1. **SIMD-РАСШИРЕНИЯ АРХИТЕКТУРЫ x86/x86-64**

SIMD-расширения (векторные расширения) были введены во многие стандартные архитектуры с целью повышения скорости обработки потоковых данных. Основная идея SIMD-вычислений заключается в одновременной обработке нескольких элементов данных (вектора) за одну команду.

**SIMD (Single Instruction, Multiple Data)** — позволяет выполнять одну и ту же операцию над несколькими элементами данных одновременно. Позволяет эффективно обрабатывать большие объемы данных, таких как изображения, звук, видео и численные вычисления.

**Зачем вводятся SIMD-расширения:**

* **Ускорение вычислений:** SIMD позволяет значительно ускорить выполнение операций, которые можно векторизовать (например, сложение массивов чисел).
  1. Расширение MMX (MultiMedia eXtension)

Первой SIMD-расширение в свой x86-процессор ввела фирма Intel – это было расширение MMX. Оно стало использоваться в процессорах Pentium MMX (расширение архитектуры Pentium или P5) и Pentium II (расширение архитектуры Pentium Pro или P6). Векторное расширение MMX работает с 64-битными регистрами MM0-MM7, логически расположенными на регистрах сопроцессора, и включает 57 новых команд для работы с ними. 64- битные регистры логически могут представляться как одно 64-битное, два 32-битных, четыре 16-битных или восемь 8-битных упакованных целых чисел.

Еще одна особенность технологии MMX – это целочисленная арифметика с насыщением, используемая, например, при обработке графики. В целочисленное арифметике с насыщением переполнение не является циклическим, как обычно, а вместо этого фиксируется минимальное или максимальное значение. Например, для 8-битного беззнакового целого x:

• обычная арифметика: x=254; x+=3; // результат x=1

• арифметика с насыщением: x=254; x+=3; // результат x=255

* 1. Расширение 3DNow!

Технология 3DNow! была введена фирмой AMD в процессорах K6-2. Это была первая технология, выполняющая потоковую обработку вещественных данных. Расширение работает с 64-битными регистрами MMX, которые теперь представляются как два 32-битных вещественных числа с одинарной точностью. Система команд расширена 21 новой командой, среди которых появилась команда предвыборки данных в кэш L1. В процессорах Athlon и Duron набор команд 3DNow! был несколько дополнен новыми командами для работы с вещественными числами, а также командами MMX и управления кэшированием.

* 1. Расширение SSE (Streaming SIMD Extension)

С процессором Intel Pentium III впервые появилось расширение SSE. Это расширение работает с независимым блоком из восьми 128-битных регистров XMM0-XMM7. Каждый регистр XMM представляет собой четыре упакованных 32-битных вещественных числа с одинарной точностью. Команды блока XMM позволяют выполнять как векторные (над всеми четырьмя значениями регистра), так и скалярные операции (только над одним самым младшим значением). Кроме команд для работы с блоком XMM в расширение SSE входят и дополнительные целочисленные команды для работы с регистрами MMX, а также команды управления кэшированием. В архитектуре x86-64 число регистров XMM было увеличено до 16-ти: XMM0-XMM15.

* 1. Расширение SSE2

В процессоре Intel Pentium4 набор команд получил очередное расширение – SSE2. Это расширение не добавило новых регистров, но позволило по-новому интерпретировать существующие регистры. Расширение SSE2 позволяет работать с 128-битными регистрами XMM как с парой упакованных 64-битных вещественных чисел двойной точности, а также с упакованными целыми числами: 16 байт, 8 слов, 4 двойных (32- битных) слова или 2 учетверенных (64-битных) слова. Соответственно, введены новые команды вещественной арифметики двойной точности и команды целочисленной арифметики: 128-разрядные для регистров XMM и 64-разрядные для регистров MMX. Ряд старых команд MMX распространили и на XMM (в 128-битном варианте). Кроме того, расширена поддержка управления кэшированием и порядком исполнения операций с памятью для многопоточных программ.

* 1. Расширения SSE3, SSSE3, SSE4, ...

В последующих процессорах Intel и AMD происходит дальнейшее расширение системы команд на регистрах MMX и XMM. Добавлены новые команды для ускорения обработки видео, текстовых данных. Особенно следует отметить появившуюся возможность горизонтальной работы с регистрами (выполнение операций с элементами одного вектора).

* 1. Расширение AVX (Advanced Vector Extensions)

В процессорах архитектуры Sandy Bridge от Intel и процессорах архитектуры Bulldozer от AMD векторные расширения сделали следующий большой шаг в развитии: появились новые векторные регистры YMM0- YMM15 размером 256 бит. Существующие ранее регистры XMM стали занимать младшую часть новых регистров. Среди особенностей расширения AVX есть поддержка трехоперандных операций вида (c = a OP b), а также менее строгие требования к выравниванию векторных данных в памяти.

**AVX-512:**

* + **Регистры:** 32 регистра по 512 бит (ZMM0-ZMM31).
  + **Типы данных:** 64 байта, 32 слова, 16 двойных слов, 8 четверных слова, 16 чисел с плавающей точкой одинарной точности (float), 8 чисел с плавающей точкой двойной точности (double).

1. **ИСПОЛЬЗОВАНИЕ SIMD-РАСШИРЕНИЙ В ПРОГРАММАХ НА ЯЗЫКЕ СИ**

Существует несколько способов, позволяющих реализовать возможности имеющихся SIMD-расширений в программах на языках высокого уровня. Условно их можно разделить на ручные, полуавтоматические, автоматические и с помощью готовых библиотек.

2.1. Использование вставок на ассемблере

Многие компиляторы языков Си и Си++ дают возможность вставлять в тело функции команды на ассемблере. Программисту, знакомому с ассемблером, это позволяет контролировать производительность программы в наибольшей степени среди всех описанных в этой лабораторной работе способов. Однако использование вставок затрудняет работу компилятора по оптимизации кода. Кроме этого, теряется переносимость кода, так как: 1) команды во вставке рассчитаны на некоторую конкретную архитектуру, 2) отсутствует стандартный синтаксис ассемблерных вставок, и различные компиляторы используют собственный синтаксис для вставок.

2.2. Использование встроенных SIMD-функций компилятора

Многие современные компиляторы поддерживают встроенные функции (intrinsics). Реализация этих функций встроена в компилятор. Вместо их вызова компилятор вставляет тело функции, т.е. все ее команды. Время исполнения встроенных функций меньше, чем обычных функций, так как операции вызова подпрограммы и возврата из подпрограммы исключены.

Одной из групп встроенных функций являются встроенные функции SIMD-расширений (SIMD intrinsics). Они обеспечивают возможность использования команд SIMD-расширений с помощью привычного синтаксиса для вызова С-функций вместо использования ассемблерного кода и работы с регистрами процессора. В отличие от варианта с использованием ассемблерных вставок, для встроенных функций компилятор выполняет оптимизацию кода.

2.3 Использование встроенных функций векторных расширений GCC

Компилятор GCC предоставляет средства для описания векторных типов данных и встроенные функции для работы с переменными этих типов. В случае их использования при компиляции исходных текстов программ необходимо указывать ключи, включающие генерацию кода для SSE или SSE2 (-msse, -msse2).

Главный недостаток данного способа векторизации – привязка к компилятору GCC. К достоинствам относится отсутствие привязки к конкретной архитектуре. То есть, возможно написание текста, который будет без модификации исполняться на различных архитектурах, имеющих SIMDрасширения. Кроме этого, у текста программы, написанного с использованием данного способа, читаемость выше, чем, например, в варианте с ассемблерными вставками и SIMD intrinsics.

Подытожим сравнение трех способов векторизации. Использование ассемблерных вставок позволяет вручную строить наиболее эффективные программы. Вариант с использованием SIMD intrinsics позволяет достигать почти такой же эффективности полуавтоматически средствами компилятора, без необходимости использования машинных команд и работы с регистрами. Оба эти варианта привязаны к одной архитектуре процессора: x86/x86-64. Этот недостаток преодолевается с помощью встроенных функций векторных расширений GCC. Данный способ подходит для разных архитектур, имеющих SIMD-команды. Кроме этого, программы, построенные с его использованием, обладают наилучшей читаемостью среди всех рассмотренных способов. Существенный недостаток этого способа заключается в привязке к компилятору GCC.

2.4 Использование автоматической векторизации компилятором

Оптимизирующий компилятор, умеющий из обычного кода генерировать код для SIMD-расширения – это наиболее простой и эффективный путь к достижению высокой производительности. Основным недостатком этого подхода является то, что код должен удовлетворять определенным критериям, чтобы компилятор мог его векторизовать. Во многих случаях компилятор не может распознать возможность эффективного применения векторных операций.

Некоторые компиляторы (например, Intel C/C++ Compiler) поддерживают специальные директивы, с помощью которых программист может дать компилятору дополнительную информацию о коде, способствующую его векторизации. Например, директива может указывать компилятору, что итерации некоторого цикла независимы друг от друга, и их можно выполнять параллельно.

(например, -O3 -ftree-vectorize в GCC).

2.5 Использование высокопроизводительных библиотек

Для многих предметных областей существуют эффективно реализованные библиотеки операций, оптимизированные под различные вычислительные системы. Наиболее ярким примером таких библиотек можно назвать BLAS и LAPACK, которые содержат процедуры, реализующие многие операции линейной алгебры (работа с векторами, матрицами).

Существуют реализации этих библиотеки под многие архитектуры, что обеспечивает переносимость программ, написанных с их использованием. Например, реализацию BLAS содержат библиотеки Intel MKL, AMD ACML, NVIDIA CuBLAS, свободно распространяемая библиотека ATLAS. Каждая реализация стремится учесть все особенности целевой архитектуры для достижения высокой производительности. В частности, для временного хранения данных эффективно используется кэш-память, а для вычислений используются векторные операции.

Выбор оптимального варианта реализации алгоритма зависит от нескольких факторов, включая производительность, простоту разработки, поддержку и переносимость. Давайте рассмотрим каждый из трех вариантов и сравним их.

### 1. Вариант без ручной векторизации

**Описание:**  
Этот вариант предполагает написание кода на языке Си без использования SIMD-инструкций. Компилятор может попытаться автоматически векторизовать код, но это зависит от его возможностей и настроек.

**Преимущества:**

* **Простота разработки:** Код легче писать и поддерживать, так как не требуется знание SIMD-инструкций.
* **Переносимость:** Код будет работать на любой архитектуре, поддерживающей Си.

**Недостатки:**

* **Производительность:** Автоматическая векторизация компилятора может быть не такой эффективной, как ручная векторизация.
* **Неопределенность:** Невозможно точно знать, будет ли компилятор векторизовать код, и насколько эффективно.

### 2. Вариант с ручной векторизацией

**Описание:**  
Этот вариант предполагает использование SIMD-инструкций напрямую через ассемблерные вставки, встроенные функции компилятора или расширения GCC.

**Преимущества:**

* **Производительность:** Ручная векторизация позволяет достичь максимальной производительности, так как программист может точно контролировать, какие инструкции используются.
* **Контроль:** Программист имеет полный контроль над тем, как данные обрабатываются, что может быть полезно для оптимизации.

**Недостатки:**

* **Сложность разработки:** Требует глубокого знания SIMD-инструкций и архитектуры процессора.
* **Поддержка и переносимость:** Код может стать менее переносимым, так как зависит от конкретной архитектуры и компилятора.

### 3. Вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS

**Описание:**  
Этот вариант предполагает использование высокооптимизированных функций из библиотеки BLAS (Basic Linear Algebra Subprograms) для выполнения матричных операций.

**Преимущества:**

* **Производительность:** Библиотеки BLAS, такие как Intel MKL, OpenBLAS и другие, оптимизированы для конкретных архитектур и могут использовать SIMD-инструкции для достижения высокой производительности.
* **Простота разработки:** Программисту не нужно знать детали реализации, достаточно вызвать соответствующие функции.
* **Поддержка и переносимость:** Библиотеки BLAS широко поддерживаются и могут быть использованы на различных платформах.

**Недостатки:**

* **Зависимость от библиотеки:** Код зависит от наличия и правильной настройки библиотеки BLAS.
* **Ограничения:** Не все операции могут быть реализованы с использованием BLAS, особенно если алгоритм включает нестандартные операции.

### Сравнение и вывод

**Какой вариант лучше?**

* **Лучший с точки зрения производительности:** Вариант с ручной векторизацией, так как он позволяет достичь максимальной производительности, контролируя использование SIMD-инструкций.
* **Лучший с точки зрения простоты разработки и поддержки:** Вариант с использованием библиотеки BLAS, так как он позволяет использовать высокооптимизированные функции без необходимости знать детали реализации.
* **Лучший с точки зрения переносимости:** Вариант без ручной векторизации, так как он не зависит от конкретных SIMD-инструкций и может быть скомпилирован на любой архитектуре, поддерживающей Си.

**Какой вариант быстрее?**

* **Быстрее всего:** Вариант с ручной векторизацией, так как он позволяет точно контролировать использование SIMD-инструкций и может быть оптимизирован под конкретную архитектуру.
* **Быстрее, чем вариант без ручной векторизации:** Вариант с использованием библиотеки BLAS, так как функции BLAS оптимизированы для использования SIMD-инструкций и многопоточности.

**Почему?**

* **Ручная векторизация:** Позволяет точно контролировать использование SIMD-инструкций и может быть оптимизирована под конкретную архитектуру.
* **Библиотека BLAS:** Оптимизированные функции BLAS используют SIMD-инструкции и многопоточность, что позволяет достичь высокой производительности без необходимости знать детали реализации.
* **Без ручной векторизации:** Автоматическая векторизация компилятора может быть не такой эффективной, как ручная векторизация или использование оптимизированных библиотек.

### Заключение

Если ваша цель — достичь максимальной производительности и вы готовы потратить время на изучение и оптимизацию SIMD-инструкций, то лучшим вариантом будет ручная векторизация. Если вам нужна высокая производительность с минимальными усилиями по разработке и поддержке, то использование библиотеки BLAS будет оптимальным выбором. Если же вам важнее простота разработки и переносимость, то вариант без ручной векторизации будет наиболее подходящим.

**Моя лаба**

2 код SSE.

float maxRowSum(float\* matrix, int size)

* **\_\_m128\* mmMatrix = (\_\_m128\*) matrix;**: Преобразуем указатель на массив float в указатель на \_\_m128 (вектор из 4 элементов типа float).
* **\_\_m128\* block;**: Объявляем указатель на \_\_m128, который будет использоваться для итерации по блокам матрицы.
* **const \_\_m128 modulemask = \_mm\_castsi128\_ps(\_mm\_set1\_epi32(0x7FFFFFFF));**: Создаем маску для вычисления абсолютных значений элементов. Маска 0x7FFFFFFF обнуляет знаковый бит, что эквивалентно вычислению абсолютного значения.
* **block = mmMatrix;**: Устанавливаем указатель block на начало матрицы.
* **\_\_m128 mmSum = \_mm\_setzero\_ps();**: Инициализируем вектор mmSum нулями. Этот вектор будет использоваться для суммирования абсолютных значений элементов в текущей строке.
* **\_\_m128 buf;**: Объявляем вектор buf, который будет использоваться для хранения текущего блока элементов.
* **for (int j = 0; j < size / 4; j++, block++)**: Запускаем цикл по блокам строки. Каждый блок содержит 4 элемента, поэтому количество итераций равно size / 4.
  + **buf = \_mm\_load\_ps((float\*)block);**: Загружаем 4 элемента из матрицы в вектор buf.
  + **buf = \_mm\_and\_ps(modulemask, buf);**: Применяем маску modulemask к вектору buf, чтобы получить абсолютные значения элементов.
  + **mmSum = \_mm\_add\_ps(mmSum, buf);**: Складываем вектор buf с вектором mmSum.
* **float fBuf[4];**: Объявляем массив fBuf для хранения результатов суммирования.
* **\_mm\_store\_ps(fBuf, mmSum);**: Сохраняем результаты суммирования из вектора mmSum в массив fBuf.
* **tempMax = fBuf[0] + fBuf[1] + fBuf[2] + fBuf[3];**: Суммируем элементы массива fBuf, чтобы получить сумму абсолютных значений элементов в текущей строке.

Функция maxRowSum эффективно вычисляет максимальную сумму абсолютных значений элементов в строках матрицы с использованием векторизации (SSE). Это позволяет значительно ускорить вычисления за счет параллельной обработки нескольких элементов одновременно.

Transpose:

* **\_\_m128\* mMatrix = (\_\_m128\*) matrix;**: Преобразуем указатель на массив float в указатель на \_\_m128 (вектор из 4 элементов типа float).
* **\_\_m128\* block;**: Объявляем указатель на \_\_m128, который будет использоваться для итерации по блокам матрицы.
* **block = mMatrix;**: Устанавливаем указатель block на начало матрицы.
* **\_\_m128 buf = \_mm\_load\_ps((float\*)block);**: Загружаем 4 элемента из матрицы в вектор buf.
  + \_mm\_load\_ps — функция, которая загружает 4 элемента типа float в вектор \_\_m128.
* **float fBuf[4];**: Объявляем массив fBuf для хранения результатов загрузки.
* **\_mm\_store\_ps(fBuf, buf);**: Сохраняем результаты загрузки из вектора buf в массив fBuf.
* **T[(j \* 4) \* size + i] = fBuf[0];**: Сохраняем элемент fBuf[0] в транспонированной матрице T.
* **T[(j \* 4 + 1) \* size + i] = fBuf[1];**: Сохраняем элемент fBuf[1] в транспонированной матрице T.
* **T[(j \* 4 + 2) \* size + i] = fBuf[2];**: Сохраняем элемент fBuf[2] в транспонированной матрице T.
* **T[(j \* 4 + 3) \* size + i] = fBuf[3];**: Сохраняем элемент fBuf[3] в транспонированной матрице T.

Sum:

* **\_\_m128\* mmMatrixA = (\_\_m128\*) m1;**: Преобразуем указатель на массив float в указатель на \_\_m128 (вектор из 4 элементов типа float) для первой матрицы.
* **\_\_m128\* mmMatrixB = (\_\_m128\*) m2;**: Преобразуем указатель на массив float в указатель на \_\_m128 для второй матрицы.
* **\_\_m128\* mmResult = (\_\_m128\*) su;**: Преобразуем указатель на массив float в указатель на \_\_m128 для результата.
* **for (int i = 0; i < size \* size / 4; i++)**: Запускаем цикл по элементам матриц. Каждый элемент матрицы представляет собой вектор из 4 элементов, поэтому количество итераций равно size \* size / 4.
  + **mmResult[i] = \_mm\_add\_ps(mmMatrixA[i], mmMatrixB[i]);**: Складываем соответствующие векторы из матриц m1 и m2 и сохраняем результат в матрицу su.
    - mmMatrixA[i] — вектор из 4 элементов первой матрицы.
    - mmMatrixB[i] — вектор из 4 элементов второй матрицы.
    - \_mm\_add\_ps — функция, которая складывает два вектора типа \_\_m128.
    - mmResult[i] — результат сложения векторов.

Multiply:

* **\_\_m128 vecA, vecB, vecR;**: Объявляем три вектора типа \_\_m128, которые будут использоваться для хранения данных.
  + vecA — для хранения строки матрицы A.
  + vecB — для хранения столбца матрицы B.
  + vecR — для хранения промежуточных результатов умножения.
* **for (int i = 0; i < n; i++)**: Запускаем цикл по строкам матрицы result. Переменная i проходит значения от 0 до n - 1.
* **for (int j = 0; j < n; j++)**: Запускаем цикл по столбцам матрицы result. Переменная j проходит значения от 0 до n - 1.
* **vecR = \_mm\_setzero\_ps();**: Инициализируем вектор vecR нулями. Этот вектор будет использоваться для накопления результатов умножения.
* **for (int k = 0; k < n; k += 4)**: Запускаем цикл по элементам строки i матрицы A и столбца j матрицы B. Переменная k проходит значения от 0 до n - 1 с шагом 4.
  + **vecA = \_mm\_loadu\_ps(&A[i \* n + k]);**: Загружаем 4 элемента строки i матрицы A в вектор vecA.
    - \_mm\_loadu\_ps — функция, которая загружает 4 элемента типа float в вектор \_\_m128.
  + **vecB = \_mm\_loadu\_ps(&B[j \* n + k]);**: Загружаем 4 элемента столбца j матрицы B в вектор vecB.
  + **vecR = \_mm\_add\_ps(vecR, \_mm\_mul\_ps(vecA, vecB));**: Умножаем векторы vecA и vecB, складываем результат с вектором vecR и сохраняем результат в vecR.
    - \_mm\_mul\_ps — функция, которая умножает два вектора типа \_\_m128.
    - \_mm\_add\_ps — функция, которая складывает два вектора типа \_\_m128.
* **result[i \* n + j] = vecR[0] + vecR[1] + vecR[2] + vecR[3];**: Суммируем элементы вектора vecR и сохраняем результат в элемент матрицы result, находящийся в строке i и столбце j.

scalarMultiply

* **\_\_m128\* mmMatrix = (\_\_m128\*) m1;**: Преобразуем указатель на массив float в указатель на \_\_m128 (вектор из 4 элементов типа float) для исходной матрицы.
* **\_\_m128\* mmResult = (\_\_m128\*) sM;**: Преобразуем указатель на массив float в указатель на \_\_m128 для результата.
* **\_\_m128 mmScalar = \_mm\_load1\_ps(&scalar);**: Загружаем скалярное значение scalar в вектор mmScalar.
  + \_mm\_load1\_ps — функция, которая загружает одно значение типа float в каждый элемент вектора \_\_m128.
* **\_\_m128\* block;**: Объявляем указатель на \_\_m128, который будет использоваться для итерации по блокам матрицы.
* **block = mmMatrix;**: Устанавливаем указатель block на начало матрицы.
* **for (int i = 0, k = 0; i < size; i++)**: Запускаем цикл по строкам матрицы. Переменная i проходит значения от 0 до size - 1.
  + k — индекс для записи результата в матрицу sM.
* **for (int j = 0; j < size / 4; j++, block++, k++)**: Запускаем цикл по блокам строки. Каждый блок содержит 4 элемента, поэтому количество итераций равно size / 4.
  + block++ — перемещаем указатель block на следующий блок.
  + k++ — увеличиваем индекс для записи результата.
* **\_\_m128 buf = \_mm\_load\_ps((float\*)block);**: Загружаем 4 элемента из матрицы в вектор buf.
  + \_mm\_load\_ps — функция, которая загружает 4 элемента типа float в вектор \_\_m128.
* **mmResult[k] = \_mm\_mul\_ps(buf, mmScalar);**: Умножаем вектор buf на вектор mmScalar и сохраняем результат в матрицу sM.
  + \_mm\_mul\_ps — функция, которая умножает два вектора типа \_\_m128.

main  
**A = (float\*)\_mm\_malloc(N \* N \* sizeof(float), 4 \* sizeof(float));**: Выделяем память для матрицы A размером N x N с выравниванием по 16 байт.

\_mm\_malloc — функция, которая выделяет блок памяти с выравниванием по 16 байт и возвращает указатель на него.

* **srand(time(NULL));**: Инициализируем генератор случайных чисел текущим временем.
  + time(NULL) — функция, которая возвращает текущее время.
  + srand — функция, которая инициализирует генератор случайных чисел.
* **for (int i = 0; i < N \* N; i++)**: Запускаем цикл по элементам матрицы A.
  + A[i] = (float)rand() / (float)RAND\_MAX \* 10; — заполняем элемент матрицы случайным числом от 0 до 10.
    - rand() — функция, которая возвращает случайное целое число.
    - RAND\_MAX — максимальное значение, которое может вернуть rand().
    - (float)rand() / (float)RAND\_MAX — преобразуем случайное число в диапазон от 0 до 1.
    - \* 10 — умножаем на 10, чтобы получить число в диапазоне от 0 до 10.
* **I = (float\*)\_mm\_malloc(N \* N \* sizeof(float), 4 \* sizeof(float));**: Выделяем память для матрицы I с выравниванием по 16 байт.
* **res = (float\*)\_mm\_malloc(N \* N \* sizeof(float), 4 \* sizeof(float));**: Выделяем память для матрицы res с выравниванием по 16 байт.
* **temp1 = (float\*)\_mm\_malloc(N \* N \* sizeof(float), 4 \* sizeof(float));**: Выделяем память для матрицы temp1 с выравниванием по 16 байт.
* **oneM(I, N);**: Создаем единичную матрицу I.
* **oneM(res, N);**: Создаем единичную матрицу res.
* **oneM(temp1, N);**: Создаем единичную матрицу temp1.

**Транспонирование матрицы A**

transpose(B, A, N);

* **transpose(B, A, N);**: Транспонируем матрицу A и сохраняем результат в матрицу B.

**Вычисление норм матрицы A**

float a = maxRowSum(A, N);

float b = maxRowSum(B, N);

float c;

c = 1.0 / (a \* b);

* **float a = maxRowSum(A, N);**: Вычисляем максимальную сумму абсолютных значений элементов в строках матрицы A.
* **float b = maxRowSum(B, N);**: Вычисляем максимальную сумму абсолютных значений элементов в строках транспонированной матрицы B (эквивалентно вычислению максимальной суммы абсолютных значений элементов в столбцах матрицы A).
* **float c;**: Объявляем переменную c.
* **c = 1.0 / (a \* b);**: Вычисляем значение c, которое равно обратной величине произведения a и b.

**Вычисление ряда**

for (int i = 0; i < M; i++) {

multiply(temp, temp1, R, N);

memcpy(temp1, temp, sizeof(float) \* N \* N);

sum(res, res, temp, N);

}

* **for (int i = 0; i < M; i++)**: Запускаем цикл по количеству итераций M.
  + **multiply(temp, temp1, R, N);**: Умножаем матрицы temp1 и R и сохраняем результат в матрицу temp.
  + **memcpy(temp1, temp, sizeof(float) \* N \* N);**: Копируем содержимое матрицы temp в матрицу temp1.
  + **sum(res, res, temp, N);**: Складываем матрицы res и temp и сохраняем результат в матрицу res.

**Умножение результата на матрицу B**

multiply(temp, res, B, N);

* **multiply(temp, res, B, N);**: Умножаем матрицы res и B и сохраняем результат в матрицу temp.
* **printf("Time taken: %f seconds\n\r", (double)(clock() - t) / CLOCKS\_PER\_SEC);**: Выводим время выполнения программы.
  + clock() - t — количество тактов процессора, прошедших с момента начала выполнения.
  + CLOCKS\_PER\_SEC — количество тактов процессора в секунду.
  + (double)(clock() - t) / CLOCKS\_PER\_SEC — преобразуем количество тактов в секунды.

2 код AVX

Код для использования встроенных SIMD-функций компилятора для расширения AVX. AVX (Advanced Vector Extensions) позволяет работать с векторами из 8 элементов типа float (256 бит) вместо 4 элементов (128 бит) в SSE.

1. **Подключение библиотеки AVX**:

#include <immintrin.h> // Для AVX

1. **Использование \_\_m256 вместо \_\_m128**:
   * В функциях maxRowSum, maxColumnSum, transpose, sum, multiply, и scalarMultiply заменены типы данных на \_\_m256 для работы с векторами из 8 элементов типа float.
2. **Изменение выравнивания памяти**:
   * Выделение памяти с выравниванием по 32 байта для использования AVX:

A = (float\*)\_mm\_malloc(N \* N \* sizeof(float), 32);

1. **Использование AVX-функций**:
   * Заменены функции SSE на соответствующие функции AVX:
     + \_mm256\_load\_ps вместо \_mm\_load\_ps
     + \_mm256\_setzero\_ps вместо \_mm\_setzero\_ps
     + \_mm256\_add\_ps вместо \_mm\_add\_ps
     + \_mm256\_mul\_ps вместо \_mm\_mul\_ps
     + \_mm256\_store\_ps вместо \_mm\_store\_ps
     + \_mm256\_set1\_ps вместо \_mm\_set1\_ps
     + \_mm256\_and\_ps вместо \_mm\_and\_ps

**Код с BLAS**

Функция maxRowSum вычисляет максимальную сумму элементов среди всех строк квадратной матрицы. Давайте разберем её по частям:

**1. Объявление массива sums**

float sums[size];

Здесь создается массив sums размером size, где будут храниться суммы элементов каждой строки матрицы.

**2. Цикл для вычисления сумм элементов строк**

for (int i = 0; i < size; ++i) {

sums[i] = cblas\_sasum(size, matrix + size \* i, 1);

}

* for (int i = 0; i < size; ++i): Цикл проходит по всем строкам матрицы.
* sums[i] = cblas\_sasum(size, matrix + size \* i, 1);: Для каждой строки вычисляется сумма её элементов.
  + cblas\_sasum — это функция из библиотеки BLAS (Basic Linear Algebra Subprograms), которая вычисляет сумму абсолютных значений элементов массива.
  + size — количество элементов в строке (так как матрица квадратная, размер строки равен size).
  + matrix + size \* i — указатель на начало i-й строки матрицы. matrix указывает на начало матрицы, а size \* i смещает указатель на начало i-й строки.
  + 1 — шаг, с которым происходит суммирование элементов (в данном случае суммируются все элементы строки подряд).

**3. Поиск максимальной суммы**

return sums[cblas\_isamax(size, sums, 1)];

* cblas\_isamax — это функция из библиотеки BLAS, которая возвращает индекс элемента с максимальным значением в массиве.
  + size — размер массива sums.
  + sums — массив, в котором ищется максимальный элемент.
  + 1 — шаг, с которым происходит поиск максимального элемента (в данном случае поиск происходит по всем элементам массива).
* sums[cblas\_isamax(size, sums, 1)]: Возвращается значение максимального элемента в массиве sums, который соответствует максимальной сумме элементов строки.

**Итог**

Функция maxRowSum вычисляет суммы элементов каждой строки квадратной матрицы, а затем возвращает максимальную из этих сумм.

**2. Цикл для вычисления сумм элементов столбцов**

for (int i = 0; i < size; ++i) {

sums[i] = cblas\_sasum(size, matrix + i, size);

}

* for (int i = 0; i < size; ++i): Цикл проходит по всем столбцам матрицы.
* sums[i] = cblas\_sasum(size, matrix + i, size);: Для каждого столбца вычисляется сумма его элементов.
  + cblas\_sasum — это функция из библиотеки BLAS (Basic Linear Algebra Subprograms), которая вычисляет сумму абсолютных значений элементов массива.
  + size — количество элементов в столбце (так как матрица квадратная, размер столбца равен size).
  + matrix + i — указатель на начало i-го столбца матрицы. matrix указывает на начало матрицы, а i смещает указатель на начало i-го столбца.
  + size — шаг, с которым происходит суммирование элементов (в данном случае суммируются элементы столбца, которые находятся через size элементов друг от друга).

**3. Поиск максимальной суммы**

return sums[cblas\_isamax(size, sums, 1)];

* cblas\_isamax — это функция из библиотеки BLAS, которая возвращает индекс элемента с максимальным значением в массиве.
  + size — размер массива sums.
  + sums — массив, в котором ищется максимальный элемент.
  + 1 — шаг, с которым происходит поиск максимального элемента (в данном случае поиск происходит по всем элементам массива).
* sums[cblas\_isamax(size, sums, 1)]: Возвращается значение максимального элемента в массиве sums, который соответствует максимальной сумме элементов столбца.

**Итог**

Функция maxColumnSum вычисляет суммы элементов каждого столбца квадратной матрицы, а затем возвращает максимальную из этих сумм.

Функция main выполняет ряд операций с матрицами, используя функции из библиотеки BLAS (Basic Linear Algebra Subprograms) для ускорения вычислений. Давайте разберем её по частям:

**4. Вычисление матрицы R**

cblas\_sgemm(CblasRowMajor, CblasTrans, CblasNoTrans, N, N, N, -1 / a / b, A, N, A, N, 0, R, N); //R = A^T(B) \* A \* -1 / a / b

* cblas\_sgemm — функция для умножения матриц из библиотеки BLAS.
* CblasRowMajor — указывает, что матрицы хранятся в строчном порядке.
* CblasTrans — указывает, что первая матрица (A) транспонируется.
* CblasNoTrans — указывает, что вторая матрица (A) не транспонируется.
* N, N, N — размеры матриц (количество строк и столбцов).
* -1 / a / b — коэффициент, на который умножается результат умножения матриц.
* A, N, A, N — указатели на матрицы и их размеры.
* 0 — коэффициент, добавляемый к результату умножения.
* R, N — указатель на результирующую матрицу и её размер.

**5. Создание единичной матрицы E и добавление её к R**

cblas\_saxpy(N \* N, 1, E, 1, R, 1); // R = 1 \* I + R

* oneM(E, N); — создает единичную матрицу E размером N x N.
* cblas\_saxpy — функция для сложения матриц с заданным коэффициентом.
  + N \* N — количество элементов в матрицах.
  + 1 — коэффициент, на который умножается матрица E.
  + E, 1 — указатель на матрицу E и шаг (1).
  + R, 1 — указатель на матрицу R и шаг (1).

**6. Итеративные вычисления**

* oneM(tmp1, N); — создает единичную матрицу tmp1.
* memcpy(tmp, R, N \* N \* sizeof(float)); — копирует матрицу R в tmp.
* Цикл for (int i = 0; i < M; ++i) выполняет M итераций:
  + cblas\_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1, tmp1, N, R, N, 0, tmp, N); — умножает матрицу tmp1 на R и сохраняет результат в tmp.
  + memcpy(tmp1, tmp, sizeof(float) \* N \* N); — копирует матрицу tmp в tmp1.
  + cblas\_saxpy(N \* N, 1, res, 0, tmp, 0); — добавляет матрицу tmp к res (с коэффициентом 1).

**7. Окончательные вычисления и освобождение памяти**

* cblas\_sgemm(CblasRowMajor, CblasNoTrans, CblasTrans, N, N, N, 1 / a / b, res, N, A, N, 0, tmp, N); — умножает матрицу res на транспонированную матрицу A с коэффициентом 1 / a / b и сохраняет результат в tmp.
* printf("Time taken: %f seconds\n\r", (double)(clock() - t) / CLOCKS\_PER\_SEC); — выводит время выполнения программы.
* cblas\_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1, A, N, tmp, N, 0, tmp1, N); — умножает матрицу A на tmp и сохраняет результат в tmp1.

******



### ****2. Что значит суффикс****ss****?****

Суффикс ss в movss означает **"Scalar Single-precision"** (скалярное значение одинарной точности). Это указывает на то, что инструкция работает с **одним** числом с плавающей запятой одинарной точности (32 бита), а не с вектором чисел.

### ****3. Как работает****movss****?****

Когда вы используете movss, вы копируете только **нижнюю часть** (младшие 32 бита) регистра XMM. Остальные 96 бит регистра XMM остаются неизменными.