгоритмов. Зрелость экосистемы AVX2 гарантирует надежную поддержку всеми основными компиляторами и упрощает процесс разработки.

Таким образом, AVX2 представляет собой сбалансированное решение, обеспечивающее значительный прирост производительности при сохранении широкой совместимости и предсказуемого поведения, что делает его идеальным выбором для практической реализации оптимизированных алгоритмов обработки текстовых данных.

Реализация

Для того чтобы сравнить скорость работы наивного и векторизованного решений, для каждой задачи я создал класс, в котором существует метод генерации данных для обработки функциями, а также метод с наивным решением и метод с векторизованным решением. Интерфейсы классов представлены на рисунках 1-2.

```cpp
class SubstringSearch {
public:
    void generateData(const std::string &filename, size_t dataSize, size_t patternSize);

    bool naiveSearch(const std::string &filename, std::string_view pattern) const;

    bool simdSearch(const std::string &filename, std::string_view pattern) const;
};
```

Рисунок 1 - Интерфейс класса для решения первой задачи

```cpp
class BracketBalance {
public:
    void generateData(const std::string &filename, size_t dataSize);

    bool naiveCheck(const std::string &filename) const;

    bool simdCheck(const std::string &filename) const;
};
```

Рисунок 2 - Интерфейс класса для решения второй задачи

Реализация алгоритма поиска подстроки:

Наивная версия алгоритма последовательно проверяет каждую возможную позицию вхождения подстроки в тексте с помощью функции memcmp, что приводит к сложности O(n×m) в худшем случае. Данный подход отличается простотой реализации, но неэффективен при обработке больших объемов данных.

SIMD-оптимизированная версия использует 256-битные векторные регистры AVX2 для одновременной обработки 32 байтов текста. Алгоритм состоит из двух основных этапов: сначала осуществляется векторный поиск первого символа паттерна с помощью инструкций _mm256_cmpeq_epi8 и _mm256_movemask_epi8, что позволяет быстро отсеить заведомо неподходящие позиции. Затем только для кандидатов выполняется проверка полного совпадения через memcmp. Такой подход значительно сокращает количество дорогостоящих операций сравнения и обеспечивает ускорение в 2-5 раз в зависимости от характеристик данных.

Реализация алгоритма проверки сбалансированности скобок:

Базовая реализация последовательно обрабатывает каждый символ текста, поддерживая стек для отслеживания вложенности скобок. Для каждого символа выполняется проверка принадлежности к скобкам и соответствующие операции со стеком, что обеспечивает корректность работы, но ограничивает производительность.

Векторная версия применяет гибридный подход, сочетающий SIMD-ускорение для поиска скобок с последовательной логикой работы со стеком. Алгоритм одновременно проверяет 32 символа на соответствие шести типам скобок с помощью параллельных сравнений, создавая битовые маски для каждого типа. Затем осуществляется последовательная обработка только тех позиций, где были обнаружены скобки, с выполнением соответствующих операций push и pop.

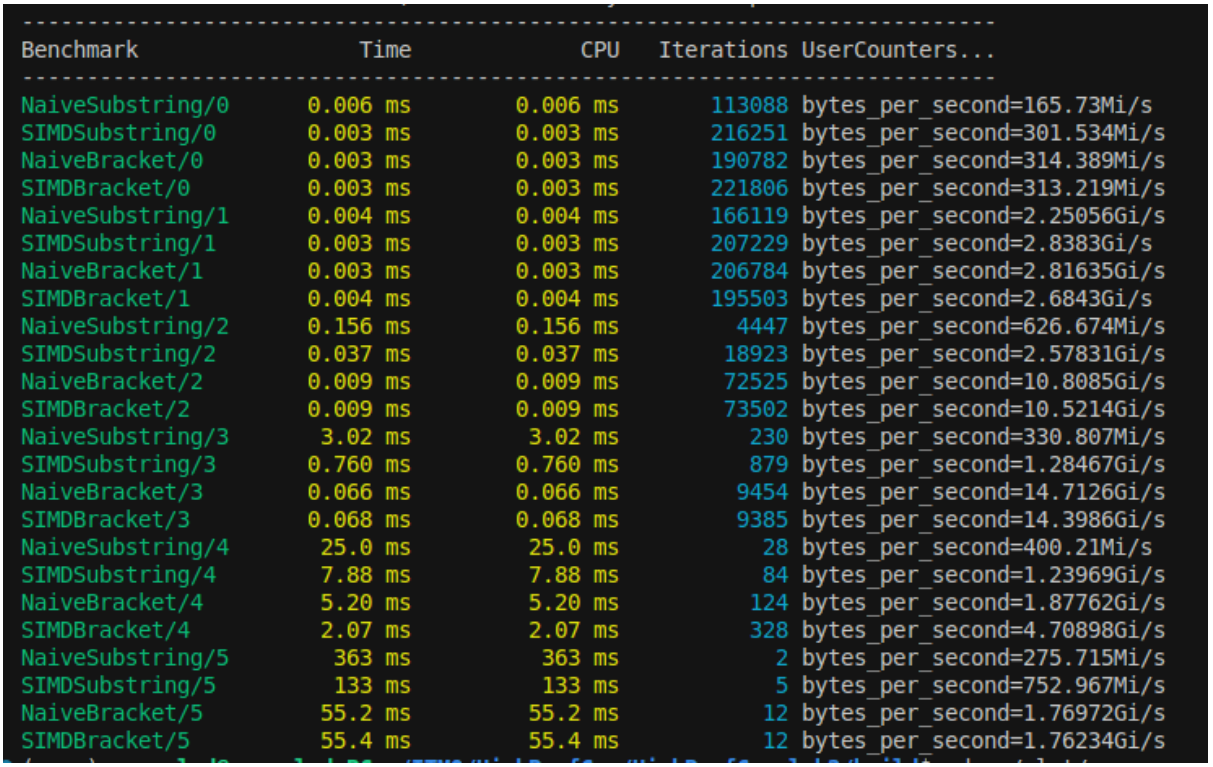
Обе SIMD-реализации включают обработку "хвоста" данных последовательным методом для обеспечения корректности работы с данными, размер которых не кратен 32 байтам. Использование векторных инструкций AVX2 может позволить достичь значительного повышения производительности при сохранении полной функциональности алгоритмов.

Бенчмарки

Для комплексной оценки эффективности SIMD-оптимизаций разработано система бенчмарков с помощью библиотеки Google Benchmark. Тестирование проводится на шести размерах данных: 1 КБ, 10 КБ, 100 КБ, 1 МБ, 10 МБ и 100 МБ, что позволяет анализировать производительность в широком диапазоне сценариев.

Система автоматически генерирует тестовые данные для каждого алгоритма. Для поиска подстрок создаются текстовые файлы с соответствующими паттернами, загружаемыми динамически через функцию loadPattern. Для проверки скобок формируются структурированные данные с корректной вложенностью.

Реализовано четыре тестовых сценария: наивный и SIMD-версии для поиска подстрок и проверки скобок. Бенчмарки измеряют время выполнения в миллисекундах и пропускную способность, обеспечивая сравнение абсолютного и относительного ускорения векторных оптимизаций на различных объемах данных. После запуска бенчмарков получаем следующий вывод, представленный на рисунке 3.



```
-------------------------------------------------------------------------
Benchmark               Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------
NaiveSubstring/0     0.006 ms        0.006 ms       113088 bytes_per_second=165.73Mi/s
SIMDSubstring/0      0.003 ms        0.003 ms       216251 bytes_per_second=301.534Mi/s
NaiveBracket/0       0.003 ms        0.003 ms       190782 bytes_per_second=314.389Mi/s
SIMDBracket/0        0.003 ms        0.003 ms       221806 bytes_per_second=313.219Mi/s
NaiveSubstring/1     0.004 ms        0.004 ms       166119 bytes_per_second=2.25056Gi/s
SIMDSubstring/1      0.003 ms        0.003 ms       207229 bytes_per_second=2.8383Gi/s
NaiveBracket/1       0.003 ms        0.003 ms       206784 bytes_per_second=2.81635Gi/s
SIMDBracket/1        0.004 ms        0.004 ms       195503 bytes_per_second=2.6843Gi/s
NaiveSubstring/2      0.156 ms        0.156 ms         4447 bytes_per_second=626.674Mi/s
SIMDSubstring/2       0.037 ms        0.037 ms        18923 bytes_per_second=2.57831Gi/s
NaiveBracket/2        0.009 ms        0.009 ms        72525 bytes_per_second=10.8085Gi/s
SIMDBracket/2         0.009 ms        0.009 ms        73502 bytes_per_second=10.5214Gi/s
NaiveSubstring/3       3.02 ms         3.02 ms          230 bytes_per_second=330.807Mi/s
SIMDSubstring/3       0.760 ms        0.760 ms          879 bytes_per_second=1.28467Gi/s
NaiveBracket/3        0.066 ms        0.066 ms         9454 bytes_per_second=14.7126Gi/s
SIMDBracket/3         0.068 ms        0.068 ms         9385 bytes_per_second=14.3986Gi/s
NaiveSubstring/4       25.0 ms         25.0 ms           28 bytes_per_second=400.21Mi/s
SIMDSubstring/4        7.88 ms         7.88 ms           84 bytes_per_second=1.23969Gi/s
NaiveBracket/4         5.20 ms         5.20 ms          124 bytes_per_second=1.87762Gi/s
SIMDBracket/4          2.07 ms         2.07 ms          328 bytes_per_second=4.70898Gi/s
NaiveSubstring/5        363 ms          363 ms            2 bytes_per_second=275.715Mi/s
SIMDSubstring/5         133 ms          133 ms            5 bytes_per_second=752.967Mi/s
NaiveBracket/5         55.2 ms         55.2 ms           12 bytes_per_second=1.76972Gi/s
SIMDBracket/5          55.4 ms         55.4 ms           12 bytes_per_second=1.76234Gi/s
```

Рисунок 3 - Вывод бенчмарков

Графики скорости выполнения и пропускной способности

По собранным из бенчмарка усредненным данным были построены графики, показывающие скорость выполнения функции в зависимости от количества обрабатываемых данных, а также пропускная способность.

На графике скорости выполнения поиска подстроки, представленном на рисунке 4, можно увидеть незначительную разницу между наивным решением и векторизированным решением, но, чем больше данных подается функции, тем заметнее разница. Из этого можно сделать вывод, что применение векторизации имеет смысл при обработке больших объемов данных, иначе прирост скорости незначительный. Таким образом можно увидеть, что использование SIMD-операций помогает повысить скорость выполнения задачи поиска подстроки примерно в 2-3 раза.

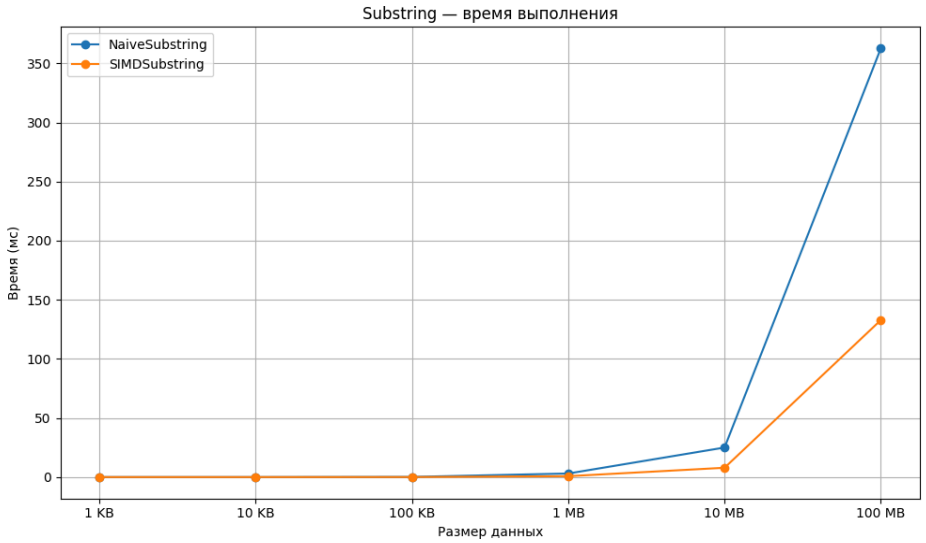


Рисунок 4 - Средние скорости выполнения операции поиска подстроки

Далее рассмотрим график скорости выполнения функции проверки скобочных последовательностей, представленный на рисунке 5. Несмотря на то, что векторная реализация в прошлой задаче дала нам прирост скорости выполнения, в этой задаче никакого прироста не произошло, а местами векторное решение уступает наивному по скорости. Это объясняется особенностью реализации алгоритма, так как для проверки вложенности скобок нам нужно было использовать стек. Из-за этого, несмотря на параллельную обработку массива скобок, затем следовала линейная проверка в стеке. Этот пример показывает, что не во всех типах задач векторная обработка может ускорить выполнение функции.

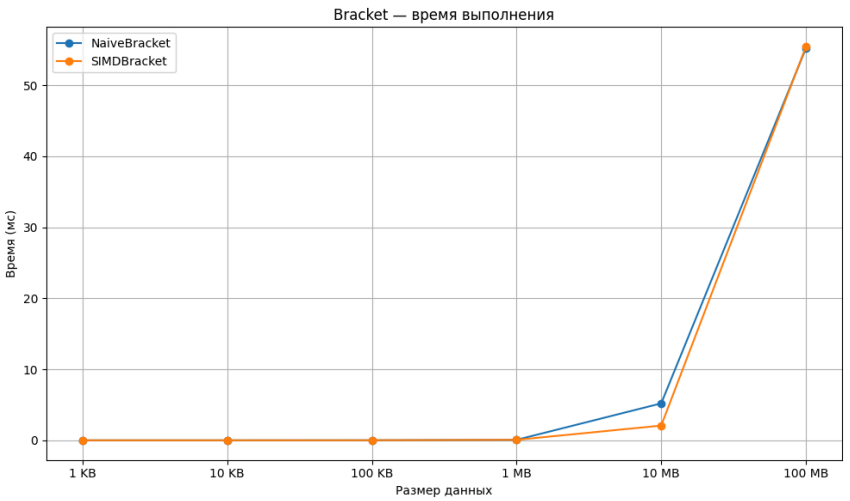


Рисунок 5 - Средние скорости выполнения операции проверки скобочных последовательностей

Также по полученным данным я составил графики средней пропускной способности, представленные на рисунках 6-7, для созданных решений. Из них можно увидеть, что пропускная способность ожидаемо растет с большим количеством обрабатываемых данных, но начиная со 100 Кб обрабатываемых данных, она резко падает и растет медленнее. Это связано с тем, что изначально обрабатываемые данные помещались в L1-L3 кэшах, но с увеличением объема данных, чтение данных переходит в область ограничений RAM памяти и файловой системы, из-за чего увеличиваются накладные расходы на чтение данных.

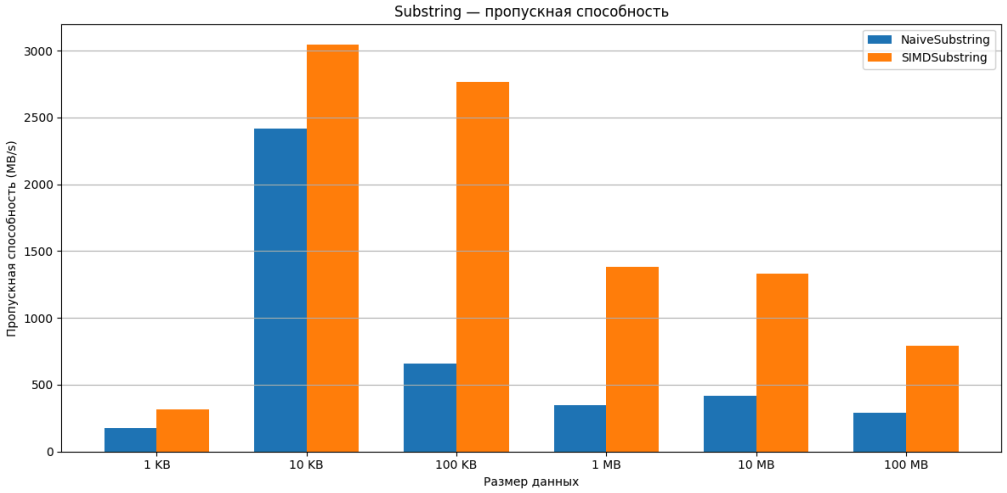


Рисунок 6 - График пропускной способности при разном количестве данных для операции поиска подстроки

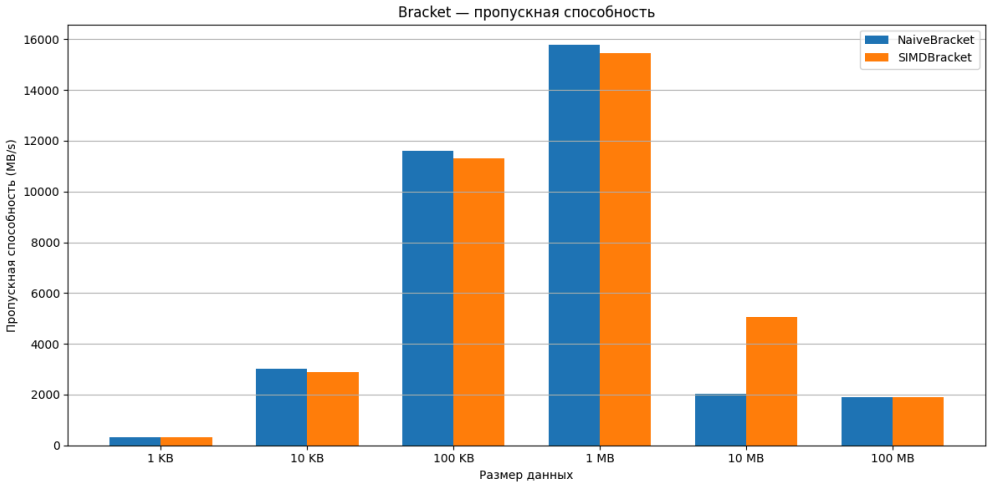


Рисунок 7 - График пропускной способности при разном количестве данных для операции проверки скобочных последовательностей

Вывод

Проведенное исследование эффективности SIMD-оптимизаций для задач обработки текста показало неоднозначные результаты, демонстрирующие как потенциал, так и ограничения векторных подходов.

Для задачи поиска подстроки применение AVX2-инструкций доказало свою эффективность, обеспечивая ускорение в 2-3 раза при обработке больших объемов данных. Наибольший выигрыш наблюдается на файлах размером от 1 МБ и выше, что подтверждает целесообразность использования SIMD-оптимизаций для работы с крупными текстовыми массивами.

Однако для задачи проверки скобочных последовательностей векторный подход не показал значительного преимущества, а в некоторых случаях даже уступал наивной реализации. Это объясняется фундаментальным ограничением алгоритма - необходимостью последовательной обработки стека, которая нивелирует выгоду от параллельного поиска скобок. Данный результат наглядно иллюстрирует, что не все алгоритмы поддаются эффективной векторизации.

Анализ пропускной способности выявил важную закономерность: при переходе порога в 100 КБ производительность начинает ограничиваться скоростью работы оперативной памяти и файловой системы, а не вычислительной мощностью процессора. Это указывает на то, что дальнейшая оптимизация должна учитывать не только вычислительную сложность, но и особенности работы с памятью.

Практическая значимость работы заключается в демонстрации избирательного подхода к применению SIMD-оптимизаций. Для задач, допускающих массовый параллелизм без сохранения состояния между операциями (как поиск подстрок), векторизация эффективна. Для алгоритмов с сильными последовательными зависимостями (как проверка скобок) традиционные подходы остаются предпочтительными.

Таким образом, решение о применении SIMD-оптимизаций должно приниматься с учетом специфики конкретной задачи и объема обрабатываемых данных, что подтверждает необходимость тщательного профилирования перед внедрением векторных подходов в production-системы.