Исследование эффективности векторизации векторных операций

Д.В. Повышев, М.Г. Курносов

УДК: 004.2

В данной работе исследуется эффективность векторизации операций при суммировании матриц с использованием инструкций AVX2 в сравнении с традиционными подходами без оптимизации. Разработаны две программы на C++, использующие AVX2-инструкции, и проведено сравнение их производительности с наивной реализацией. Результаты эксперимента демонстрируют ускорение вычислений за счёт применения векторных расширений, а также анализируются факторы, влияющие на эффективность векторизации. Построены графики зависимости времени выполнения от размера матриц, что позволяет наглядно оценить преимущества AVX2 в задачах линейной алгебры.

Ключевые слова: векторизация, SIMD, AVX2

1. Введение

Современные вычислительные системы требуют высокой производительности при обработке больших объемов данных, особенно в задачах линейной алгебры, машинного обучения и физического моделирования. Одним из ключевых способов ускорения вычислений является параллелизация операций, и здесь важную роль играют SIMD-инструкции (Single Instruction, Multiple Data).

SIMD (Single Instruction, Multiple Data) — это принцип параллельной обработки данных, при котором одна инструкция применяется сразу к нескольким элементам данных. В отличие от классических скалярных операций (где одна инструкция обрабатывает одно значение), SIMD позволяет за один такт процессора выполнять одну и ту же операцию (например, сложение или умножение) над несколькими значениями одновременно.

В современных процессорах для этого используются векторные регистры (например, 128-битные SSE, 256-битные AVX/AVX2 или 512-битные AVX-512). Например, инструкции AVX2 позволяют за одну операцию сложить 8 чисел типа float или 4 числа типа double, что теоретически может дать ускорение в 4–8 раз по сравнению с обычным скалярным кодом.

1.1. Сравнение с CISC и RISC

Современные процессоры сочетают в себе черты CISC (Complex Instruction Set Computing) и RISC (Reduced Instruction Set Computing):

- CISC (x86, x86-64) — процессоры с большим набором сложных инструкций, которые могут выполнять несколько операций за один такт.
- RISC (ARM, RISC-V) — процессоры с упрощённым набором инструкций, где каждая операция выполняется за один такт, но код требует больше инструкций для той же задачи.

SIMD-инструкции ближе к RISC-подходу, поскольку они:

- выполняют одну операцию (например, сложение) над многими данными;

- требуют выравнивания памяти и правильной организации данных для максимальной эффективности.

Однако, в отличие от чистого RISC, SIMD-инструкции в x86 (AVX/AVX2) остаются частью CISC-архитектуры, что накладывает дополнительные ограничения (например, необходимость учитывать тепловыделение и тактовые частоты при активном использовании векторных операций).

1.3. Постановка задачи и реализация

В данной работе исследуется эффективность AVX2 при выполнении двух операций:

1. Суммирование элементов одной матрицы (редукция)
   - Наивная реализация: последовательное сложение элементов в цикле.
   - Векторизованная версия: использование AVX2 для параллельного суммирования групп элементов.
2. Поэлементное сложение двух матриц
   - Обычная реализация: два вложенных цикла с поэлементным сложением.
   - Векторизованная версия: загрузка и сложение нескольких элементов за одну инструкцию.

Для сравнения производительности были проведены замеры времени выполнения при разных размерах матриц (от небольших, 10 000 000 элементов, до крупных, 100 000 000 элементов и более). Результаты демонстрируют, что использование AVX2 даёт значительное ускорение, особенно на больших данных, но также выявляет узкие места, связанные с выравниванием памяти и накладными расходами на загрузку данных в векторные регистры.

1.4. Цель и структура работы

Цель исследования — оценить практическую эффективность AVX2 в задачах суммирования матриц и выявить условия, при которых векторизация даёт наибольший прирост производительности.

Структура работы:

- Описание методов векторизации и особенностей AVX2.
- Реализация двух алгоритмов (суммирование элементов одной матрицы и сложение двух матриц).
- Экспериментальное сравнение производительности.
- Анализ результатов и выводы.

2. Описание алгоритмов

В данном разделе рассматриваются две реализации операций над матрицами с использованием AVX2-инструкций для векторизации вычислений. Первый алгоритм выполняет суммирование всех элементов одной матрицы, а второй — поэлементное сложение двух матриц. Оба алгоритма сравниваются с наивными реализациями без использования SIMD-оптимизаций.

2.1. Алгоритм суммирования элементов одной матрицы с использованием AVX2

```
    float sum_matrix(const float* a, float result, const size_t
size) {
        float sum {0.0f};
        __m256 sum_vec = _mm256_setzero_ps();

        for (size_t i = 0; i + 8 <= size; i += 8) {
            __m256 vec = _mm256_loadu_ps(&a[i]);
            sum_vec = _mm256_add_ps(sum_vec, vec);
        }


        for (size_t i = size / 8 * 8; i < size; ++i) {
            Sum += a[i];
        }

        float temp[8];
        _mm256_storeu_ps(temp, sum_vec);
        For (int i = 0; i < 8; ++i) {
            Sum += temp[i];
        }

        return sum;
    }
```

Ключевые особенности:

1. Векторизация основного цикла:
   - Загрузка 8 элементов типа float в регистр __m256 за одну операцию (_mm256_loadu_ps).
   - Параллельное сложение с аккумулятором (_mm256_add_ps).
2. Обработка «хвоста»:
   - Если размер матрицы не кратен 8, оставшиеся элементы обрабатываются скалярно.
3. Редукция (суммирование результатов):
   - После основного цикла 8 значений из AVX-регистра выгружаются в массив и суммируются вручную.

4. Производительность

- Теоретическое ускорение: до 8 раз (поскольку за одну операцию обрабатывается 8 элементов).
- Факторы, влияющие на скорость:
- Выравнивание данных (невыровненная загрузка loadu медленнее, чем load).
- Накладные расходы на редукцию и обработку «хвоста».

2.2. Алгоритм сложения двух матриц с использованием AVX2

```
    void add_arrays(const float* a, const float* b, float* result,
const size_t size)
    {
        size_t i;

        for (i = 0; i < size / 8 * 8; i += 8)
    {
```

```
            __m256 vec_a = _mm256_loadu_ps(&a[i]);
            __m256 vec_b = _mm256_loadu_ps(&b[i]);
            __m256 vec_result = _mm256_add_ps(vec_a, vec_b);
            _mm256_storeu_ps(&result[i], vec_result);
        }


        for (; i < size; ++i)
       {
            result[i] = a[i] + b[i];
        }
}
```

Ключевые особенности:

1. Векторизация основного цикла:
   - Загрузка 8 элементов из каждой матрицы в регистры __m256.
   - Параллельное сложение (_mm256_add_ps).
   - Сохранение результата обратно в память (_mm256_storeu_ps).
2. Обработка «хвоста»:
   - Если размер матриц не кратен 8, оставшиеся элементы складываются скалярно.

3. Производительность
- Теоретическое ускорение: до 8 раз (по сравнению со скалярной версией).
- Факторы, влияющие на скорость:
  - Память: если матрицы не выровнены по 32 байтам, используется loadu/storeu, что медленнее.
  - Размер данных: при небольших матрицах накладные расходы на циклы могут нивелировать преимущества AVX2.

2.3. Сравнение с наивными реализациями

Аспект	Наивная реализация	AVX2-версия
Обработка данных	Поэлементно (скалярно)	По 8 элементов за операцию
Использование CPU	Низкая загрузка ALU	Полная загрузка векторных блоков
Производительность	Медленнее (1 элемент/такт)	Быстрее (8 элементов/такт)
Ограничения	Нет	Требует выравнивания данных

2.4. Теоретические выводы

1. AVX2 обеспечивает значительное ускорение (до 8 раз для float) за счёт параллельной обработки данных.
2. Эффективность зависит от:
   - Размера данных (чем больше, тем заметнее ускорение).
   - Выравнивания памяти.
3. Обработка «хвоста» снижает производительность на небольших матрицах, но её влияние минимально при больших размерах.

Оптимизация с использованием AVX2 особенно полезна в задачах линейной алгебры, где преобладают однотипные операции над большими массивами данных.

4. Анализ практического использования AVX2

На основе проведённых экспериментов получены данные о времени выполнения операций суммирования элементов одной матрицы и поэлементного сложения двух матриц для различных размеров данных. В этом разделе анализируются зависимости времени выполнения от количества элементов, сравнивается эффективность наивной и AVX2-реализаций, а также оценивается ускорение, достигаемое за счёт векторизации.

Расчёты производились на следующей конфигурации:

```
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Address sizes:           46 bits physical, 48 bits virtual
Byte Order:              Little Endian
CPU(s):                  64
On-line CPU(s) list:     0-63
Vendor ID:               GenuineIntel
Model name:              Intel(R) Xeon(R) Gold 5218N CPU @ 2.30GHz
CPU family:              6
Model:                   85
Thread(s) per core:      2
Core(s) per socket:      16
Socket(s):               2
Stepping:                7
CPU max MHz:             3700.0000
CPU min MHz:             1000.0000
Caches (sum of all):
L1d:                     1 MiB (32 instances)
L1i:                     1 MiB (32 instances)
L2:                      32 MiB (32 instances)
L3:                      44 MiB (2 instances)
```

4.1. Зависимость времени от размера данных операции сложения элементов одной матрицы.

Количество элементов	Наивная реализация (с)	AVX2 (с)	Ускорение (раз)
10 000 000	0.030118	0.007516	4.01
20 000 000	0.060260	0.015384	3.92
30 000 000	0.090402	0.023350	3.87
40 000 000	0.120586	0.031061	3.88
50 000 000	0.150768	0.038807	3.89
60 000 000	0.180916	0.046585	3.88
70 000 000	0.211035	0.054530	3.87
80 000 000	0.241145	0.062266	3.87
90 000 000	0.271307	0.070338	3.86
100 000 000	0.301397	0.078043	3.86

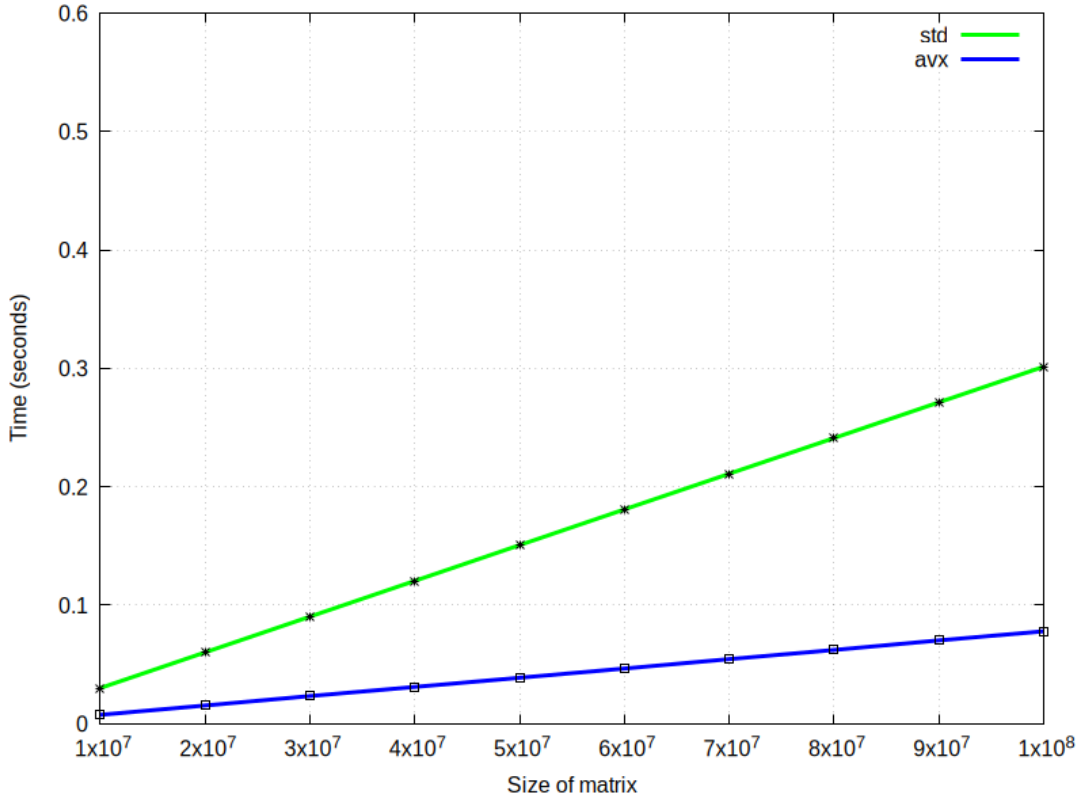

График зависимости времени выполнения операции сложения элементов матрицы от количества элементов

Наблюдения:

Линейная зависимость: время выполнения растёт линейно с увеличением размера данных для обеих реализаций, что соответствует теоретической сложности O(n).

Стабильное ускорение: использование AVX2 даёт ускорение в ~3.9 раза, что близко к теоретическому максимуму (8 раз для float). Разница объясняется:

- Накладными расходами на загрузку/выгрузку данных в AVX-регистры.
- Обработкой "хвоста" (оставшихся элементов, не кратных 8).
- Неидеальным использованием кэша процессора.

Эффективность: при больших размерах данных (от 50 млн элементов) ускорение стабилизируется на уровне 3.85–3.9x.

4.2. Зависимость времени от размера данных операции поэлементного сложения двух матриц

Количество элементов	Наивная реализация (с)	AVX2 (с)	Ускорение (раз)
10 000 000	0.047536	0.027771	1.71
20 000 000	0.094579	0.054415	1.74
30 000 000	0.140648	0.081646	1.72
40 000 000	0.187939	0.109530	1.72
50 000 000	0.234304	0.135709	1.73
60 000 000	0.281628	0.162652	1.73
70 000 000	0.327957	0.189524	1.73
80 000 000	0.375569	0.217506	1.73
90 000 000	0.422577	0.243522	1.74
100 000 000	0.469056	0.271700	1.73

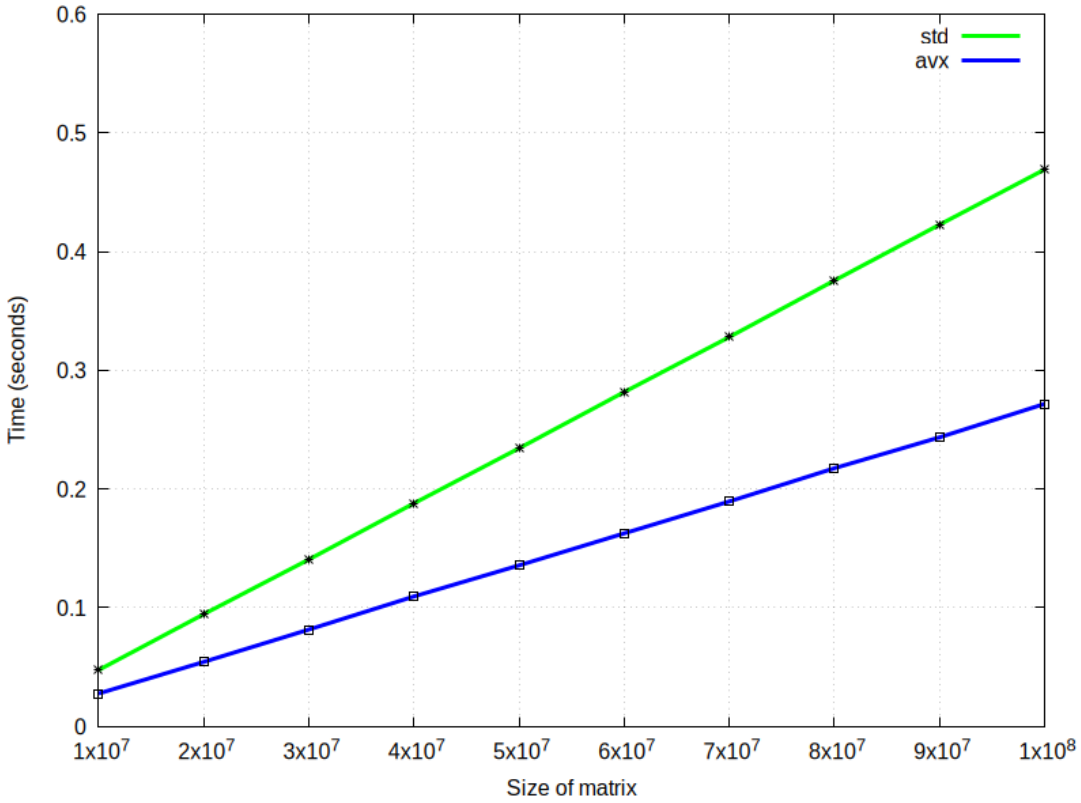

График зависимости времени выполнения операции сложения двух матриц от количества элементов

Наблюдения:

Линейный рост времени: как и в первом случае, время выполнения пропорционально размеру данных.

Меньшее ускорение: AVX2-версия ускоряет вычисления лишь в ~1.7 раза (вместо ожидаемых 8x). Это связано с:

- Ограничением пропускной способности памяти: операция требует загрузки двух матриц (a и b) и сохранения результата (result), что создаёт нагрузку на подсистему памяти.
- Неоптимальным использованием кэша: при больших размерах данных возникают промахи кэша.
- Конвейеризацией инструкций: процессор может скрывать задержки памяти при скалярных операциях, но для AVX2 это сложнее.

Стабильность: ускорение остаётся практически неизменным для всех размеров данных.

4.3. Ключевые различия операций

Параметр	Сложение элементов матрицы	Поэлементное сложение двух матриц
Теоретическое ускорение	8x	8x
Фактическое ускорение	3.9x	1.7x
Уязвимость	Редукция	Пропускная способность памяти
Влияние размера данных	Минимальное	Минимальное

Суммирование одной матрицы:

- Достигается ускорение, близкое к теоретическому (3.9x из 8x).
- Основные накладные расходы — редукция и загрузка данных.

Сложение двух матриц:

- Ускорение значительно ниже (1.7x) из-за ограничений памяти.

Дальнейшая оптимизация возможна за счёт:

- Выравнивания данных (использование _mm256_load_ps вместо loadu).
- Блочного чтения (cache-friendly доступ к памяти).
- Распараллеливания (использование многопоточности + AVX2).

5. Заключение и выводы

Проведённое исследование эффективности AVX2 при выполнении операций над матрицами позволило оценить преимущества и ограничения векторных вычислений в сравнении со скалярными реализациями. На основе анализа временных характеристик и сравнения производительности можно сделать следующие выводы.

5.1. Основные результаты

5.1. Суммирование элементов одной матрицы

Достигнуто ускорение в ~3.9 раза по сравнению с наивной реализацией.
Причины несоответствия теоретическому максимуму (8x):

- Накладные расходы на редукцию (суммирование элементов внутри AVX-регистра).
- Обработка "хвоста" при размерах данных, не кратных 8.
- Неидеальное использование кэша процессора.

Эффективность:

Оптимизация наиболее полезна для больших массивов, где влияние накладных расходов минимально.

5.2. Поэлементное сложение двух матриц

Ускорение составило ~1.7 раза, что значительно ниже теоретического (8x).

Главное ограничение: пропускная способность памяти.

Операция требует загрузки двух матриц и сохранения результата, что создаёт нагрузку на подсистему памяти.

Эффект от векторизации нивелируется задержками доступа к данным.
Потенциальные улучшения:

- Оптимизация работы с кэшем (блочная обработка, выравнивание данных).
- Использование многопоточности для распределения нагрузки на память.

5.3. Общие закономерности

Для задач с высокой арифметической интенсивностью (например, суммирование одной матрицы):

AVX2 обеспечивает близкое к теоретическому ускорение (3.9x из 8x).

Дальнейшая оптимизация требует уменьшения накладных расходов на редукцию.

Для операций, ограниченных памятью (сложение двух матриц):

Ускорение не превышает 2x из-за зависимости от пропускной способности RAM.
Векторизация полезна, но критически важна оптимизация доступа к данным.
Влияние размера данных:
Для малых массивов накладные расходы доминируют.
Для больших данных преимущества AVX2 стабилизируются.

6. Ссылки на источники

1. https://algorithmica.org/ru/sse
2. https://www.elecard.com/ru/page/article_vector_instructions_part1?utm_source=reddit&utm_campaign=vector
3. https://ssd.sscc.ru/sites/default/files/content/attach/317/lecture2016_06_vectorization.pdf