###### **МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

###### **ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

###### **НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

###### **Факультет информационных технологий**

**Кафедра параллельных вычислений**

**ОТЧЕТ** **О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ**

**Практическая работа №7**

ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ

студента 2 курса, группы 23201

Сорокина Матвея Павловича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.С. Матвеев

Новосибирск 2024

**СОДЕРЖАНИЕ**

[**ЦЕЛЬ** 3](#_Toc181300384)

[**ЗАДАНИЕ** 3](#_Toc181300385)

[**ОПИСАНИЕ РАБОТЫ** 4](#_Toc181300386)

[**ЗАКЛЮЧЕНИЕ** 8](#_Toc181300387)

[**ПРИЛОЖЕНИЯ** 9](#_Toc181300388)

[**Приложение 1:Исходный код программы *main.cpp*** 9](#_Toc181300389)

[**Приложение 2: пример результата работы программы** 10](#_Toc181300390)

# **ЦЕЛЬ**

1. Изучение SIMD-расширений архитектуры x86/x86-64.
2. Изучение способов использования SIMD-расширений в программах на языке Си.
3. Получение навыков использования SIMD-расширений.

# **ЗАДАНИЕ**

1. Написать три варианта программы, реализующей алгоритм обращения матрицы:

* вариант без ручной векторизации,
* вариант с ручной векторизацией (выбрать любой вариант из возможных трех: ассемблерная вставка, встроенные функции компилятора, расширение GCC),
* вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS.

Для элементов матриц использовать тип данных ***float***.

1. Проверить правильность работы программ на нескольких небольших тестовых наборах входных данных.
2. Каждый вариант программы оптимизировать по скорости, насколько это возможно.
3. Сравнить время работы трех вариантов программы для *N=2048, M=10*.
4. Составить отчет по лабораторной работе.

# **ОПИСАНИЕ РАБОТЫ**

Изначально нужно было разобраться с самим вычислительным алгоритмом. После этого было написана и протестирована программа, не использующая ручную векторизацию.

Далее был написан вариант программы, использующей библиотеку *BLAS*. Были изучены основные функции этой библиотеки необходимые для выполнения задачи.

После этого был написан третий вариант программы – с ручной векторизацией.

Теоритическая справка:

* Для использования инструкций *SSE (Streaming SIMD Extensions)* в *C++* используется заголовочный файл ***xmmintrin.h***, который предоставляет доступ к низкоуровневым операциям *SIMD (Single Instruction, Multiple Data)* для работы с векторными данными.

Подробнее здесь: https://www.cs.uaf.edu/2009/fall/cs301/lecture/11\_13\_sse\_intrinsics.html

* ***SIMD (Single Instruction, Multiple Data)*** - архитектурное расширение, которое позволяет выполнять одну операцию над несколькими данными одновременно.
* ***SSE*** — это одно из расширений набора инструкций x86 процессоров от Intel. SSE использует 128-битные регистры, каждый из которых может хранить например до 4 значений типа *float*. В отличие от стандартной обработки данных по одному элементу за раз, *SSE* позволяет выполнять операцию над 4 элементами одновременно.

***Отсюда и наша задача*** – ускорить выполнение первой программы без векторизации с помощью параллельных операций расширения *SSE*.

Для работы с векторными данными, содержащими несколько упакованных значений, в языках С/С++ используются следующие типы данных:

* ***\_\_m64*** - 64-бит (регистр MMX): 1 \* 64-битное целое, 2 \* 32-битных целых, 4 \* 16-битных целых, 8 \* 8-битных целых.
* ***\_\_m128*** - 128-бит (регистр XMM): 4 \* 32-битных вещественных (SSE), 2 \* 64-битных вещественных (SSE2).

В данной версии программы фактически одним значительным отличием от предыдущего варианта является вычисление скалярного произведения векторов с использованием SIMD инструкций:

|  |
| --- |
| float dot(const float\* a, const float\* b) {      // 1  const \_\_m128\* x = reinterpret\_cast<const \_\_m128\*>(a);      const \_\_m128\* y = reinterpret\_cast<const \_\_m128\*>(b);      // 2  \_\_m128 result\_vector = \_mm\_setzero\_ps();  // 3      for (unsigned int i = 0; i < N / 4; i++) {          result\_vector = \_mm\_add\_ps(result\_vector, \_mm\_mul\_ps(x[i], y[i]));      }  // 4      float result[4];      \_mm\_storeu\_ps(result, result\_vector);  // 5      return result[0] + result[1] + result[2] + result[3];  } |

Разбор функции:

1. Преобразование данных векторов в указатели \_\_m128 - тип данных, используемый в SSE для хранения 128-битных регистров, которые могут содержать 4 числа float.
2. Инициализация вектора-результата

* ***\_mm\_setzero\_ps()*** — SSE-инструкция, которая заполняет 128-битный регистр нулями. Используется для накопления суммы произведений.

Описание: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=\_mm\_setzero\_ps&ig\_expand=5866

1. Основной цикл:

* Цикл проходит по массивам x и y, которые были преобразованы в массивы 128-битных регистров. Поскольку каждый *\_\_m128* обрабатывает 4 числа *float*, количество итераций равно *N/4*.
* ***\_mm\_mul\_ps(x[i], y[i])****:* SSE-инструкция, которая умножает 4 числа из регистра x[i] на соответствующие 4 числа из y[i] (покомпонентное умножение).

Описание: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=\_mm\_mul\_ps&ig\_expand=4684

* ***\_mm\_add\_ps(result\_vector, ...)****:* SSE-инструкция, которая добавляет результат умножения к накопленному в *result\_vector*.

Описание: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=\_mm\_add\_ps&ig\_expand=137

1. Сохранение результата

* ***\_mm\_storeu\_ps(result, result\_vector)****:* SSE-инструкция, извлекает 4 числа из регистра result\_vector и сохраняет их в массив result на памяти.

Описание: https://doc.rust-lang.org/beta/core/arch/x86\_64/fn.\_mm\_storeu\_ps.html

1. Вычисление итогового результата

* Суммируются 4 элемента из массива *result* – получаем финальное значение искомого скалярного произведения.

Далее были произведены замеры времени работы программ в зависимости от входных данных.  
  
Об алгоритме: алгоритм является итерационным и аппроксимирующим, то есть получить точную обратную матрицу можно только для единичной матрицы, для остальных же матриц он будет давать приближение (аппроксимацию), и чем больше будет использоваться итераций, тем меньше будет разница от реальной обратной матрицы.

Полные листинг программы и пример ее импользования предоставлены. (см. Приложения 1,2)

Полученные результаты измерений представлены в таблице.

(*N –* размерность матриц, *M –* количество итераций в алгоритме, *T1 –* время работы варианта программы без ручной векторизации, *T2 –* время работы варианта программы с ручной векторизацией, *T3* – время работы программы с использованием матричных операций библиотеки BLAS).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **M** | **Т1, с** | **Т2, с** | **Т3, с** |
| 256 | 10 | 0.71758 | 0.125701 | 0.0543788 |
| 512 | 10 | 5.84197 | 1.01763 | 0.389335 |
| 1024 | 10 | 55.1751 | 8.86624 | 3.1173 |
| 2048 | 10 | 811.022 | 70.1284 | 26.2972 |

Из результатов следует, что вариант программы без векторизации работает значительно медленнее всех. Вариант с ручной векторизацией работает в несколько раз быстрее. Это довольно неплохой прирост, учитывая, что для векторизации были использованы инструкции расширения *SSE2*, и при этом размер регистров был 128 бит. Так как по заданию нужно было использовать тип данных *float* для элементов матрицы, то максимальный прирост производительности мог быть в 4 раза. Быстрее же всех работает вариант с использованием библиотеки *BLAS*.

# **ЗАКЛЮЧЕНИЕ**

В ходе данной лабораторной работы я пришел к следующим выводам:

1. Использование SIMD-инструкций компилятора может в разы увеличить производительность программы.
2. Библиотека BLAS - мощный инструмент при работе с линейной алгеброй, способный на порядки увеличить производительность программы.

# **ПРИЛОЖЕНИЯ**

# **Приложение 1:Исходный код программы *main.cpp***

|  |
| --- |
|  |

# **Приложение 2: пример результата работы программы**