Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление подготовки: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Организация и программирование вычислительных и информационных систем”

Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

К защите допустить:
Заведующий кафедрой
д. т. н., профессор _____ М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема: “Анализ и оптимизация применения векторных SIMD-инструкций для векторизации кода”

Студент _____ А. П. Буглин

Руководитель
к. т. н., доцент _____ А. А. Пазников

Консультант по нормативно-правовому регулированию
к. э. н., доцент _____ Ю. А. Елисеева

Консультант от кафедры
к. т. н., доцент, с. н. с. _____ И. С. Зуев

Санкт-Петербург
2023 г.

Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Организация и программирование вычислительных и информационных систем”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 2023 г.

ЗАДАНИЕ
на выпускную квалификационную работу

Студент А. П. Буглин Группа № 9305

1. Тема. Анализ и оптимизация векторных SIMD-инструкций для векторизации кода
(утверждена приказом № _____ от _____)

Место выполнения ВКР: Кафедра ВТ

2. Объект и предмет исследования
Объект исследования – векторные SIMD-инструкции. Предмет исследования – применение SIMD-инструкций для оптимизации алгоритмов.

3. Цель
Изучение и анализ векторных SIMD-инструкций с целью разработки оптимизированных алгоритмов.

4. Исходные данные
Научные статьи по теме работы.

5. Содержание
Описание векторных SIMD-инструкций, описание алгоритмов, которые могу быть векторизованы, реализация векторизованных операций с матрицами и числами с плавающей точкой.

6. Технические требования
Алгоритмы должны быть реализованы с помощью SIMD-инструкций и корректно работать.

7. Дополнительные разделы
Нормативно-правовое регулирование интеллектуальной деятельности.

8. Результаты
Пояснительная записка, программный код на языке программирования C++.

Дата выдачи задания
«_14_»_____марта_____2023 г.

Дата представления ВКР к защите
«_21_»_____июня_____2023 г.

Студент _____ А. П. Буглин

Руководитель _____ А. А. Пазников

Санкт-Петербургский государственный электротехнический университет

“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Организация и программирование вычислительных и информационных систем”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 2023 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Тема Анализ и оптимизация векторных SIMD-инструкций для векторизации кода

Студент А. П. Буглин Группа № 9305

№ этапа	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	15.03-21.03
2	Изучение SIMD библиотек xmmintrin и emmintrin	21.04-27.04
3	Реализация алгоритмов с матрицами и числами с плавающей точкой	28.04-04.05
4	Реализация оптимизированных версий алгоритмов с применением векторизации	05.05-15.05
5	Тестирование реализованных алгоритмов	16.05-24.05
6	Нормативно-правовое регулирование проекта	25.05-26.05
7	Оформление пояснительной записки	27.05-08.06
8	Представление работы к защите	21.06.2023

Руководитель
к. т. н., доцент _____ А. А. Пазников
Студент _____ А. П. Буглин

РЕФЕРАТ

Пояснительная записка содержит: 46стр., 12рис., 1 приложение.

Выпускная квалификационная работа посвящена анализу и оптимизации векторных SIMD-инструкций с целью векторизации кода.

Целью работы является изучение и анализ векторных SIMD-инструкций, а также разработка оптимизированных алгоритмов и структур данных, которые могут эффективно использовать эти инструкции. Предметом исследования являются различные алгоритмы, которые могут быть векторизованы с использованием SIMD-инструкций.

В рамках работы проводится анализ существующих векторных SIMD-инструкций и их применение для различных типов задач и алгоритмов. Изучаются особенности векторизации кода и оптимизации с использованием SIMD-инструкций.

Далее, на основе полученных знаний, разрабатываются оптимизированные версии алгоритмов, которые максимально используют возможности векторных SIMD-инструкций. Производится реализация этих оптимизированных алгоритмов на языке программирования C++.

Результаты работы показывают эффективность и преимущества применения векторных SIMD-инструкций для векторизации кода. Оптимизированные алгоритмы и структуры данных демонстрируют улучшенную производительность и скорость выполнения по сравнению с исходными версиями.

ABSTRACT

The work is devoted to the analysis and optimization of vector SIMD instructions for the purpose of code vectorization.

The aim of the work is to study and analyze vector SIMD instructions, as well as to develop optimized algorithms and data structures that can effectively use these instructions. The subject of the study is various algorithms that can be vectorized using SIMD instructions.

As part of the work, the analysis of existing vector SIMD instructions and their application for various types of tasks and algorithms is carried out. The features of code vectorization and optimization using SIMD instructions are studied.

Further, on the basis of the acquired knowledge, optimized versions of algorithms and data structures are developed that maximize the capabilities of vector SIMD instructions. These optimized algorithms are implemented in the C++ programming language.

The results of the work show the efficiency and advantages of using vector SIMD instructions for code vectorization. Optimized algorithms and data structures demonstrate improved performance and execution speed compared to the original versions.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

AVX – Advanced Vector Extensions. Расширение SIMD-инструкций, разработанное компанией Intel.

FMA – Fused Multiply-Add. Операция, которая выполняет одновременное умножение и сложение двух чисел.

SIMD – Single Instruction, Multiple Data. Технология, позволяющая выполнять одну инструкцию на нескольких элементах данных одновременно.

SISD – Single Instruction, Single Data. Классификация компьютерных систем, в которых каждая инструкция работает только с одним элементом данных за раз.

SSE – Streaming SIMD Extensions. Расширение SIMD-инструкций, разработанное компанией Intel.

ЭВМ – Электронная вычислительная машина. Компьютерная система или устройство, предназначенное для выполнения вычислений и обработки информации.

ВВЕДЕНИЕ

В современном мире, все больше и больше приложений и программ требуют максимальной производительности для быстрого и эффективного решения задач. Это касается не только научных и технических вычислений, но и многих других сфер деятельности, таких как мультимедиа, финансы, машинное обучение и т. д.

Для того, чтобы оптимизировать производительность приложений и программ, многие разработчики и исследователи используют различные методы оптимизации, включая оптимизацию алгоритмов, использование параллелизма и векторизации кода. Использование векторных SIMD-инструкций является одним из ключевых методов векторизации кода, который позволяет одновременно обрабатывать несколько элементов данных. Это существенно ускоряет выполнение операций и, как следствие, увеличивает производительность системы в целом.

В свете быстрого роста количества ядер и потоков в современных процессорах, оптимизация использования SIMD-инструкций становится еще более актуальной. Существует множество различных архитектур процессоров, которые могут использовать SIMD-инструкции по-разному, что требует от разработчиков знания о том, как эффективно использовать эти инструкции на разных платформах. Таким образом, оптимизация использования векторных SIMD-инструкций является важной задачей для разработчиков программного обеспечения и исследователей, занимающихся оптимизацией вычислительных алгоритмов.

Целью данной работы является изучение и анализ векторных SIMD-инструкций с целью разработки оптимизированных алгоритмов, которые могут эффективно использовать эти инструкции.

Объектом исследования данной работы являются векторные SIMD-инструкции и их применение для оптимизации и векторизации кода. Предме-

том исследования является применение SIMD-инструкций для оптимизации алгоритмов

Для достижения обозначенной цели необходимо выполнить следующие задачи:

- Изучить принципы работы векторных SIMD-инструкций.
- Проанализировать существующих алгоритмов с точки зрения их пригодности для векторизации с использованием SIMD-инструкций.
- Разработать оптимизированные версии алгоритмов, максимально использующие возможности векторных SIMD-инструкций.
- Провести измерения производительности и сравнение с исходными версиями без векторизации.
- Оценить полученные результаты и сделать выводы о применимости и эффективности использования векторных SIMD-инструкций.

В первом разделе работы рассматриваются основы работы с векторными SIMD-инструкциями, включая их принципы функционирования, особенности и возможности. Во втором разделе разрабатываются и описываются оптимизированные версии алгоритмов, которые максимально используют возможности векторных SIMD-инструкций. Третий раздел представляет собой измерения производительности разработанных оптимизированных алгоритмов. Осуществляется сравнение с исходными версиями без векторизации и анализ полученных результатов. Четвертый раздел содержит нормативно-правовое регулирование интеллектуальной деятельности.

1 Теоретические основы векторизации кода

В данном разделе будут рассмотрены теоретические основы векторизации кода и использования векторных SIMD-инструкций. Будут рассмотрены основные принципы работы SIMD-инструкций, их классификация, а также методы векторизации кода. Понимание этих основных принципов является необходимым для эффективного использования SIMD-инструкций при оптимизации кода.

1.1 Определение векторизации кода и ее применение

Векторизация кода - это метод оптимизации программного кода, который заключается в преобразовании последовательности операций над отдельными элементами данных в одну операцию над вектором данных. Такой подход позволяет достичь значительного увеличения производительности программы за счет параллельной обработки нескольких элементов данных одновременно.

Большинство персональных ЭВМ до последнего времени являлись обычными, последовательными компьютерами (SISD [1]), в которых в каждый момент времени выполняется лишь одна операция над одним элементом данных (числовым или каким-либо другим значением).

SIMD [2] – принцип компьютерных вычислений, позволяющий обеспечить параллелизм на уровне данных. SIMD-процессоры называются также векторными.

Векторизация широко применяется в различных областях, где требуется обработка больших объемов данных, в том числе в научных вычислениях, обработке изображений и звука, графических приложениях и т.д. Однако, для эффективной векторизации кода необходимо уметь определять, какие алгоритмы и задачи могут быть векторизованы, и использовать соответствующие методы оптимизации, включая SIMD-инструкции.

Аппаратная поддержка векторизации находится на уровне процессора и представляет собой специализированные блоки, которые позволяют выполнить SIMD-инструкции. В современных процессорах поддержка векторизации является обычным элементом архитектуры и способствует значительному ускорению выполнения задач, которые могут быть векторизованы. Однако, не все процессоры имеют одинаковую поддержку векторизации, и векторные SIMD-инструкции могут иметь различные особенности и ограничения в использовании на разных процессорах.

Например, SIMD-инструкции на процессорах Intel могут использоваться в различных контекстах, таких как SSE, AVX, AVX-512, FMA [3], и другие. Каждый из этих контекстов имеет свои особенности, поддерживаемые типы данных и размеры векторов, которые необходимо учитывать при оптимизации кода.

Для достижения оптимальной производительности с использованием векторизации и SIMD-инструкций, необходимо проводить анализ процессора и его возможностей, а также оптимизировать код под конкретную архитектуру процессора. Это включает в себя выбор подходящих SIMD-инструкций и оптимального размера вектора, а также оптимизацию памяти и использование специализированных инструкций для обработки данных.

Применение векторизации кода широко распространено в различных областях вычислительной техники, таких как научные вычисления, графические приложения, обработка сигналов и другие области.

В научных вычислениях векторизация часто используется для ускорения вычислений в задачах, требующих многократных вычислений одного и того же алгоритма на разных наборах данных. Например, векторизация может использоваться для ускорения вычислений в задачах математического моделирования, молекулярной динамики, анализа изображений и других областях науки.

В графических приложениях векторизация применяется для ускорения обработки графических данных, таких как изображения и видео. Векторные

SIMD-инструкции могут использоваться для обработки пиксельных данных, изменения размеров изображений, а также для реализации различных эффектов и фильтров.

Обработка сигналов также является областью, в которой векторизация широко применяется. Например, векторизация может использоваться для ускорения обработки аудио и видеоданных, а также для реализации алгоритмов обработки сигналов в области радиосвязи, медицины и других областях.

В общем, применение векторизации кода может быть весьма эффективно в различных областях, где требуется высокая производительность вычислений. Однако, для достижения максимальной эффективности использования векторизации, необходимо проводить анализ задач и алгоритмов на предмет возможности их векторизации, а также оптимизировать код под конкретную архитектуру процессора.

1.2 Основные принципы работы SIMD-инструкций

В настоящее время наиболее распространенными SIMD-инструкциями являются MMX, SSE, AVX и NEON [4].

MMX является первым набором SIMD-инструкций, который был разработан компанией Intel в 1997 году. Он был предназначен для ускорения обработки мультимедийных данных, таких как изображения и звуковые файлы. MMX-инструкции позволяют выполнить операции над восемью 8-битными целыми числами, четырьмя 16-битными целыми числами, двумя 32-битными целыми числами или двумя 32-битными числами с плавающей точкой одновременно.

SSE является расширением MMX, которое было разработано Intel в 1999 году. SSE-инструкции позволяют выполнить операции над четырьмя 32-битными числами с плавающей точкой или четырьмя 32-битными целыми числами одновременно. Кроме того, SSE-инструкции поддерживают более широкий набор операций, чем MMX.

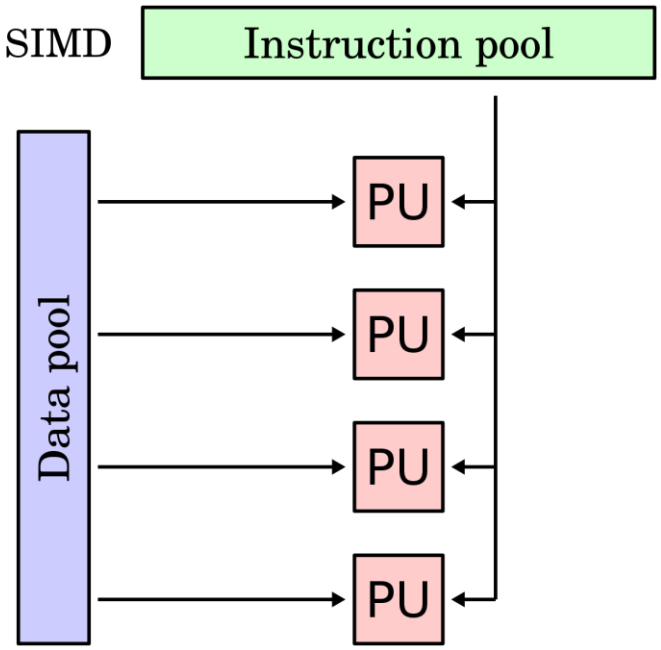

Рисунок 1.1 – Операции SSE с четырьмя числами

AVX является дополнительным расширением SSE, которое было представлено компанией Intel в 2011 году. AVX-инструкции позволяют выполнять операции над восемью 32-битными числами с плавающей точкой или восемью 32-битными целыми числами одновременно. Кроме того, AVX-инструкции поддерживают новые операции, такие как операции сравнения и перемещения данных.

NEON является набором SIMD-инструкций, разработанных компанией ARM [5] для ее процессоров с архитектурой ARM. NEON-инструкции поддерживают выполнение операций над восемью 8-битными целыми числами, четырьмя 16-битными целыми числами, двумя 32-битными целыми числами или четырьмя 32-битными числами с плавающей точкой одновременно.

С целью обеспечения удобства разработки, Intel предлагает использовать свою библиотеку интринсиков, которая позволяет взаимодействовать с SIMD регистрами и инструкциями, используя код на языке C/C++.

Инструкции процессора написаны последовательно, исходя из предположения, что они будут выполняться последовательно. Однако фактически несколько инструкций могут выполняться одновременно. У ядра процессора имеется несколько портов, каждый из которых обладает своим набором ин-

струкций, которые он может выполнять. Если две инструкции могут быть размещены на разных портах и результат второй инструкции не зависит от результата первой, то такие инструкции могут быть выполнены параллельно. Поэтому важно разрабатывать код таким образом, чтобы инструкции имели возможность выполняться параллельно.

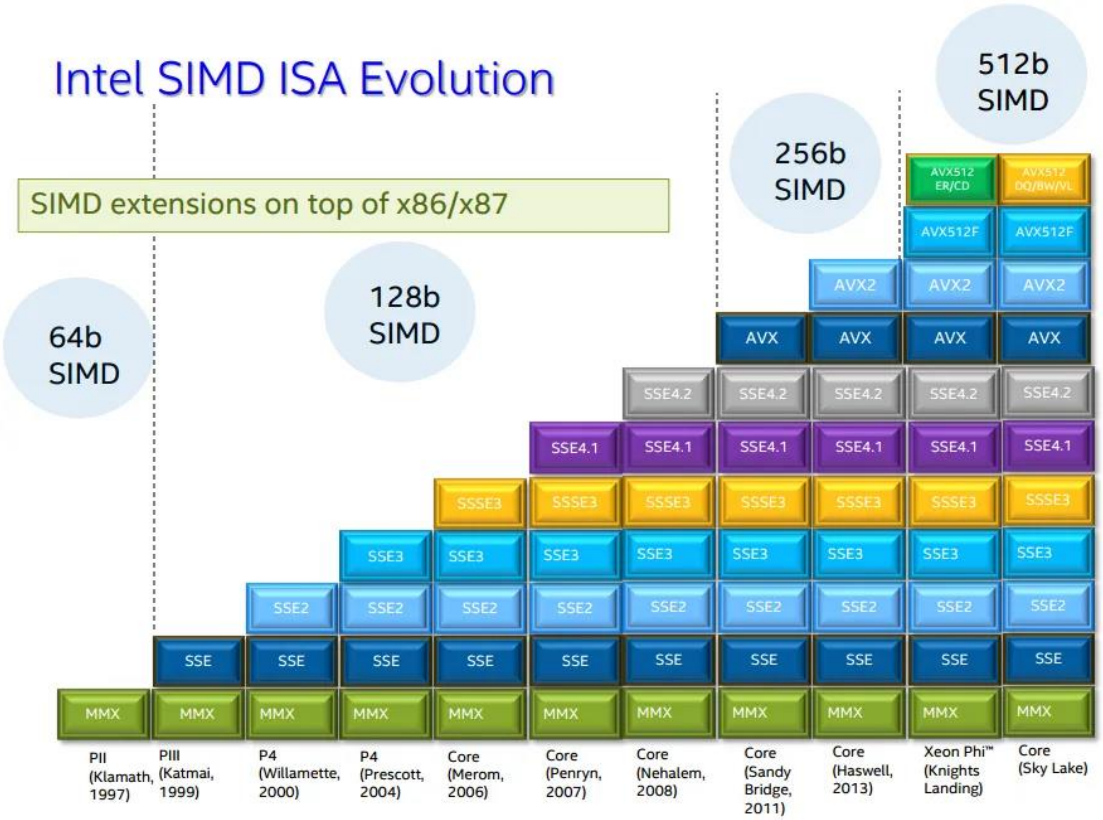

Рисунок 1.2 – Расширение Intel SIMD

Чтобы операции могли распараллеливаться, нужно, чтобы подряд идущие инструкции использовали разные регистры, при этом важно, чтобы код не использовал больше регистров, чем доступно, так как это может привести к тому, что компилятор будет выгружать промежуточные результаты в память, чтобы освободить регистры.

К тому же важно, чтобы в коде было как можно меньше ветвлений, чтобы процессор знал, какая инструкция идёт следующей. Для этого используется разворачивание циклов и отказ от условий в “If”, которые нельзя вычислить во время компиляции.

1.3 Алгоритмы, которые могут быть векторизованы

Векторизация кода позволяет ускорить выполнение операций, которые могут быть параллельно выполнены на нескольких элементах данных. Среди алгоритмов, которые могут быть векторизованы, можно выделить следующие [6]:

- Матричные операции часто используются в научных вычислениях и графике. Умножение матриц, нахождение обратной матрицы и другие операции могут быть векторизованы с использованием SIMD-инструкций.
- Арифметические операции также могут быть векторизованы. Сложение, вычитание, умножение и деление чисел могут быть выполнены параллельно на нескольких элементах данных с помощью SIMD-инструкций.
- Операции с плавающей запятой, такие как вычисление квадратного корня, функций тригонометрии и другие, также могут быть векторизованы. Векторизация таких операций может существенно ускорить выполнение вычислений в научных приложениях и играх.
- Обработка сигналов, такая как фильтрация, корреляция и другие, также могут быть векторизованы. Векторизация таких операций может существенно ускорить выполнение обработки аудио и видеоданных.

Особенности векторизации различных алгоритмов могут быть связаны с выравниванием памяти, использованием условных операций, зависимостями данных и т.д.

Выравнивание памяти может играть важную роль при векторизации некоторых алгоритмов, особенно тех, которые работают с большими объемами данных. Например, для корректной работы SIMD-инструкций векторы данных должны быть выровнены по определенному адресу в памяти. Если это условие не выполняется, то могут возникнуть проблемы с производи-

тельностью или корректностью работы программы. Поэтому, при векторизации алгоритмов, необходимо учитывать этот фактор и, при необходимости, использовать специальные инструкции для выравнивания данных.

Использование условных операций также может представлять сложность при векторизации алгоритмов. SIMD-инструкции предназначены для выполнения одних и тех же операций на нескольких элементах данных одновременно, что подразумевает отсутствие условных операций. Если в алгоритме имеются условные операции, то может потребоваться их модификация для векторизации или использование других подходов для обхода этой проблемы.

Зависимости данных также могут оказать влияние на векторизацию алгоритмов. Если алгоритм содержит циклы с зависимостью по данным, то векторизация может быть затруднена или невозможна. В таких случаях может потребоваться изменение алгоритма или использование специальных инструкций для работы с зависимостями данных.

2 Оптимизация алгоритмов и структур данных с использованием векторных SIMD-инструкций

Раздел описывает разработку и оптимизацию алгоритмов. Каждый из алгоритмов представлен в виде соответствующей функции и использует векторные SIMD-инструкции для достижения оптимальной производительности при выполнении вычислений.

2.1 Вычисление среднего значения вектора

Оптимизированный алгоритм вычисления среднего значения позволяет выполнять суммирование элементов вектора, обрабатывая 4 элемента одновременно.

Опишем алгоритм вычисления среднего значения и приведем его код (рисунок 2.1):

1. Создается переменная sumx4 типа __m128 (регистр SIMD), инициализируемая нулевыми значениями. __m128 является 128-битным регистром SIMD, способным содержать четыре 32-битных числа с плавающей запятой (56 строка).
2. Цикл выполняется от 0 до length - 1, с шагом 4 (j = j + 4), чтобы обрабатывать 4 элемента вектора одновременно (57 строка).
3. Внутри цикла происходит загрузка 4 последовательных элементов вектора a в регистр SIMD с помощью _mm_loadu_ps() функции (59 строка).
4. Далее происходит сложение содержимого регистра sumx4 и загруженных значений вектора при помощи _mm_add_ps() функции (59 строка).
5. Создается объект Base::Lanes<__m128, float>, который позволяет получить доступ к отдельным значениям вектора в регистре SIMD (61 строка).

6. Используя объект lanes, суммы значений регистра SIMD разделяются на 4 компоненты. Производится сложение компонент x(), y(), z(), w(), которые содержат суммы 4-х элементов вектора (62 строка).
7. Затем полученная сумма делится на значение length, чтобы получить среднее значение (62 строка).
8. Результатом функции является вычисленное среднее значение.

```
53    static float SimdAverageKernel()
54    {
55        preventOptimize++;
56        __m128 sumx4 = _mm_set_ps1(0.0);
57        for (uint32_t j = 0, l = length; j < l; j = j + 4)
58        {
59            sumx4 = _mm_add_ps(sumx4, _mm_loadu_ps(&(a[j])));
60        }
61        Base::Lanes<__m128, float> lanes(sumx4);
62        return (lanes.x() + lanes.y() + lanes.z() + lanes.w()) / length;
63    }
```

Рисунок 2.1 – Код функции среднего значения

2.2. Нахождение обратной матрицы

Оптимизированный алгоритм нахождения обратной матрицы выполняется путем загрузки и транспонирования исходной матрицы, вычисления миноров и определителя матрицы с использованием SIMD-инструкций, вычисления обратных значений и сохранения результатов.

Опишем алгоритм нахождения обратной матрицы:

1. Создаются регистры SIMD типа __m128, такие как src0, src1, src2, src3, row0, row1, row2, row3, tmp1, minor0, minor1, minor2, minor3, det.
2. Регистр __m128 представляет собой 128-битный SIMD-регистр, способный содержать четыре 32-битных числа с плавающей запятой.
3. Исходная матрица загружается из памяти в регистры SIMD с помощью _mm_loadu_ps() функции.
4. Происходит транспонирование матрицы с использованием SIMD-инструкций _mm_shuffle_ps() для перестановки элементов матрицы и формирования новых регистров row0, row1, row2, row3.

5. Используя SIMD-инструкции, вычисляются миноры матрицы путем перемножения и перестановки элементов регистров row0, row1, row2, row3 с использованием операций _mm_mul_ps(), _mm_shuffle_ps(), _mm_sub_ps() и _mm_add_ps().
6. Результаты вычислений сохраняются в регистрах minor0, minor1, minor2, minor3.
7. Определитель матрицы вычисляется путем перемножения элементов регистра row0 с соответствующими элементами регистра minor0 с использованием SIMD-инструкций _mm_mul_ps() и _mm_shuffle_ps().
8. Затем происходит суммирование элементов с использованием SIMD-инструкций _mm_add_ps() и _mm_shuffle_ps().
9. Для получения обратного определителя выполняется операция реципрокного значения _mm_rcp_ps().
10. Обратные значения матрицы вычисляются путем умножения регистров minor0, minor1, minor2, minor3 на обратный определитель с использованием SIMD-инструкции _mm_mul_ps().
11. Вычисленные обратные значения сохраняются в память с использованием _mm_storeu_ps() функции.
12. Происходит инкремент переменной preventOptimize, чтобы избежать оптимизации компилятором.
13. После завершения цикла возвращается значение preventOptimize.

2.3. Умножение матрицы на вектор

Умножение матрицы на вектор - это операция, при которой каждый элемент результирующего вектора вычисляется путем скалярного произведения соответствующей строки матрицы и вектора. Для оптимизации данной операции используются SIMD-инструкции.

Оптимизированный алгоритм выполняется путем загрузки значений строк матрицы и вектора в регистры SIMD, выполнения умножения и скалярного сложения с использованием SIMD-инструкций и сохранения результатов.

Опишем алгоритм умножения матрицы на вектор и приведем его код (рисунок 2.2):

1. Создаются регистры SIMD типа __m128, такие как a0, a1, a2, a3, b0, b1, b2, b3, out0, out1, out2, out3.
2. Регистр __m128 представляет собой 128-битный SIMD-регистр, способный содержать четыре 32-битных числа с плавающей запятой.
3. Итерируемся по всем элементам вектора (размерность n) с помощью переменной i (249 строка).
4. Загружаем из памяти значения элементов 4-х последовательных строк матрицы (t1x4) в регистры a0, a1, a2, a3.
5. Загружаем из памяти значения 4-х последовательных элементов вектора (t2x4) в регистры b0, b1, b2, b3 (251 - 254 строки).
6. Используя SIMD-инструкции, производим умножение и скалярное сложение элементов регистров a0, a1, a2, a3 и b0, b1, b2, b3 с использованием операций _mm_mul_ps() и _mm_add_ps() (256, 267, 278, 289 строки).
7. Результаты вычислений сохраняются в регистрах out0, out1, out2, out3.
8. Сохраняем значения элементов регистров out0, out1, out2, out3 в память (outx4) с использованием _mm_storeu_ps() функции (258, 269, 280, 291 строки).
9. После выполнения итерации увеличиваем переменную preventOptimize, чтобы избежать оптимизации компилятором (299 строка).
10. По завершении цикла возвращается значение preventOptimize (301 строка).

```
static uint64_t SimdMultiply(uint64_t n)
{
    for (uint64_t i = 0; i < n; i++)
    {
        __m128 a0 = _mm_loadu_ps(&t1x4[0]);
        __m128 a1 = _mm_loadu_ps(&t1x4[4]);
        __m128 a2 = _mm_loadu_ps(&t1x4[8]);
        __m128 a3 = _mm_loadu_ps(&t1x4[12]);

        __m128 b0 = _mm_loadu_ps(&t2x4[0]);
        _mm_storeu_ps(
            &outx4[0],
            _mm_add_ps(
                _mm_mul_ps(_mm_shuffle_ps(b0, b0, _MM_SHUFFLE(0, 0, 0, 0)), a0),
                _mm_add_ps(
                    _mm_mul_ps(_mm_shuffle_ps(b0, b0, _MM_SHUFFLE(1, 1, 1, 1)), a1),
                    _mm_add_ps(
                        _mm_mul_ps(_mm_shuffle_ps(b0, b0, _MM_SHUFFLE(2, 2, 2, 2)), a2),
                        _mm_mul_ps(_mm_shuffle_ps(b0, b0, _MM_SHUFFLE(3, 3, 3, 3)), a3)))));

        __m128 b1 = _mm_loadu_ps(&t2x4[4]);
        _mm_storeu_ps(
            &outx4[4],
            _mm_add_ps(
                _mm_mul_ps(_mm_shuffle_ps(b1, b1, _MM_SHUFFLE(0, 0, 0, 0)), a0),
                _mm_add_ps(
                    _mm_mul_ps(_mm_shuffle_ps(b1, b1, _MM_SHUFFLE(1, 1, 1, 1)), a1),
                    _mm_add_ps(
                        _mm_mul_ps(_mm_shuffle_ps(b1, b1, _MM_SHUFFLE(2, 2, 2, 2)), a2),
                        _mm_mul_ps(_mm_shuffle_ps(b1, b1, _MM_SHUFFLE(3, 3, 3, 3)), a3)))));

        __m128 b2 = _mm_loadu_ps(&t2x4[8]);
        _mm_storeu_ps(
            &outx4[8],
            _mm_add_ps(
                _mm_mul_ps(_mm_shuffle_ps(b2, b2, _MM_SHUFFLE(0, 0, 0, 0)), a0),
                _mm_add_ps(
                    _mm_mul_ps(_mm_shuffle_ps(b2, b2, _MM_SHUFFLE(1, 1, 1, 1)), a1),
                    _mm_add_ps(
                        _mm_mul_ps(_mm_shuffle_ps(b2, b2, _MM_SHUFFLE(2, 2, 2, 2)), a2),
                        _mm_mul_ps(_mm_shuffle_ps(b2, b2, _MM_SHUFFLE(3, 3, 3, 3)), a3)))));

        __m128 b3 = _mm_loadu_ps(&t2x4[12]);
        _mm_storeu_ps(
            &outx4[12],
            _mm_add_ps(
                _mm_mul_ps(_mm_shuffle_ps(b3, b3, _MM_SHUFFLE(0, 0, 0, 0)), a0),
                _mm_add_ps(
                    _mm_mul_ps(_mm_shuffle_ps(b3, b3, _MM_SHUFFLE(1, 1, 1, 1)), a1),
                    _mm_add_ps(
                        _mm_mul_ps(_mm_shuffle_ps(b3, b3, _MM_SHUFFLE(2, 2, 2, 2)), a2),
                        _mm_mul_ps(_mm_shuffle_ps(b3, b3, _MM_SHUFFLE(3, 3, 3, 3)), a3)))));
        preventOptimize++;
    }
    return preventOptimize;
}
};
```

Рисунок 2.2 – Код функции умножения матрицы

2.4. Транспонирование матрицы

Транспонирование матрицы - это операция, при которой строки матрицы становятся столбцами и наоборот. В оптимизированном алгоритме ис-

пользуются SIMD-инструкции для эффективного выполнения операции транспонирования.

Оптимизированный алгоритм выполняется путем загрузки значений элементов матрицы в регистры, выполнения перестановок элементов с использованием инструкций, сохранения результатов.

Опишем алгоритм транспонирования матрицы и приведем его код (рисунок 2.3):

1. Создаются регистры SIMD типа __m128, такие как src0, src1, src2, src3, tmp01, tmp23, dst0, dst1, dst2, dst3.
2. Регистр __m128 представляет собой 128-битный SIMD-регистр, способный содержать четыре 32-битных числа с плавающей запятой.
3. Итерируемся по всем элементам матрицы (размерность n) с помощью переменной i.
4. Загружаем из памяти значения 16 последовательных элементов матрицы (srcx4) в регистры src0, src1, src2, src3.
5. Используя SIMD-инструкции _mm_shuffle_ps(), производим перестановку элементов в регистрах src0, src1, src2, src3 для формирования временных регистров tmp01, tmp23.
6. Для формирования конечных регистров dst0, dst1, dst2, dst3 также используется перестановка элементов в регистрах tmp01, tmp23.
7. Сохраняем значения элементов регистров dst0, dst1, dst2, dst3 в память (dstx4) с использованием _mm_storeu_ps() функции.
8. После выполнения итерации увеличиваем переменную preventOptimize, чтобы избежать оптимизации компилятором.
9. По завершении цикла возвращается значение preventOptimize.

```
175     static uint64_t SimdTranspose(uint64_t n)
176     {
177         for (uint64_t i = 0; i < n; ++i)
178         {
179             __m128 src0 = _mm_loadu_ps(&srcx4[0]);
180             __m128 src1 = _mm_loadu_ps(&srcx4[4]);
181             __m128 src2 = _mm_loadu_ps(&srcx4[8]);
182             __m128 src3 = _mm_loadu_ps(&srcx4[12]);
183 
184             __m128 tmp01 = _mm_shuffle_ps(src0, src1, _MM_SHUFFLE(1, 0, 1, 0));
185             __m128 tmp23 = _mm_shuffle_ps(src2, src3, _MM_SHUFFLE(1, 0, 1, 0));
186             __m128 dst0 = _mm_shuffle_ps(tmp01, tmp23, _MM_SHUFFLE(2, 0, 2, 0));
187             __m128 dst1 = _mm_shuffle_ps(tmp01, tmp23, _MM_SHUFFLE(3, 1, 3, 1));
188 
189             tmp01 = _mm_shuffle_ps(src0, src1, _MM_SHUFFLE(3, 2, 3, 2));
190             tmp23 = _mm_shuffle_ps(src2, src3, _MM_SHUFFLE(3, 2, 3, 2));
191             __m128 dst2 = _mm_shuffle_ps(tmp01, tmp23, _MM_SHUFFLE(2, 0, 2, 0));
192             __m128 dst3 = _mm_shuffle_ps(tmp01, tmp23, _MM_SHUFFLE(3, 1, 3, 1));
193 
194             _mm_storeu_ps(&dstx4[0], dst0);
195             _mm_storeu_ps(&dstx4[4], dst1);
196             _mm_storeu_ps(&dstx4[8], dst2);
197             _mm_storeu_ps(&dstx4[12], dst3);
198 
199             preventOptimize++;
200         }
201         return preventOptimize;
202     }
```

Рисунок 2.3 – Код функции транспонирования матрицы

2.5. Преобразование вершины

Преобразование вершины - это операция, при которой координаты вершины умножаются на соответствующие элементы матрицы преобразования.

Опишем алгоритм преобразования вершины и приведем его код (рисунок 2.4):

1. Создаются регистры SIMD типа __m128, такие как z, v, xxxx, yyyy, zzzz, wwww.
2. Регистр __m128 представляет собой 128-битный SIMD-регистр, способный содержать четыре 32-битных числа с плавающей запятой.
3. Итерируемся по всем вершинам (размерность n) с помощью переменной i.
4. Загружаем значения координат вершины из памяти в регистр v с использованием _mm_loadu_ps().

5. Загружаем соответствующие элементы матрицы преобразования из памяти в регистры tx4 с использованием _mm_loadu_ps().
6. С помощью SIMD-инструкции _mm_shuffle_ps() производим перестановку элементов в регистре v для формирования регистров xxxx, yyyy, zzzz, wwww.
7. Умножаем значения элементов регистров xxxx, yyyy, zzzz, wwww на соответствующие элементы матрицы, загруженные в регистры tx4, с использованием SIMD-инструкции _mm_mul_ps().
8. Для каждого элемента координаты вершины выполняется умножение на соответствующий элемент матрицы и суммирование результатов в регистре z с использованием SIMD-инструкции _mm_add_ps().
9. Сохраняем значения элементов регистра z в память с помощью _mm_storeu_ps().
10. Увеличиваем значение preventOptimize для избежания оптимизации компилятором.
11. Возвращаем значение preventOptimize.

```
167    static uint64_t SimdVertextTransform(uint64_t n)
168    {
169        for (uint64_t i = 0; i < n; i++)
170        {
171            __m128 z = _mm_set1_ps(0.0);
172            __m128 v = _mm_loadu_ps(&vx4[0]);
173            __m128 xxxx = _mm_shuffle_ps(v, v, _MM_SHUFFLE(0, 0, 0, 0));
174            z = _mm_add_ps(z, _mm_mul_ps(xxxx, _mm_loadu_ps(&tx4[0])));
175            __m128 yyyy = _mm_shuffle_ps(v, v, _MM_SHUFFLE(1, 1, 1, 1));
176            z = _mm_add_ps(z, _mm_mul_ps(yyyy, _mm_loadu_ps(&tx4[4])));
177            __m128 zzzz = _mm_shuffle_ps(v, v, _MM_SHUFFLE(2, 2, 2, 2));
178            z = _mm_add_ps(z, _mm_mul_ps(zzzz, _mm_loadu_ps(&tx4[8])));
179            __m128 wwww = _mm_shuffle_ps(v, v, _MM_SHUFFLE(3, 3, 3, 3));
180            z = _mm_add_ps(z, _mm_mul_ps(wwww, _mm_loadu_ps(&tx4[12])));
181            _mm_storeu_ps(&outx4[0], z);
182            preventOptimize++;
183        }
184        return preventOptimize;
185    }
```

Рисунок 2.4 – Код функции преобразования вершины

3 Сравнительный анализ производительности алгоритмов

Сравнительный анализ производительности позволяет определить, насколько успешно оптимизированные версии алгоритмов улучшают производительность по сравнению с исходными версиями. Мы сравниваем время выполнения, чтобы получить объективную оценку эффективности оптимизаций.

3.1 Описание используемых инструментов и средств измерения

Перед проведением сравнительного анализа производительности мы должны обеспечить точные и надежные измерения. Для этого мы используем специализированные инструменты и средства, которые позволяют нам собирать данные о производительности алгоритмов.

В основе нашего исследования лежит использование профилировщика, который позволяют измерять время выполнения алгоритмов.

Основные функции профилировщика включают в себя:

- Now(): Эта функция используется для получения текущего времени в миллисекундах. Она использует структуру timeb, которая предоставляет информацию о текущем времени, включая секунды и миллисекунды.
- TimeKernel(): Данная функция измеряет время выполнения конкретного ядра (kernel) алгоритма. Она принимает на вход указатель на функцию ядра (kernel) и количество итераций (iterations). Функция TimeKernel() измеряет время начала выполнения (start), запускает ядро с заданным количеством итераций и затем измеряет время окончания выполнения (stop). Разность между stop и start предоставляет время выполнения ядра.
- ComputeIterations(): Эта функция определяет количество итераций, необходимых для достижения заданного времени выполнения (desiredRuntime) самого медленного ядра. Она использует функцию TimeKernel() для измерения времени выполнения ядер с различными

количествами итераций и находит максимальное время (maxTime). Затем она вычисляет количество итераций, необходимых для достижения 1-секундного времени выполнения самого медленного ядра.

- RunOne(): Эта функция запускает профилировку для одного конкретного алгоритма (benchmark). Она инициализирует ядра, определяет количество итераций и выполняет измерения времени выполнения для SIMD-ядра, а также для не-SIMD ядер с ширинами 32 и 64 бита. В конце профилировки она также проверяет корректность выполнения и очищает ресурсы.
- Report(): Данная функция генерирует отчет о производительности алгоритма (benchmark). Она вычисляет отношения времени выполнения не-SIMD ядер к времени выполнения SIMD-ядра и выводит результаты, включая количество итераций, время выполнения каждого ядра и соответствующие отношения, используя функцию PrintColumns().
- RunAll(): Эта функция запускает профилирование для всех алгоритмов в списке benchmarkList. Она вызывает PrintHeaders() для вывода заголовков таблицы результатов и затем последовательно выполняет профилировку для каждого алгоритма, используя функции RunOne() и Report().
- Add(): Эта функция добавляет алгоритм (benchmark) в список benchmarkList для последующего профилирования.

Устройство профилировщика позволяет сравнивать производительность различных алгоритмов и измерять влияние оптимизаций векторизации SIMD на время выполнения. Это важный инструмент для анализа и оптимизации алгоритмов, а также для принятия решений об оптимальном выборе алгоритма в конкретных ситуациях.

3.2 Описание используемых инструментов и средств измерения

Для проведения сравнительного анализа производительности алгоритмов и оценки эффективности оптимизированных версий сравнительно с исходными версиями без векторизации, мы определили следующие параметры и условия экспериментов:

- Выбор алгоритмов: Выбранный набор алгоритмов представляет интерес для исследования и оптимизации. Включение алгоритмов основано на их значимости, распространенности и потенциале для оптимизации с использованием векторизации SIMD.
- Программная реализация: Каждый алгоритм был реализован в виде программного кода с использованием языка программирования C++. Использовались подходящие оптимизации и техники для реализации алгоритмов векторизации.
- Используемые инструменты и средства измерения: Как описано в предыдущем подразделе, использовался разработанный нами устройство профилировщика. Оно предоставляет возможность измерения времени выполнения ядер алгоритмов, сравнения результатов и определения производительности.
- Характеристики системы: Эксперименты проводились на одном компьютере с определенными характеристиками: процессор Intel Core i5-6200U, объем оперативной памяти 6,00 ГБ. Указание этих характеристик позволяет оценить влияние аппаратной конфигурации на результаты экспериментов.
- Входные данные: Для каждого алгоритма был определен набор входных данных, который использовался для проведения экспериментов. Входные данные были выбраны таким образом, чтобы они представляли типичные сценарии использования алгоритмов и позволяли оценить их производительность.

- Методика измерения: Проводились несколько запусков каждого алгоритма с разными параметрами и записывали время выполнения для каждого измерения. Для повышения достоверности результатов, усреднение времени выполнения и другие статистические методы могут быть использованы для анализа данных.

3.3 Сравнение времени выполнения оптимизированных и неоптимизированных алгоритмов

Используемый профилировщик позволил получить результаты, которые приведены на рисунке 3.1

Name	:	Iterations	Scalar32(ns)	Scalar64(ns)	SIMD32(ns)
AverageFloat32x4	:	27978	35456	35921	13832
VertexTransform	:	1424695	701	262	61
MatrixMultiplication	:	18267874	53	93	52
MatrixTranspose	:	49344752	20	14	14
MatrixInverse	:	4615211	218	216	136

Рисунок 3.1 – Результат работы программы

Алгоритм "Вычисление среднего значения вектора" (рисунок 3.2):

- Количество итераций: 27978.
- Время выполнения для неоптимизированной версии на скалярных операциях (Scalar32): 35456 наносекунд.
- Время выполнения для оптимизированной версии с использованием SIMD-инструкций (SIMD32): 13832 наносекунд.

Алгоритм "Нахождение обратной матрицы" (рисунок 3.3):

- Количество итераций: 4615211.
- Время выполнения для неоптимизированной версии на скалярных операциях (Scalar32): 218 наносекунд.
- Время выполнения для оптимизированной версии с использованием SIMD-инструкций (SIMD32): 136 наносекунд.

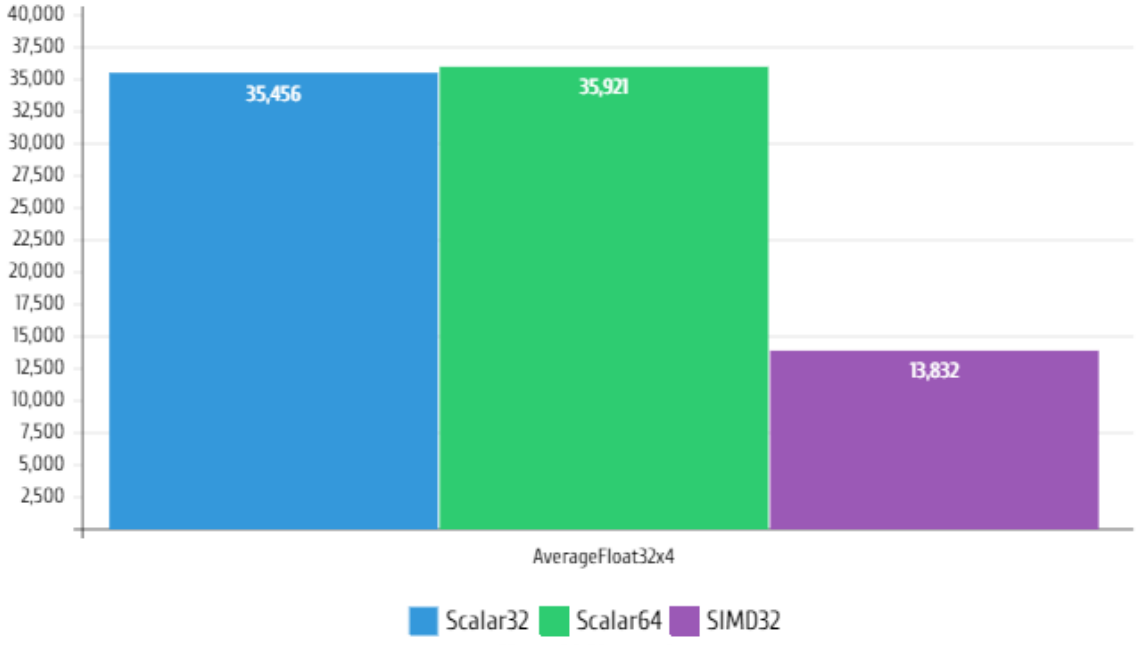

Рисунок 3.2 – Диаграмма времени выполнения алгоритмов вычисления среднего значения вектора

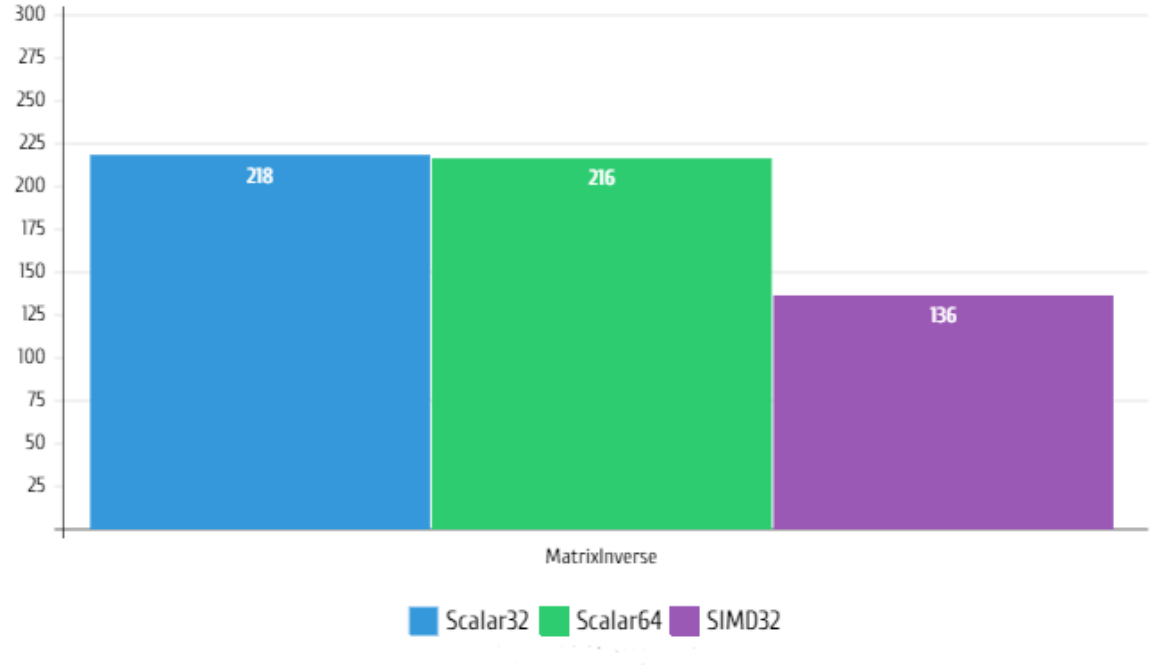

Рисунок 3.3 – Диаграмма времени выполнения алгоритмов нахождения обратной матрицы

Алгоритм "Умножение матрицы на вектор" (рисунок 3.4):

- Количество итераций: 18267874.
- Время выполнения для неоптимизированной версии на скалярных операциях (Scalar32): 53 наносекунд.
- Время выполнения для оптимизированной версии с использованием SIMD-инструкций (SIMD32): 52 наносекунд.

Алгоритм "Транспонирование матрицы" (рисунок 3.5):

- Количество итераций: 49344752.
- Время выполнения для неоптимизированной версии на скалярных операциях (Scalar32): 20 наносекунд.
- Время выполнения для оптимизированной версии с использованием SIMD-инструкций (SIMD32): 14 наносекунд.

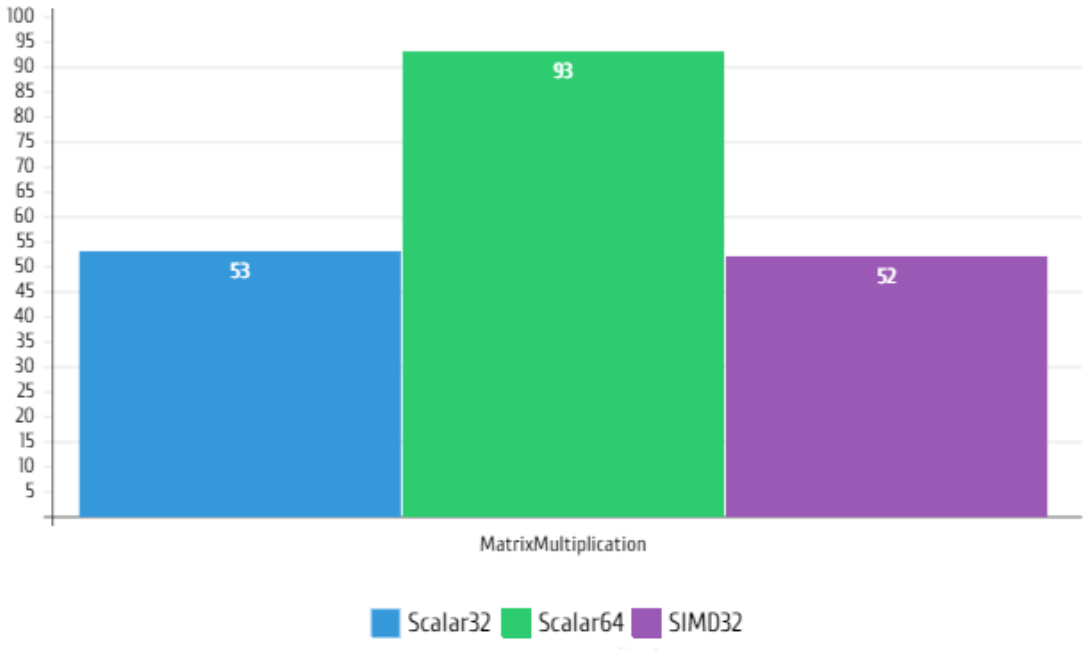

Рисунок 3.4 – Диаграмма времени выполнения алгоритмов умножения матрицы на вектор

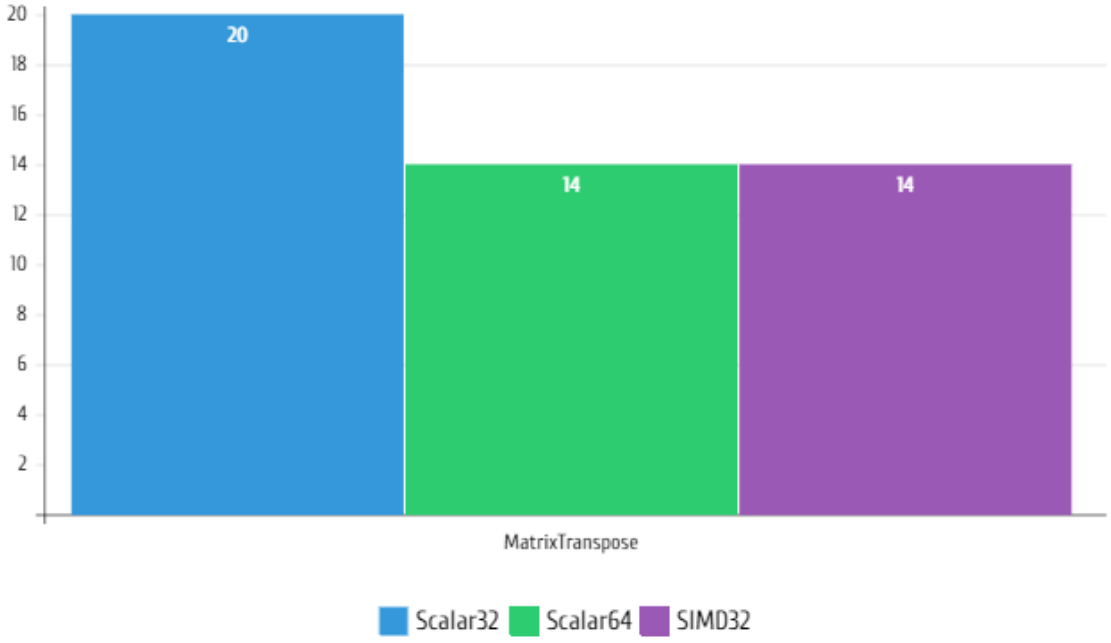

Рисунок 3.5 – Диаграмма времени выполнения алгоритмов транспонирования матрицы

Алгоритм " Преобразование вершины" (рисунок 3.6):

- Количество итераций: 1424695.
- Время выполнения для неоптимизированной версии на скалярных операциях (Scalar32): 701 наносекунд.
- Время выполнения для оптимизированной версии с использованием SIMD-инструкций (SIMD32): 61 наносекунд.

В большинстве алгоритмах оптимизированные версии, использующие SIMD-инструкции, демонстрируют значительное улучшение производительности по сравнению с неоптимизированными версиями на скалярных операциях. Это свидетельствует о эффективности векторизации и использовании SIMD-инструкций для ускорения вычислений.

Размер выигрыша в производительности может варьироваться в зависимости от конкретного алгоритма. Например, алгоритм "Вычисление среднего значения вектора" демонстрирует более значительное улучшение производительности (более чем в 2 раза), в то время как алгоритмы "Умножение матрицы на вектор" и "Транспонирование матрицы" имеют небольшие раз-

личия во времени выполнения между неоптимизированной и оптимизированной версиями.

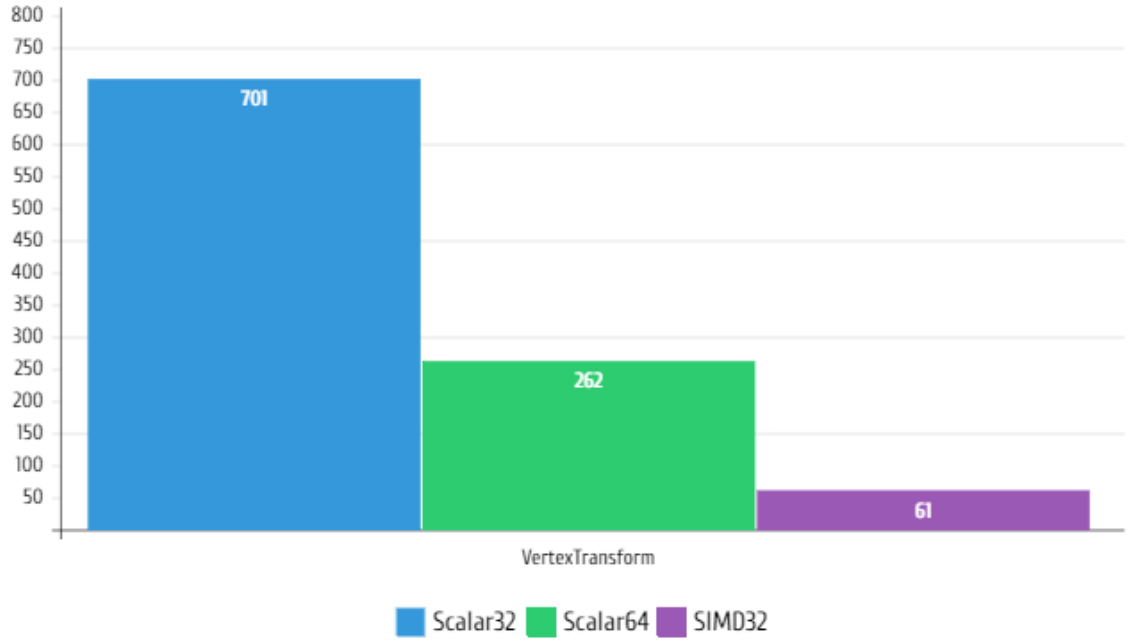

Рисунок 3.6 – Диаграмма времени выполнения алгоритмов преобразования вершины

Время выполнения для оптимизированных версий алгоритмов может быть существенно меньше, что может иметь значительное практическое значение в случаях, где требуется высокая производительность и быстрая обработка данных.

4 Нормативно-правовое регулирование интеллектуальной деятельности

4.1 Основы нормативно-правового регулирования результатов интеллектуальной деятельности

В основе нормативно-правового регулирования результатов интеллектуальной деятельности лежит понятие авторского права [7]. Согласно статье 1255 ГК авторские права это - интеллектуальные права на произведения науки, литературы и искусства [8]. Так же основы правового регулирования описаны в других нормативно-правовых документах действующих в Российской Федерации, таких как Кодекс об административных правонарушениях, Налоговый и Уголовный кодекс.

Автору произведения принадлежат следующие права:

1) исключительное право на произведение;
2) право авторства;
3) право автора на имя;
4) право на неприкосновенность произведения;
5) право на обнародование произведения.

Статья 1261 ГК определяет программу для ЭВМ как представленную в объективной форме совокупность данных и команд, предназначенных для функционирования ЭВМ и других компьютерных устройств в целях получения определенного результата, включая подготовительные материалы, полученные в ходе разработки программы для ЭВМ, и порождаемые ею аудиовизуальные отображения. Так же статьи 1274, 1275 ГК регламентируют использование произведений (в том числе программ для ЭВМ) в информационных, научных, учебных или культурных целях и свободное использование произведения библиотеками, архивами и образовательными организациями. Со-

гласно ст. 1259 ГК программы для ЭВМ так же являются объектами авторских прав, которые охраняются как литературные произведения.

Кодекс об административных правонарушениях в части статьи 13.31 регулирует обязанности организатора распространения информации в сети "Интернет" уведомить уполномоченный федеральный орган исполнительной власти о начале осуществления деятельности по обеспечению функционирования информационных систем и (или) программ для электронных вычислительных машин, которые предназначены и (или) используются для приема, передачи, доставки и (или) обработки электронных сообщений пользователей сети "Интернет" [9]

В Налоговом кодексе описано, что исключительные права на программы для электронных вычислительных машин и базы данных не подлежат налогообложению на территории РФ (ст. 149 НК).[43] Если доходы были получены в виде исключительных прав на изобретения, полезные модели, промышленные образцы, программы для электронных вычислительных машин, базы данных, топологии интегральных микросхем, секреты производства (ноу-хау), созданные в ходе реализации государственного контракта, которые переданы исполнителю этого государственного контракта его государственным заказчиком по договору о безвозмездном отчуждении, то при определении налоговой базы они не учитываются (ст. 251 НК). Статья 257 НК дает возможность признать исключительное право автора и иного правообладателя на использование программы для ЭВМ, базы данных как нематериального актива.

В Уголовном кодексе прописана уголовная ответственность за правонарушения, касающиеся нарушения авторских и смежных прав (ст. 146). Так, например присвоение авторства (плагиат), если это деяние причинило крупный ущерб автору или иному правообладателю, наказывается штрафом в размере до двухсот тысяч рублей или в размере заработной платы или иного дохода осужденного за период до восемнадцати месяцев, либо обязательными работами на срок до четырехсот восьмидесяти часов, либо исправитель-

ными работами на срок до одного года, либо арестом на срок до шести месяцев. (ч.1 ст. 146) [10]

4.2 Описание объекта исследования

Результатами интеллектуальной деятельности и приравненными к ним средствами индивидуализации юридических лиц, товаров, работ, услуг и предприятий, которым предоставляется правовая охрана (интеллектуальной собственностью), являются:

− произведения науки, литературы и искусства;

− программы для электронных вычислительных машин (программы для ЭВМ);

− базы данных;

− исполнения;

− фонограммы;

− сообщение в эфир или по кабелю радио- или телепередач (вещание организаций эфирного или кабельного вещания);

− изобретения;

− полезные модели;

− промышленные образцы;

− селекционные достижения;

− топологии интегральных микросхем;

− секреты производства (ноу-хау);

− фирменные наименования;

− товарные знаки и знаки обслуживания;

− наименования мест происхождения товаров;

− коммерческие обозначения.

В ходе анализа ВКР было определено, что результат написания ВКР может быть отнесен к особого вида нематериальным активам, которым потребуется правовая защита.

В результате выполнения данной дипломной работы была разработана и реализована программа, предназначенная для анализа и оптимизации векторных SIMD-инструкций с целью векторизации кода. Эта программа представляет собой модель, которая может быть использована как в учебных целях, для отработки навыков и знаний о применении SIMD-инструкций, так и в коммерческих целях, для сокращения временных и экономических затрат на разработку и оптимизацию программного кода, использующего векторные вычисления.

Следовательно, объектом исследования дополнительного раздела будет являться программа для ЭВМ.

4.3 Правовая защита объекта исследования

В данной работе объектом исследования является программа для ЭВМ. Программа для ЭВМ – объективная форма предоставления совокупности данных и команд, предназначенных для функционирования ЭВМ и других компьютерных устройств с целью получения определенного результата. Как и любому другому результату интеллектуальной деятельности, ему требуется правовая защита. Права авторов и правообладателей регулярно нарушаются, хотя существует достаточное количество различных инструментов для сотрудничества в сфере интеллектуальной собственности.

Гражданский кодекс содержит термин "правовая охрана" и "защита" применительно к результатам интеллектуальной деятельности и средствам индивидуализации. Эти два понятия законодателем разграничиваются.

По смыслу нормативного регулирования под охраной права понимается установление правового режима для осуществления интеллектуальных прав, тогда как защита — это меры, преимущественно применяемые в случае

нарушения прав. Охрана прав представляет собой общий правовой порядок в сфере интеллектуальной собственности, закрепленный в нормативных актах.

Государственная поддержка в области охраны объектов интеллектуальной собственности осуществляется посредством специальных органов государственной власти. В их числе Федеральная служба по интеллектуальной собственности (Роспатент), которая осуществляет функции по контролю и надзору в сфере правовой охраны и использования объектов интеллектуальной собственности, патентов и товарных знаков и результатов интеллектуальной деятельности.

Органом судебной власти по защите интеллектуальных прав в случае их нарушения является Арбитражный суд (в том числе Суд по интеллектуальным правам), а также суды общей юрисдикции.

Необходимо различать внесудебные и досудебные способы урегулирования споров. Внесудебные способы — это переговоры, участие в которых для сторон исключительно добровольное, а это означает, что они в любой момент могут обратиться в суд. Досудебные способы — это обязательные средства урегулирования спора, без которых чаще всего обращение в суд невозможно.

Существуют административно-правовая, гражданско-правовая и уголовно-правовая формы защиты.

Административно-правовая защита интеллектуальной собственности

Административно-правовые нормы, охраняющие интеллектуальные права, сконцентрированы в КоАП. Санкциями выступают штраф и конфискация контрафактных экземпляров произведения или фонограммы, материалов и оборудования, используемых для их воспроизведения.

Административная ответственность применяется за совершение следующих нарушений авторских и смежных прав:

– нарушение авторских и смежных прав, изобретательских и патентных прав (ст. 7.12 КоАП);

– нарушение установленного порядка патентования объектов промышленной собственности в иностранных государствах (ст. 7.28 КоАП);

– незаконное использование средств индивидуализации товаров, работ, услуг (ст. 14.10 КоАП);

– недобросовестная конкуренция в отношении или с использованием интеллектуальной собственности (ст. 14.33 КоАП).

Например, статья 7.12 КоАП "Нарушение авторских и смежных прав, изобретательских и патентных прав" устанавливает возможность административного взыскания в случаях ввоза, продажи, сдачи в прокат и иного незаконного использования экземпляров произведения или фонограмм в целях извлечения дохода при одном из следующих условий (либо совокупность таких условий):

– экземпляры являются контрафактными;

– на экземплярах указана ложная информация об их изготовителях или о местах производства;

– иным образом нарушаются интеллектуальные права с целью извлечения дохода.

Гражданско-правовая защита интеллектуальной собственности

Гражданско-правовая защита интеллектуальной собственности используется для восстановления нарушенного права и/или взыскания компенсации.

Способы защиты, которые применяются при защите прав, существенно различаются в зависимости от того, какие именно права нарушены.

В случае нарушения личных неимущественных прав автора их защита осуществляется следующими способами:

– признание права;

– восстановление положения, существовавшего до нарушения права;

– пресечение действий, нарушающих право или создающих угрозу его нарушения;

– компенсация морального вреда;

– публикация решения суда о допущенном нарушении.

Защита исключительных прав осуществляется путем предъявления следующих требований (ст. 1252 ГК):

– о признании права — к лицу, которое отрицает или иным образом не признает право, нарушая тем самым интересы правообладателя;

– о пресечении действий, нарушающих право или создающих угрозу его нарушения;

– о возмещении убытков или выплате компенсации — к лицу, неправомерно использовавшему результат интеллектуальной деятельности или средство индивидуализации без заключения соглашения с правообладателем;

– об изъятии материального носителя;

– о публикации решения суда о допущенном нарушении с указанием действительного правообладателя.

Защита личных неимущественных прав автора осуществляется вне зависимости от вины их нарушителя и независимо от нарушения имущественных интересов автора.

Уголовно-правовая защита интеллектуальной собственности

Под охраной действующего уголовного закона находятся практически все объекты интеллектуальной собственности, а именно:

– объекты авторских и смежных прав (ст. 146 УК РФ);

– объекты патентных прав (ст. 147 УК РФ);

– товарные знаки и знаки обслуживания, наименование места происхождения товара (ст. 180 УК РФ).

Данный способ защиты связан с применением уголовных наказаний, перечень которых закреплен в Уголовном кодексе, и предполагает обращение правообладателя (автора) с заявлением о привлечении к уголовной ответственности в уполномоченные органы.

Гражданский иск в уголовном процессе является дополнительным механизмом защиты имущественных прав потерпевших от преступлений против интеллектуальной собственности. Иск подается в процессе расследования уголовного дела.

Специфика и преимущества гражданского иска в уголовном процессе в том, что, в отличие от гражданского процесса, где каждая сторона должна доказать те обстоятельства, на которые она ссылается, сбор и фиксацию доказательств в уголовном процессе осуществляют органы предварительного следствия. Кроме того, с большей долей вероятности обвиняемый будет стараться возместить причиненный ущерб, так как это может привести к смягчению наказания либо вообще освобождению от него. При этом нужно учитывать, что не все уголовные дела доходят до суда с обвинительным заключением, по части проверочных материалов могут быть вынесены постановления об отказе в возбуждении уголовного дела. Подобные ситуации возникают в связи с наличием проблем квалификации, труднодоказуемости реального ущерба, сложности оценки объекта интеллектуальной собственности и т.д.

ЗАКЛЮЧЕНИЕ

В данной дипломной работе был проведен анализ и оптимизация векторных SIMD-инструкций для векторизации кода. Целью работы было исследование эффективности использования SIMD-инструкций в различных алгоритмах и оценка их влияния на производительность.

В рамках работы были выполнены следующие задачи:

1. Изучены основные принципы работы и преимущества SIMD-инструкций. Были рассмотрены основные типы инструкций, такие как SSE, AVX и NEON, и их применение в оптимизации кода для параллельной обработки данных.
2. Проведен сравнительный анализ производительности алгоритмов, включающий неоптимизированные и оптимизированные версии с использованием SIMD-инструкций. Для этого были измерены времена выполнения алгоритмов и сравнены результаты.
3. Разработаны и реализованы оптимизированные версии алгоритмов, использующие SIMD-инструкции. Были применены соответствующие оптимизации, такие как векторизация циклов и использование специфичных инструкций для параллельной обработки данных.
4. Проведен анализ полученных результатов сравнительного анализа. Было выявлено, что использование SIMD-инструкций приводит к значительному улучшению производительности во многих случаях, особенно при работе с массивами данных и выполнении операций над векторами.

В результате работы было установлено, что применение векторных SIMD-инструкций позволяет значительно ускорить выполнение алгоритмов и повысить общую производительность кода. Оптимизированные версии алгоритмов с использованием SIMD-инструкций демонстрируют существенное сокращение времени выполнения и улучшение эффективности работы с данными.

Однако следует отметить, что эффективность применения SIMD-инструкций может зависеть от конкретных алгоритмов, типов данных и характеристик аппаратной платформы. Поэтому важно проводить детальный анализ и тестирование при выборе оптимального подхода к векторизации кода.

В заключение, проведенный анализ и оптимизация векторных SIMD-инструкций подтверждают их значимость и потенциал для улучшения производительности и эффективности вычислений. Применение SIMD-инструкций может быть особенно полезным при работе с большими объемами данных и требовательных вычислительных задачах.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Intel Architectures Optimization Reference Manual [Электронный ресурс] // Software.intel.com: [сайт]. URL: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf. (Дата обращения: 20.04.2023).

2. Optimizing software in C++: An optimization guide for Windows, Linux, and Mac platforms [Электронный ресурс] // Agner.org: [сайт]. URL: https://www.agner.org/optimize/ (дата обращения: 21.04.2023).

3. Introduction to SIMD Architecture and SSE Instructions [Электронный ресурс] // Software.intel.com: [сайт]. URL: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-simd-architecture-and-sse-instructions.html (дата обращения: 24.04.2023).

4. Vectorization: What It Is and Why It Matters [Электронный ресурс] // Codeproject.com: [сайт]. URL: https://developer.ibm.com/articles/what-is-vectorization/ (дата обращения: 27.04.2023).

5. Vectorization in C++ with Intel C++ Compiler [Электронный ресурс] // Codeproject.com: [сайт]. URL: https://www.codeproject.com/Articles/874396/Vectorization-in-Cplusplus-with-Intel-Cplusplus-Co (дата обращения: 28.04.2023).

6. Introduction to SIMD Programming [Электронный ресурс] // Techopedia.com: [сайт]. URL: https://www.techopedia.com/definition/29561/single-instruction-multiple-data-simd (дата обращения: 29.04.2023).

7. Нормативно-правовое регулирование результатов интеллектуальной деятельности: учебно-методическое пособие по выполнению дополнительного раздела выпускных квалификационных работ бакалавров / сост.: М.Н. Магомедов, М.В. Чигирь. СПб.: Изд-во СПбГЭТУ “ЛЭТИ”, 2019. 20 с.

8. Гражданский кодекс Российской Федерации (часть четвертая) от 18.12.2006 N 230-ФЗ (ред. от 11.06.2021) (с изм. и доп., вступ. в силу с 01.01.2022) (Статья 1255,1259, 1261, 1274, 1275, 1280).

9. Кодекс Российской Федерации об административных правонарушениях от 30.12.2001 N 195-ФЗ (ред. от 16.04.2022) (с изм. и доп., вступ. в силу с 27.04.2022) (Статья 7.12, 7.28, 13.31, 14.20).

10. Налоговый кодекс Российской Федерации (часть вторая) от 05.08.2000 N 117-ФЗ (ред. от 01.05.2022) (с изм. и доп., вступ. в силу с 16.05.2022) (Статья 149, 251, 257).

11. Налоговый кодекс Российской Федерации (часть вторая) от 05.08.2000 N 117-ФЗ (ред. от 01.05.2022) (с изм. и доп., вступ. в силу с 16.05.2022) (Статья 149, 251, 257).

ПРИЛОЖЕНИЕ А

Фрагменты исходного кода

Файл run.cpp. Запуск и выполнение набора бенчмарков для анализа производительности оптимизированных алгоритмов.

```
#include <stdio.h>
#include "base.h"
#include "kernel-template.h"
#include "average_float32x4.h"
#include "matrix_multiplication.h"
#include "vertex_transform.h"
#include "matrix_transpose.h"
#include "matrix_inverse.h"

// Функции для вывода результатов, ошибок и оценки производительности
void PrintResult(char* str)
{
    printf("%s\n", str);
}

void PrintError(char* str)
{
    printf("%s\n", str);
}

void PrintScore(char* str)
{
    printf("%s\n", str);
}

int main()
{
    // Создание объекта outputFunctions для передачи в качестве аргумента в
RunAll
    Base::OutputFunctions outputFunctions(PrintResult, PrintError,
PrintScore);

    // Создание объектов для каждого бенчмарка, выполнение которых будет про-
изведено в конструкторе
    KernelTemplate    kernelTemplate;
    AverageFloat32x4  averageFloat32x4;
    MatrixMultiplication matrixMultiplication;
    VertexTransform vertexTransform;
    MatrixTranspose matrixTranspose;
    MatrixInverse matrixInverse;

    // Запуск всех объявленных бенчмарков
    Base::benchmarks.RunAll(outputFunctions, true);

    return 0;
}
```

Файл base.h. Определение основных классов, структур и функций, используемых в программе для выполнения бенчмарков.

```
#pragma once
#ifndef _BASE_H
#define _BASE_H

#include <string>
#include <stdint.h>
#include <emmintrin.h>
#include <xmmintrin.h>

using namespace std;

namespace Base
{
class OutputFunctions; // Класс для вывода результатов, ошибок и оценки производительности
class Configuration; // Класс для конфигурации бенчмарка
class Benchmark; // Класс для выполнения бенчмарка
class Benchmarks; // Класс для хранения и выполнения всех бенчмарков

typedef void     (*PrintFunction)(char *str); // Тип функции для вывода текста
typedef bool     (*InitFunction)(void); // Тип функции инициализации
typedef bool     (*CleanupFunction)(void); // Тип функции очистки
typedef uint64_t (*KernelFunction)(uint64_t n); // Тип функции ядра бенчмарка

template<typename X4, typename X>
class Lanes
{
private:
    union
    {
        X4  m128;
        X   lanes[4];
    } lanes;
public:
    Lanes(X4 m128)
    {
        lanes.m128 = m128;
    }
    X x()
    {
        return lanes.lanes[0];
    }
    X y()
    {
        return lanes.lanes[1];
    }
    X z()
    {
        return lanes.lanes[2];
    }
    X w()
    {
        return lanes.lanes[3];
    }
};

typedef union
```

```
{
    float  m128_f32[4];
    __m128 f32x4;
} M128;

#define M128_INIT(m128) Base::M128 m128##overlay; m128##overlay.f32x4 = m128
#define M128_X(m128) (m128##overlay.m128_f32[0])
#define M128_Y(m128) (m128##overlay.m128_f32[1])
#define M128_Z(m128) (m128##overlay.m128_f32[2])
#define M128_W(m128) (m128##overlay.m128_f32[3])

typedef union
{
    int  m128i_i32[4];
    __m128i i32x4;
} M128I;

#define M128I_INIT(m128i) Base::M128I m128i##overlay; m128i##overlay.i32x4 =
m128i
#define M128I_X(m128i) (m128i##overlay.m128i_i32[0])
#define M128I_Y(m128i) (m128i##overlay.m128i_i32[1])
#define M128I_Z(m128i) (m128i##overlay.m128i_i32[2])
#define M128I_W(m128i) (m128i##overlay.m128i_i32[3])

class OutputFunctions
{
public:
    OutputFunctions(PrintFunction PrintResult, PrintFunction PrintError,
PrintFunction PrintScore)
        : PrintResult(PrintResult)
        , PrintError(PrintError)
        , PrintScore(PrintScore)
    {
    }

    PrintFunction PrintResult; // Функция для вывода результатов
    PrintFunction PrintError; // Функция для вывода ошибок
    PrintFunction PrintScore; // Функция для вывода оценки производительности
};

class Configuration
{
public:
    Configuration(string        name,
                  InitFunction    Init,
                  CleanupFunction Cleanup,
                  KernelFunction  Simd,
                  KernelFunction  nonSimd32,
                  KernelFunction  nonSimd64,
                  uint64_t        iterations)
        : kernelName(name)
        , kernelInit(Init)
        , kernelCleanup(Cleanup)
        , kernelSimd(Simd)
        , kernelNonSimd32(nonSimd32)
        , kernelNonSimd64(nonSimd64)
        , kernelIterations(iterations)
    {
    }

    string           kernelName; // Название бенчмарка
    InitFunction     kernelInit; // Функция инициализации
    CleanupFunction  kernelCleanup; // Функция очистки
```

```
    KernelFunction   kernelSimd; // Функция ядра бенчмарка для SIMD
    KernelFunction   kernelNonSimd32; // Функция ядра бенчмарка для 32-битной
архитектуры
    KernelFunction   kernelNonSimd64; // Функция ядра бенчмарка для 64-битной
архитектуры
    uint64_t         kernelIterations; // Количество итераций бенчмарка
};

class Benchmarks
{
public:
    static void RunAll(OutputFunctions& outputFunctions, bool useAutoItera-
tions); // Выполнение всех бенчмарков
    static void Add(Benchmark* benchmark); // Добавление бенчмарка
};

extern Benchmarks benchmarks; // Объект для хранения и выполнения всех
бенчмарков

class Benchmark
{
public:
    Benchmark(Configuration* config)
        : config(config)
        , useAutoIterations(false)
        , initOk(true)
        , cleanupOk(true)
    {
        Base::benchmarks.Add(this);
    }

    Configuration* config; // Конфигурация бенчмарка
    bool           useAutoIterations; // Флаг использования автоматического
количества итераций
    bool           initOk; // Флаг успешной инициализации
    bool           cleanupOk; // Флаг успешной очистки
    uint64_t       autoIterations; // Автоматическое количество итераций
    uint64_t       actualIterations; // Фактическое количество итераций
    uint64_t       simdTime; // Время выполнения SIMD-алгоритма
    uint64_t       nonSimd32Time; // Время выполнения алгоритма для 32-битной
архитектуры
    uint64_t       nonSimd64Time; // Время выполнения алгоритма для 64-битной
архитектуры
};
} //namespace Base
#endif
```

Файл averagefloat32x4.h. Определение класса AverageFloat32x4, который представляет вычисления среднего значения элементов вектора типа float.

```
#pragma once
#ifndef _AVERAGEFLOAT32X4_H
#define _AVERAGEFLOAT32X4_H

#include <stdio.h>
#include <stdint.h>
#include <math.h>
```

```
#include "base.h"

class AverageFloat32x4 : public Base::Benchmark
{
public:
    AverageFloat32x4()
        : Base::Benchmark(
            new Base::Configuration(
                string("AverageFloat32x4"), // Название бенчмарка
                InitArray, // Функция инициализации массива
                Cleanup, // Функция очистки
                SimdAverage, // Функция ядра SIMD
                Average32, // Функция ядра для 32-битной архитектуры
                Average64, // Функция ядра для 64-битной архитектуры
                1000)) {} // Количество итераций

    static uint64_t preventOptimize;

    static const uint32_t length = 10000; // Длина массива
    static float a[length];

    static bool SanityCheck()
    {
        float simdVal = SimdAverageKernel();
        float nonSimd32Val = NonSimdAverageKernel32();
        float nonSimd64Val = (float)NonSimdAverageKernel64();
        return fabs(simdVal - nonSimd32Val) < 0.0001 &&
            fabs(simdVal - nonSimd64Val) < 0.0001;
    }

    static bool InitArray()
    {
        for (uint32_t i = 0; i < length; ++i)
        {
            a[i] = 0.1f;
        }
        // Проверка, что две функции ядра дают одинаковый результат.
        return SanityCheck();
    }

    static bool Cleanup()
    {
        return SanityCheck();
    };

    static float SimdAverageKernel()
    {
        preventOptimize++;
        __m128 sumx4 = _mm_set_ps1(0.0);
        for (uint32_t j = 0, l = length; j < l; j = j + 4)
        {
            sumx4 = _mm_add_ps(sumx4, _mm_loadu_ps(&(a[j])));
        }
        Base::Lanes<__m128, float> lanes(sumx4);
        return (lanes.x() + lanes.y() + lanes.z() + lanes.w()) / length;
    }

    static float NonSimdAverageKernel32()
    {
        preventOptimize++;
        float sum = 0.0;
        for (uint32_t j = 0, l = length; j < l; ++j)
        {
```

```
            sum += a[j];
        }
        return sum / length;
    }

    static double NonSimdAverageKernel64()
    {
        preventOptimize++;
        double sum = 0.0;
        for (uint32_t j = 0, l = length; j < l; ++j)
        {
            sum += (double)a[j];
        }
        return sum / length;
    }

    static uint64_t SimdAverage(uint64_t n)
    {
        float val;
        for (uint64_t i = 0; i < n; ++i)
        {
            val = SimdAverageKernel();
        }
        return (uint64_t)val;
    };

    static uint64_t Average32(uint64_t n)
    {
        float val;
        for (uint64_t i = 0; i < n; ++i)
        {
            val = NonSimdAverageKernel32();
        }
        return (uint64_t)val;
    };

    static uint64_t Average64(uint64_t n)
    {
        double val;
        for (uint64_t i = 0; i < n; ++i)
        {
            val = NonSimdAverageKernel64();
        }
        return (uint64_t)val;
    };
};

uint64_t AverageFloat32x4::preventOptimize = 0;
float AverageFloat32x4::a[AverageFloat32x4::length];

#endif
```

Файл matrix_multiplication.h. Определение класса MatrixMultiplication, который представляет умножение матрицы на вектор.

```
#pragma once
#ifndef _MATRIX_MULTIPLICATION_H
#define _MATRIX_MULTIPLICATION_H
```

```
#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include "base.h"

class MatrixMultiplication : public Base::Benchmark
{
public:
    MatrixMultiplication()
        : Base::Benchmark(
            new Base::Configuration(
                string("MatrixMultiplication"),
                Init,
                Cleanup,
                SimdMultiply,
                Multiply32,
                Multiply64,
                1000)) {}

    static uint64_t preventOptimize;

    static float* t1;
    static float* t2;
    static float* out;
    static float* t1x4;
    static float* t2x4;
    static float* outx4;

    static bool Equals(const float* t1, const float* t2)
    {
        return (t1[0] == t2[0]) &&
            (t1[1] == t2[1]) &&
            (t1[2] == t2[2]) &&
            (t1[3] == t2[3]) &&
            (t1[4] == t2[4]) &&
            (t1[5] == t2[5]) &&
            (t1[6] == t2[6]) &&
            (t1[7] == t2[7]) &&
            (t1[8] == t2[8]) &&
            (t1[9] == t2[9]) &&
            (t1[10] == t2[10]) &&
            (t1[11] == t2[11]) &&
            (t1[12] == t2[12]) &&
            (t1[13] == t2[13]) &&
            (t1[14] == t2[14]) &&
            (t1[15] == t2[15]);
    }

    static bool Init()
    {
        t1 = new float[16];
        t2 = new float[16];
        t1x4 = new float[16];
        t2x4 = new float[16];
        out = new float[16];
        outx4 = new float[16];

        t1[0] = 1.0;
        t1[5] = 1.0;
        t1[10] = 1.0;
        t1[15] = 1.0;
```

```
        t2[0] = 2.0;
        t2[5] = 2.0;
        t2[10] = 2.0;
        t2[15] = 2.0;

        t1x4[0] = 1.0;
        t1x4[5] = 1.0;
        t1x4[10] = 1.0;
        t1x4[15] = 1.0;

        t2x4[0] = 2.0;
        t2x4[5] = 2.0;
        t2x4[10] = 2.0;
        t2x4[15] = 2.0;

        Multiply32(1);
        SimdMultiply(1);

        return Equals(t1, t1x4) && Equals(t2, t2x4) && Equals(out, outx4);
    }

    static bool Cleanup()
    {
        t1[0] = 1.0;
        t1[5] = 1.0;
        t1[10] = 1.0;
        t1[15] = 1.0;

        t2[0] = 2.0;
        t2[5] = 2.0;
        t2[10] = 2.0;
        t2[15] = 2.0;

        t1x4[0] = 1.0;
        t1x4[5] = 1.0;
        t1x4[10] = 1.0;
        t1x4[15] = 1.0;

        t2x4[0] = 2.0;
        t2x4[5] = 2.0;
        t2x4[10] = 2.0;
        t2x4[15] = 2.0;

        Multiply32(1);
        SimdMultiply(1);

        bool ret = Equals(t1, t1x4) && Equals(t2, t2x4) && Equals(out,
outx4);

        delete[] t1;
        delete[] t2;
        delete[] t1x4;
        delete[] t2x4;
        delete[] out;
        delete[] outx4;

        return ret;
    };

    static uint64_t Multiply32(uint64_t n)
    {
        for (uint64_t i = 0; i < n; i++)
        {
```

```
        float a00 = t1[0];
        float a01 = t1[1];
        float a02 = t1[2];
        float a03 = t1[3];
        float a10 = t1[4];
        float a11 = t1[5];
        float a12 = t1[6];
        float a13 = t1[7];
        float a20 = t1[8];
        float a21 = t1[9];
        float a22 = t1[10];
        float a23 = t1[11];
        float a30 = t1[12];
        float a31 = t1[13];
        float a32 = t1[14];
        float a33 = t1[15];

        float b0 = t2[0];
        float b1 = t2[1];
        float b2 = t2[2];
        float b3 = t2[3];
        out[0] = b0 * a00 + b1 * a10 + b2 * a20 + b3 * a30;
        out[1] = b0 * a01 + b1 * a11 + b2 * a21 + b3 * a31;
        out[2] = b0 * a02 + b1 * a12 + b2 * a22 + b3 * a32;
        out[3] = b0 * a03 + b1 * a13 + b2 * a23 + b3 * a33;

        b0 = t2[4];
        b1 = t2[5];
        b2 = t2[6];
        b3 = t2[7];
        out[4] = b0 * a00 + b1 * a10 + b2 * a20 + b3 * a30;
        out[5] = b0 * a01 + b1 * a11 + b2 * a21 + b3 * a31;
        out[6] = b0 * a02 + b1 * a12 + b2 * a22 + b3 * a32;
        out[7] = b0 * a03 + b1 * a13 + b2 * a23 + b3 * a33;

        b0 = t2[8];
        b1 = t2[9];
        b2 = t2[10];
        b3 = t2[11];
        out[8] = b0 * a00 + b1 * a10 + b2 * a20 + b3 * a30;
        out[9] = b0 * a01 + b1 * a11 + b2 * a21 + b3 * a31;
        out[10] = b0 * a02 + b1 * a12 + b2 * a22 + b3 * a32;
        out[11] = b0 * a03 + b1 * a13 + b2 * a23 + b3 * a33;

        b0 = t2[12];
        b1 = t2[13];
        b2 = t2[14];
        b3 = t2[15];
        out[12] = b0 * a00 + b1 * a10 + b2 * a20 + b3 * a30;
        out[13] = b0 * a01 + b1 * a11 + b2 * a21 + b3 * a31;
        out[14] = b0 * a02 + b1 * a12 + b2 * a22 + b3 * a32;
        out[15] = b0 * a03 + b1 * a13 + b2 * a23 + b3 * a33;
        preventOptimize++;
    }
    return preventOptimize;
}

static uint64_t Multiply64(uint64_t n)
{
    for (uint64_t i = 0; i < n; i++)
    {
        double a00 = t1[0];
        double a01 = t1[1];
```

```
            double a02 = t1[2];
            double a03 = t1[3];
            double a10 = t1[4];
            double a11 = t1[5];
            double a12 = t1[6];
            double a13 = t1[7];
            double a20 = t1[8];
            double a21 = t1[9];
            double a22 = t1[10];
            double a23 = t1[11];
            double a30 = t1[12];
            double a31 = t1[13];
            double a32 = t1[14];
            double a33 = t1[15];

            double b0 = t2[0];
            double b1 = t2[1];
            double b2 = t2[2];
            double b3 = t2[3];
            out[0] = b0 * a00 + b1 * a10 + b2 * a20 + b3 * a30;
            out[1] = b0 * a01 + b1 * a11 + b2 * a21 + b3 * a31;
            out[2] = b0 * a02 + b1 * a12 + b2 * a22 + b3 * a32;
            out[3] = b0 * a03 + b1 * a13 + b2 * a23 + b3 * a33;

            b0 = t2[4];
            b1 = t2[5];
            b2 = t2[6];
            b3 = t2[7];
            out[4] = b0 * a00 + b1 * a10 + b2 * a20 + b3 * a30;
            out[5] = b0 * a01 + b1 * a11 + b2 * a21 + b3 * a31;
            out[6] = b0 * a02 + b1 * a12 + b2 * a22 + b3 * a32;
            out[7] = b0 * a03 + b1 * a13 + b2 * a23 + b3 * a33;

            b0 = t2[8];
            b1 = t2[9];
            b2 = t2[10];
            b3 = t2[11];
            out[8] = b0 * a00 + b1 * a10 + b2 * a20 + b3 * a30;
            out[9] = b0 * a01 + b1 * a11 + b2 * a21 + b3 * a31;
            out[10] = b0 * a02 + b1 * a12 + b2 * a22 + b3 * a32;
            out[11] = b0 * a03 + b1 * a13 + b2 * a23 + b3 * a33;

            b0 = t2[12];
            b1 = t2[13];
            b2 = t2[14];
            b3 = t2[15];
            out[12] = b0 * a00 + b1 * a10 + b2 * a20 + b3 * a30;
            out[13] = b0 * a01 + b1 * a11 + b2 * a21 + b3 * a31;
            out[14] = b0 * a02 + b1 * a12 + b2 * a22 + b3 * a32;
            out[15] = b0 * a03 + b1 * a13 + b2 * a23 + b3 * a33;
            preventOptimize++;
        }
        return preventOptimize;
    }

    static uint64_t SimdMultiply(uint64_t n)
    {
        for (uint64_t i = 0; i < n; i++)
        {
            __m128 a0 = _mm_loadu_ps(&t1x4[0]);
            __m128 a1 = _mm_loadu_ps(&t1x4[4]);
            __m128 a2 = _mm_loadu_ps(&t1x4[8]);
            __m128 a3 = _mm_loadu_ps(&t1x4[12]);
```

```
            __m128 b0 = _mm_loadu_ps(&t2x4[0]);
            _mm_storeu_ps(
                &outx4[0],
                _mm_add_ps(
                    _mm_mul_ps(_mm_shuffle_ps(b0, b0, _MM_SHUFFLE(0, 0, 0,
0)), a0),
                    _mm_add_ps(
                        _mm_mul_ps(_mm_shuffle_ps(b0, b0, _MM_SHUFFLE(1, 1,
1, 1)), a1),
                        _mm_add_ps(
                            _mm_mul_ps(_mm_shuffle_ps(b0, b0, _MM_SHUFFLE(2,
2, 2, 2)), a2),
                            _mm_mul_ps(_mm_shuffle_ps(b0, b0, _MM_SHUFFLE(3,
3, 3, 3)), a3)))));

            __m128 b1 = _mm_loadu_ps(&t2x4[4]);
            _mm_storeu_ps(
                &outx4[4],
                _mm_add_ps(
                    _mm_mul_ps(_mm_shuffle_ps(b1, b1, _MM_SHUFFLE(0, 0, 0,
0)), a0),
                    _mm_add_ps(
                        _mm_mul_ps(_mm_shuffle_ps(b1, b1, _MM_SHUFFLE(1, 1,
1, 1)), a1),
                        _mm_add_ps(
                            _mm_mul_ps(_mm_shuffle_ps(b1, b1, _MM_SHUFFLE(2,
2, 2, 2)), a2),
                            _mm_mul_ps(_mm_shuffle_ps(b1, b1, _MM_SHUFFLE(3,
3, 3, 3)), a3)))));

            __m128 b2 = _mm_loadu_ps(&t2x4[8]);
            _mm_storeu_ps(
                &outx4[8],
                _mm_add_ps(
                    _mm_mul_ps(_mm_shuffle_ps(b2, b2, _MM_SHUFFLE(0, 0, 0,
0)), a0),
                    _mm_add_ps(
                        _mm_mul_ps(_mm_shuffle_ps(b2, b2, _MM_SHUFFLE(1, 1,
1, 1)), a1),
                        _mm_add_ps(
                            _mm_mul_ps(_mm_shuffle_ps(b2, b2, _MM_SHUFFLE(2,
2, 2, 2)), a2),
                            _mm_mul_ps(_mm_shuffle_ps(b2, b2, _MM_SHUFFLE(3,
3, 3, 3)), a3)))));

            __m128 b3 = _mm_loadu_ps(&t2x4[12]);
            _mm_storeu_ps(
                &outx4[12],
                _mm_add_ps(
                    _mm_mul_ps(_mm_shuffle_ps(b3, b3, _MM_SHUFFLE(0, 0, 0,
0)), a0),
                    _mm_add_ps(
                        _mm_mul_ps(_mm_shuffle_ps(b3, b3, _MM_SHUFFLE(1, 1,
1, 1)), a1),
                        _mm_add_ps(
                            _mm_mul_ps(_mm_shuffle_ps(b3, b3, _MM_SHUFFLE(2,
2, 2, 2)), a2),
                            _mm_mul_ps(_mm_shuffle_ps(b3, b3, _MM_SHUFFLE(3,
3, 3, 3)), a3)))));
            preventOptimize++;
        }
        return preventOptimize;
```

```
    }
};

uint64_t MatrixMultiplication::preventOptimize = 0;

float* MatrixMultiplication::t1 = NULL;
float* MatrixMultiplication::t2 = NULL;
float* MatrixMultiplication::t1x4 = NULL;
float* MatrixMultiplication::t2x4 = NULL;
float* MatrixMultiplication::out = NULL;
float* MatrixMultiplication::outx4 = NULL;

#endif
```

Файл matrix_transpose.h. Определение класса MatrixTranspose, который представляет транспонирование матрицы.

```
#pragma once
#ifndef _MATRIX_TRANSPOSE_H
#define _MATRIX_TRANSPOSE_H

#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include "base.h"

class MatrixTranspose : public Base::Benchmark
{
public:
    MatrixTranspose()
        : Base::Benchmark(
            new Base::Configuration(
                string("MatrixTranspose"),
                Init,
                Cleanup,
                SimdTranspose,
                Transpose32,
                Transpose64,
                1000)) {}

    static uint64_t preventOptimize;

    static float* src;
    static float* srcx4;
    static float* dst;
    static float* dstx4;
    static float* tsrc;
    static float* tsrcx4;

    static void PrintMatrix(const float* matrix)
    {
        for (int r = 0; r < 4; ++r)
        {
            int ri = r * 4;
            for (int c = 0; c < 4; ++c)
            {
                float value = matrix[ri + c];
                printf("%f ", value);
            }
            printf("\n");
        }
```

```
    printf("\n");
}

static void InitMatrix(float* matrix, float* matrixTransposed)
{
    for (int r = 0; r < 4; ++r)
    {
        int r4 = 4 * r;
        for (int c = 0; c < 4; ++c)
        {
            matrix[r4 + c] = r4 + c;
            matrixTransposed[r + c * 4] = r4 + c;
        }
    }
}

static bool CompareEqualMatrix(const float* m1, const float* m2)
{
    for (int i = 0; i < 16; ++i)
    {
        if (m1[i] != m2[i])
        {
            return false;
        }
    }
    return true;
}

static bool Init()
{
    src = new float[16];
    srcx4 = src;
    dst = new float[16];
    dstx4 = dst;
    tsrc = new float[16];
    tsrcx4 = tsrc;

    InitMatrix(src, tsrc);
    Transpose32(1);

    if (!CompareEqualMatrix(tsrc, dst))
    {
        return false;
    }

    SimdTranspose(1);

    if (!CompareEqualMatrix(tsrc, dst))
    {
        return false;
    }

    return true;
}

static bool Cleanup()
{
    bool ret = true;
    InitMatrix(src, tsrc);
    Transpose32(1);

    if (!CompareEqualMatrix(tsrc, dst))
    {
```

```
        ret = false;
    }

    SimdTranspose(1);

    if (!CompareEqualMatrix(tsrc, dst))
    {
        ret = false;
    }

    delete[] src;
    delete[] dst;
    delete[] tsrc;

    return ret;
}

static uint64_t Transpose32(uint64_t n)
{
    for (uint64_t i = 0; i < n; ++i)
    {
        dst[0] = src[0];
        dst[1] = src[4];
        dst[2] = src[8];
        dst[3] = src[12];
        dst[4] = src[1];
        dst[5] = src[5];
        dst[6] = src[9];
        dst[7] = src[13];
        dst[8] = src[2];
        dst[9] = src[6];
        dst[10] = src[10];
        dst[11] = src[14];
        dst[12] = src[3];
        dst[13] = src[7];
        dst[14] = src[11];
        dst[15] = src[15];
        preventOptimize++;
    }
    return preventOptimize;
}

static uint64_t Transpose64(uint64_t n)
{
    for (uint64_t i = 0; i < n; ++i)
    {
        dst[0] = src[0];
        dst[1] = src[4];
        dst[2] = src[8];
        dst[3] = src[12];
        dst[4] = src[1];
        dst[5] = src[5];
        dst[6] = src[9];
        dst[7] = src[13];
        dst[8] = src[2];
        dst[9] = src[6];
        dst[10] = src[10];
        dst[11] = src[14];
        dst[12] = src[3];
        dst[13] = src[7];
        dst[14] = src[11];
        dst[15] = src[15];
        preventOptimize++;
```

```
        }
        return preventOptimize;
    }

    static uint64_t SimdTranspose(uint64_t n)
    {
        for (uint64_t i = 0; i < n; ++i)
        {
            __m128 src0 = _mm_loadu_ps(&srcx4[0]);
            __m128 src1 = _mm_loadu_ps(&srcx4[4]);
            __m128 src2 = _mm_loadu_ps(&srcx4[8]);
            __m128 src3 = _mm_loadu_ps(&srcx4[12]);

            __m128 tmp01 = _mm_shuffle_ps(src0, src1, _MM_SHUFFLE(1, 0, 1,
0));
            __m128 tmp23 = _mm_shuffle_ps(src2, src3, _MM_SHUFFLE(1, 0, 1,
0));
            __m128 dst0 = _mm_shuffle_ps(tmp01, tmp23, _MM_SHUFFLE(2, 0, 2,
0));
            __m128 dst1 = _mm_shuffle_ps(tmp01, tmp23, _MM_SHUFFLE(3, 1, 3,
1));

            tmp01 = _mm_shuffle_ps(src0, src1, _MM_SHUFFLE(3, 2, 3, 2));
            tmp23 = _mm_shuffle_ps(src2, src3, _MM_SHUFFLE(3, 2, 3, 2));
            __m128 dst2 = _mm_shuffle_ps(tmp01, tmp23, _MM_SHUFFLE(2, 0, 2,
0));
            __m128 dst3 = _mm_shuffle_ps(tmp01, tmp23, _MM_SHUFFLE(3, 1, 3,
1));

            _mm_storeu_ps(&dstx4[0], dst0);
            _mm_storeu_ps(&dstx4[4], dst1);
            _mm_storeu_ps(&dstx4[8], dst2);
            _mm_storeu_ps(&dstx4[12], dst3);

            preventOptimize++;
        }
        return preventOptimize;
    }
};

uint64_t MatrixTranspose::preventOptimize = 0;

float* MatrixTranspose::src = NULL;
float* MatrixTranspose::srcx4 = NULL;
float* MatrixTranspose::dst = NULL;
float* MatrixTranspose::dstx4 = NULL;
float* MatrixTranspose::tsrc = NULL;
float* MatrixTranspose::tsrcx4 = NULL;

#endif
```

Файл vertex_transform.h. Определение класса VertexTransform, который представляет преобразование вершины.

```
#pragma once
#ifndef _VERTEX_TRANSFORM_H
#define _VERTEX_TRANSFORM_H

#include <stdio.h>
#include <stdint.h>
#include <math.h>
```

```
#include "base.h"

class VertexTransform : public Base::Benchmark
{
public:
    VertexTransform()
        : Base::Benchmark(
            new Base::Configuration(
                string("VertexTransform"),
                Init,
                Cleanup,
                SimdVertextTransform,
                VertextTransform32,
                VertextTransform64,
                1000)) {}

    static uint64_t preventOptimize;

    static float* t;
    static float* v;
    static float* out;
    static float* tx4;
    static float* vx4;
    static float* outx4;

    static bool Init()
    {
        t = new float[16];
        v = new float[4];
        out = new float[4];
        tx4 = new float[16];
        vx4 = new float[4];
        outx4 = new float[4];

        t[0] = 1.0;
        t[5] = 1.0;
        t[10] = 1.0;
        t[15] = 1.0;
        v[0] = 1.0;
        v[1] = 2.0;
        v[2] = 3.0;
        v[3] = 1.0;

        tx4[0] = 1.0;
        tx4[5] = 1.0;
        tx4[10] = 1.0;
        tx4[15] = 1.0;
        vx4[0] = 1.0;
        vx4[1] = 2.0;
        vx4[2] = 3.0;
        vx4[3] = 1.0;

        SimdVertextTransform(1);
        VertextTransform32(1);
        return (outx4[0] == out[0]) && (outx4[1] == out[1]) &&
            (outx4[2] == out[2]) && (outx4[3] == out[3]);
    }

    static bool Cleanup()
    {
        t[0] = 1.0;
        t[5] = 1.0;
        t[10] = 1.0;
```

```
    t[15] = 1.0;
    v[0] = 1.0;
    v[1] = 2.0;
    v[2] = 3.0;
    v[3] = 1.0;

    tx4[0] = 1.0;
    tx4[5] = 1.0;
    tx4[10] = 1.0;
    tx4[15] = 1.0;
    vx4[0] = 1.0;
    vx4[1] = 2.0;
    vx4[2] = 3.0;
    vx4[3] = 1.0;

    SimdVertextTransform(1);
    VertextTransform32(1);
    bool ret = (outx4[0] == out[0]) && (outx4[1] == out[1]) &&
        (outx4[2] == out[2]) && (outx4[3] == out[3]);

    delete[] t;
    delete[] v;
    delete[] out;
    delete[] tx4;
    delete[] vx4;
    delete[] outx4;

    return ret;
}

static uint64_t VertextTransform32(uint64_t n)
{
    for (uint64_t i = 0; i < n; i++)
    {
        float x = v[0];
        float y = v[1];
        float z = v[2];
        float w = v[3];
        float m0 = t[0];
        float m4 = t[4];
        float m8 = t[8];
        float m12 = t[12];
        out[0] = (m0 * x + m4 * y + m8 * z + m12 * w);
        float m1 = t[1];
        float m5 = t[5];
        float m9 = t[9];
        float m13 = t[13];
        out[1] = (m1 * x + m5 * y + m9 * z + m13 * w);
        float m2 = t[2];
        float m6 = t[6];
        float m10 = t[10];
        float m14 = t[14];
        out[2] = (m2 * x + m6 * y + m10 * z + m14 * w);
        float m3 = t[3];
        float m7 = t[7];
        float m11 = t[11];
        float m15 = t[15];
        out[3] = (m3 * x + m7 * y + m11 * z + m15 * w);
        preventOptimize++;
    }
    return preventOptimize;
}
```

```
    static uint64_t VertextTransform64(uint64_t n)
    {
        for (uint64_t i = 0; i < n; i++)
        {
            double x = v[0];
            double y = v[1];
            double z = v[2];
            double w = v[3];
            double m0 = t[0];
            double m4 = t[4];
            double m8 = t[8];
            double m12 = t[12];
            out[0] = (m0 * x + m4 * y + m8 * z + m12 * w);
            double m1 = t[1];
            double m5 = t[5];
            double m9 = t[9];
            double m13 = t[13];
            out[1] = (m1 * x + m5 * y + m9 * z + m13 * w);
            double m2 = t[2];
            double m6 = t[6];
            double m10 = t[10];
            double m14 = t[14];
            out[2] = (m2 * x + m6 * y + m10 * z + m14 * w);
            double m3 = t[3];
            double m7 = t[7];
            double m11 = t[11];
            double m15 = t[15];
            out[3] = (m3 * x + m7 * y + m11 * z + m15 * w);
            preventOptimize++;
        }
        return preventOptimize;
    }

    static uint64_t SimdVertextTransform(uint64_t n)
    {
        for (uint64_t i = 0; i < n; i++)
        {
            __m128 z = _mm_set1_ps(0.0);
            __m128 v = _mm_loadu_ps(&vx4[0]);
            __m128 xxxx = _mm_shuffle_ps(v, v, _MM_SHUFFLE(0, 0, 0, 0));
            z = _mm_add_ps(z, _mm_mul_ps(xxxx, _mm_loadu_ps(&tx4[0])));
            __m128 yyyy = _mm_shuffle_ps(v, v, _MM_SHUFFLE(1, 1, 1, 1));
            z = _mm_add_ps(z, _mm_mul_ps(yyyy, _mm_loadu_ps(&tx4[4])));
            __m128 zzzz = _mm_shuffle_ps(v, v, _MM_SHUFFLE(2, 2, 2, 2));
            z = _mm_add_ps(z, _mm_mul_ps(zzzz, _mm_loadu_ps(&tx4[8])));
            __m128 wwww = _mm_shuffle_ps(v, v, _MM_SHUFFLE(3, 3, 3, 3));
            z = _mm_add_ps(z, _mm_mul_ps(wwww, _mm_loadu_ps(&tx4[12])));
            _mm_storeu_ps(&outx4[0], z);
            preventOptimize++;
        }
        return preventOptimize;
    }
};
uint64_t VertexTransform::preventOptimize = 0;

float* VertexTransform::t = NULL;
float* VertexTransform::v = NULL;
float* VertexTransform::out = NULL;
float* VertexTransform::tx4 = NULL;
float* VertexTransform::vx4 = NULL;
float* VertexTransform::outx4 = NULL;

#endif
```