Отчет о выполнении лабораторной работы 3565.32.50494452
Оценка эффективности SIMD-оптимизаций программы Mandelbrat2

Швабра Владимир Сергеевич, Б01-411

March 30, 2025

1 Аннотация

1.1 Цели работы

1. Научиться применять различные низкоуровневые оптимизации на примере расчёта множества Мандельброта.

2. Научиться оценивать эффективность работы программы

1.2 Теоретическая справка

Сначала рассмотрим базовую последовательную модель выполнения инструкций (SEQ), а затем перейдем к современным конвейерным реализациям (PIPE) с предсказанием переходов.

1.2.1 Последовательная модель выполнения (SEQ)

На рисунке 1 представлена последовательная модель обработки инструкций:

Этапы последовательной обработки

1. **Выгрузка (F)**:
   - Извлекает инструкцию из памяти по адресу, указанному в Program Counter (PC)
   - Используется *кэш инструкций* для ускорения доступа
   - PC увеличивается на размер инструкции (или изменяется для инструкций перехода)
2. **Декодирование (D)**:
   - Определяется *тип операции* (арифметическая, логическая, перехода и т.д.)
   - Выявляются *операнды* (регистры, непосредственные значения, адреса памяти)
   - Для операций с памятью вычисляются эффективные адреса
   - Подготавливаются сигналы управления для следующих этапов
3. **Выполнение (E)**:
   - Выполнение арифметических или логических операций в *АЛУ* (Арифметико-Логическом Устройстве)
   - Для условных переходов - проверка условий и вычисление адреса перехода
4. **Обращение к памяти (M)**:

   Этот этап присутствует только для инструкций работы с памятью.
   - Для загрузки: чтение данных из памяти по вычисленному адресу
   - Для записи: запись данных в память по вычисленному адресу
   - Используется *кэш данных* для ускорения доступа
   - Обработка кэш-промахов и обращение к основной памяти при необходимости
5. **Обратная запись (W)**:
   - Для арифметических операций - запись в регистр-приемник
   - Для операций загрузки - запись данных из памяти в регистр

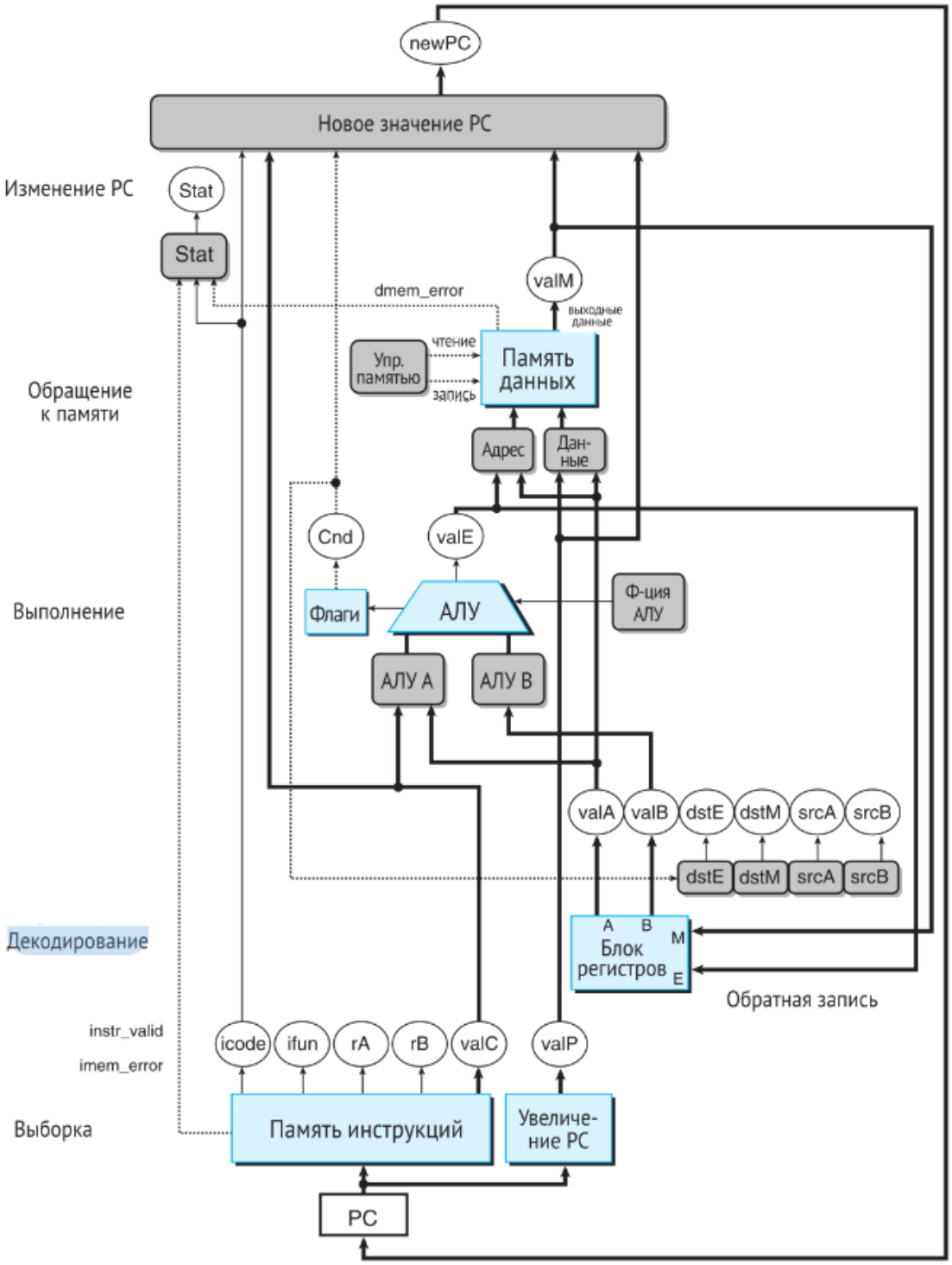

Рис. 1: Блок-схема последовательной обработки инструкций (SEQ)

6. **Изменение PC**:

- Для обычных инструкций: PC уже был увеличен на этапе выгрузки
- Для инструкций перехода: PC изменяется на новое значение
- Для условных переходов: PC изменяется только если условие истинно
- Обработка исключений и прерываний также может изменить PC

Ограничения SEQ Главный недостаток последовательной модели — низкая эффективность. Каждый этап задействует только часть аппаратных ресурсов процессора, остальные простаивают.

1.3 Конвейерная модель выполнения (PIPE)

Современные процессоры используют конвейерную обработку, как показано на рисунке 2.

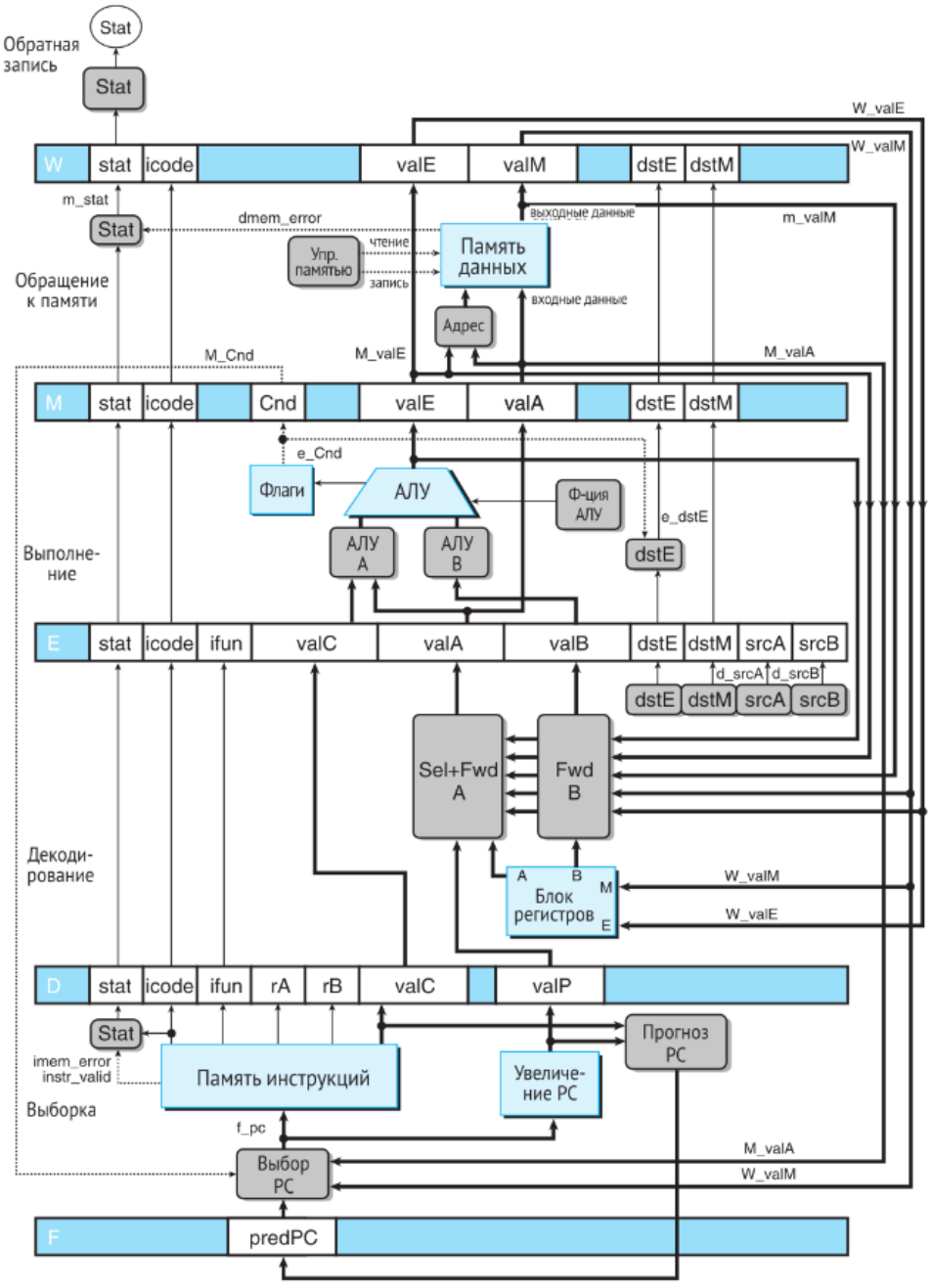

Рис. 2: Блок-схема конвейерной обработки инструкций (PIPE)

Из общих изменений в схеме добавились конвейерные регистры между каждой соседней парой этапов обработки инструкций. Это сделано для возможности выполнения разных этапов разных инструкций параллельно, как показно на рисунке 3.

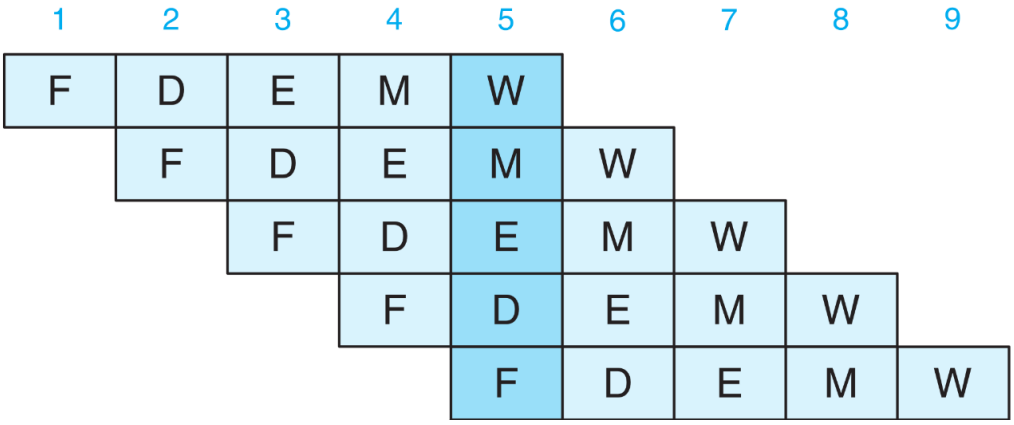

Рис. 3: Визуализация конвейерной обработки инструкций

1.3.1 Предсказание PC

Если обрабатываемая инструкция не является условным переходом, то мы можем однозначно определить следующее значение PC, но иначе нет.

- **Инструкции не выполняющие "прыжок" -** так как мы знаем размер каждой инструкции, то можем прибавить его к PC.
- **CALL, JMP, RET -** мы знаем адрес, на который нужно перейти *в любом* случае.
- **Условный переход -** До проверки условия мы не знаем куда надо делать "прыжок".

Проблему с условными переходами можно решить приостановкой загрузки новых инструкций в конвейер до момента вычисления условия, но есть решение лучше. Мы попробуем угадать произойдёт переход или нет и продолжать подгружать операции с соответствующего места, а в случае ошибочного предсказания сбросить загруженные инструкции, что, конечно, ударит по производительности.

Существует много различных стратегий для предсказания переходов, но мы не будем в это углубляться. Отметим лишь, что для облегчения работы предсказателю, не стоит писать сложных условий без необходимости. Также нужно знать об операциях условного перемещения, которые могут улучшить производительность ветвлений.

1.3.2 Риски по данным

Но схема на рисунке 3 будет работать корректно только если во всех инструкциях, которые выполняются параллельно не будет *зависимости по данным*, что может часто не соблюдаться. Приведём методы разрешения таких зависимостей.

- **NOP:** самое лёгкое решение. На этапе генерации ассемблера проставить пустые инструкции в количестве необходимом, для такого чтобы устранить параллельное выполнение инструкций зависимых по данным (Рис. 4).
- **"Пузырёк" (Привет дед!):** Аппаратная приостановка конкретных этапов обработки инструкций, зависящих по данным (Рис. 5).
- **Продвижение:** Мы добавляем в нашу конструкцию процессора (Рис. 2) возможность передачи значений (регистров, либо памяти) в предыдущие этапы обработки инструкций, чтобы ожидающая инструкция могла использовать уже вычисленное значение до окончания обработки ожидаемой инструкции.

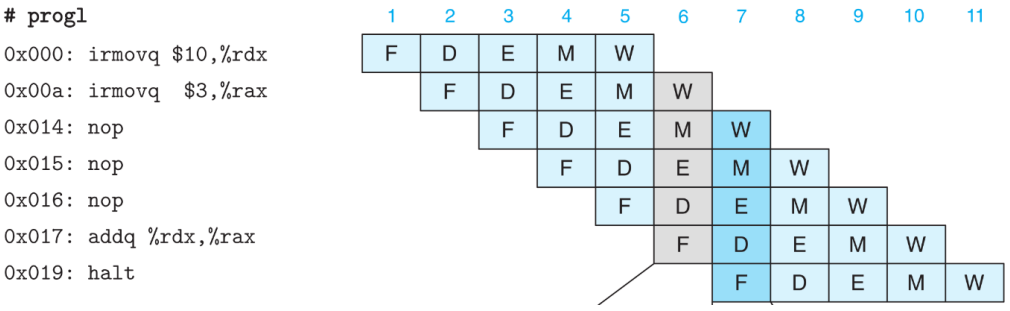

Рис. 4: Останова, реализованная через NOP для решения проблемы риска по данным

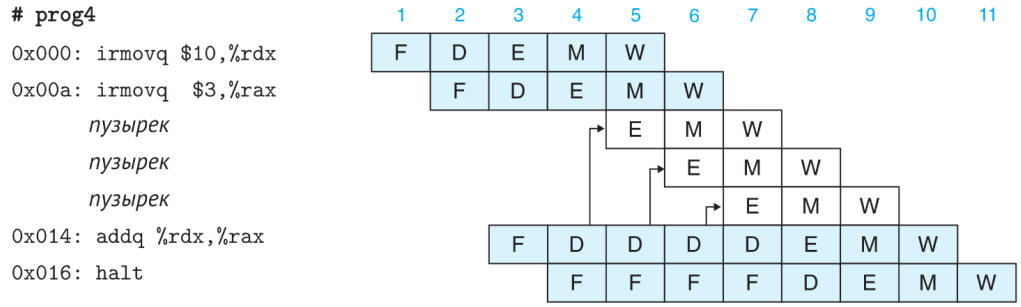

Рис. 5: Останова, реализованная через "пузырёк" для решения проблемы риска по данным

1.3.3 Взаимосвязь функциональных блоков

В настоящих процессорах, очевидно, есть несколько функциональных блоков, каждый из которых занимается обработкой операций того или иного типа. Это позволяет выполнять некоторые операции не по порядку. Такая архитектура процессора называется суперскалярной (Рис 6) и хорошо дополняет конвейерное устройство функциональных блоков.

Распределением инструкций по функциональным блокам занимается *блок управления инструкциями (БУИ (это я ввёл сам, так не называют, просто чтобы меньше писать (лисп сосать (дед уприс (а стоп, это же не гайд на летку (ну ладно, пока (швабра)))))))*. В частности на всех современных процессорах есть блок для обработки ветвлений, который пытается предсказать условные переходы (аналогично предсказателю PC в рамках одного конвейера, но на уровень выше). Такая организация обработки инструкций называется *спекулятивным выполнением*.

1.4 Оптимизации

На основе знаний изложенных выше мы можем "класть" наш код на аппаратуру применением низкоуровневых оптимизаций, чтобы увеличить производительность. Некоторые из них мы сейчас опишем

1.4.1 Вынос кода

Если в цикле вычисляется значение переменной и не меняется на всех итерациях цикла, то для данного значения стоит завести константу вне цикла и использовать её. Аналогично можно поступить при постоянном обращении к памяти, правда это будет эффективно в случае, если есть свободные регистры для сохранения переменной.

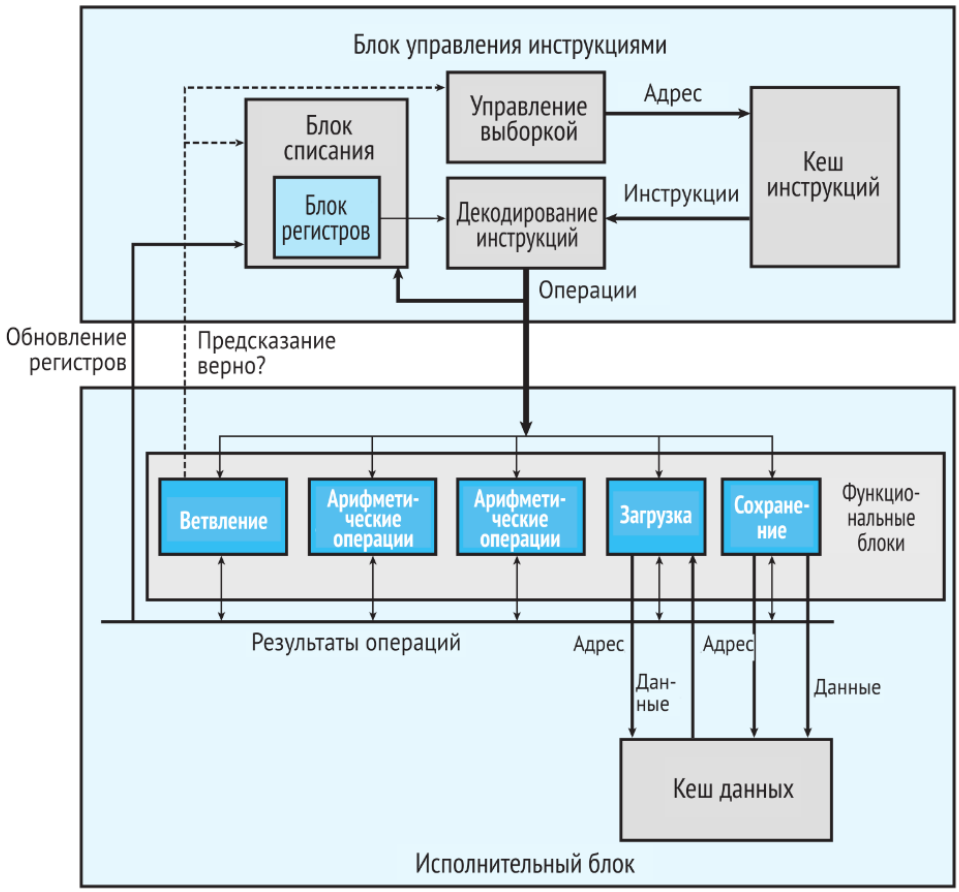

Рис. 6: Блок-схема суперскалярного процессора

Пример

```
// До оптимизации
for (size_t i = 0; i < n; ++i) {
    double result = sin(x) * arr[y] + i;  // sin(x) и arr[y] не зависят от i
    printf("%f\n", result);
}

// После оптимизации
double sin_x = sin(x);
double arr_y = arr[y];
for (size_t i = 0; i < n; ++i) {
    double result = sin_x * arr_y + i;
    printf("%f\n", result);
}
```

1.4.2 Уменьшение рисков по данным

Стоит уметь представлять в голове как конвейерная и суперскалярная архитектура будут обрабатывать инструкции и минимизировать простои процессора. Допустим, если есть возможность переставить вычисление независимых операций, то их можно поставить между зависимыми, чтобы занять неиспользуемое время.

Пример

```
// До оптимизации
float x = y * z;  // Независимая операция
float a = b * c;
float d = a + e;  // Зависит от a

// После оптимизации
float a = b * c;
float x = y * z;   // Независимая операция
float d = a + e;
```

1.4.3 Комбинация развёртывания цикла и аккумулятора

Развёртывание: это преобразование программы, которое уменьшает количество итераций за счет увеличения количества вычислений в каждой итерации. Развертывание цикла может улучшить производительность по двум причинам. Во-первых, уменьшается количество операций, не влияющих непосредственно на результат программы, таких как вычисление индекса и условное ветвление. Во-вторых, открываются возможности для дальнейшего преобразования кода при помощи аккумуляторов.

Аккумуляторы: Для ассоциативной и коммутативной комбинирующей операции (например для целочисленного сложения или умножения) производительность можно повысить, разбив набор комбинирующих операций на две или более частей, и комбинировать их результаты в конце.

Без учёта накладных расходов использование описанных подходов можно увеличить скорость выполнения цикла во столько раз, на сколько порядков проведено развёртывание. Конечно, это можно делать не бесконечно, ведь в какой-то момент регистров для аккумуляторов не будет хватать и они будут сохраняться в переменные, что только замедлит обработку инструкций. Подробную оценку производительности читайте в БХ (байт для будущих пекусов).

Пример

```
// До оптимизации
float sum = 0;
for (int i = 0; i < n; i++) {
    sum += array[i];
}

// После оптимизации (развёртывание 4x + аккумуляторы)
float sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
for (int i = 0; i < n; i += 4) {
    sum0 += array[i];
    sum1 += array[i+1];
    sum2 += array[i+2];
    sum3 += array[i+3];
}
float sum = sum0 + sum1 + sum2 + sum3;
```

1.4.4 Применение векторных инструкций (SIMD)

Модель выполнения инструкций SIMD позволяет производить одну операцию сразу с несколькими данными в соответствующих функциональных блоках с поддержкой векторных операций. Использование таких инструкций очень хорошо сочетается с вышеперечисленными оптимизациями.

Пример

```
#include <immintrin.h>

// До оптимизации
for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
}

// После оптимизации
for (int i = 0; i < n; i += 4) {
    __m128 va = _mm_load_ps(&a[i]);
    __m128 vb = _mm_load_ps(&b[i]);
    __m128 vc = _mm_add_ps(va, vb);
    _mm_store_ps(&c[i], vc);
}
```