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

Швабра Владимир Сергеевич, Б01-411

April 8, 2025

Contents

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 5-ступенчатую модель выполнения инструкций (SEQ, Sequential Execution Model), а затем перейдем к современным конвейерным реализациям (PIPE, Pipelined Execution) с предсказанием переходов. Далее будет описано 6 этапов а не 5. Это решение было принято, чтобы акцентировать внимание на выборе нового PC (Program Counter, счётчик команд).

1.2.1 Последовательная модель выполнения (SEQ)

На рисунке 1 представлена последовательная модель обработки инструкций:

Этапы последовательной обработки

1. **Выгрузка (F, Fetch)**:

   - 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, Decode)**:

   - 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, Execute)**:

   - Выполнение арифметических или логических операций в *АЛУ* (Арифметико-Логическом Устройстве, ALU - Arithmetic Logic Unit)
   - Для условных переходов - проверка условий и вычисление адреса перехода

4. **Обращение к памяти (M, Memory)**:

   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, Writeback)**:

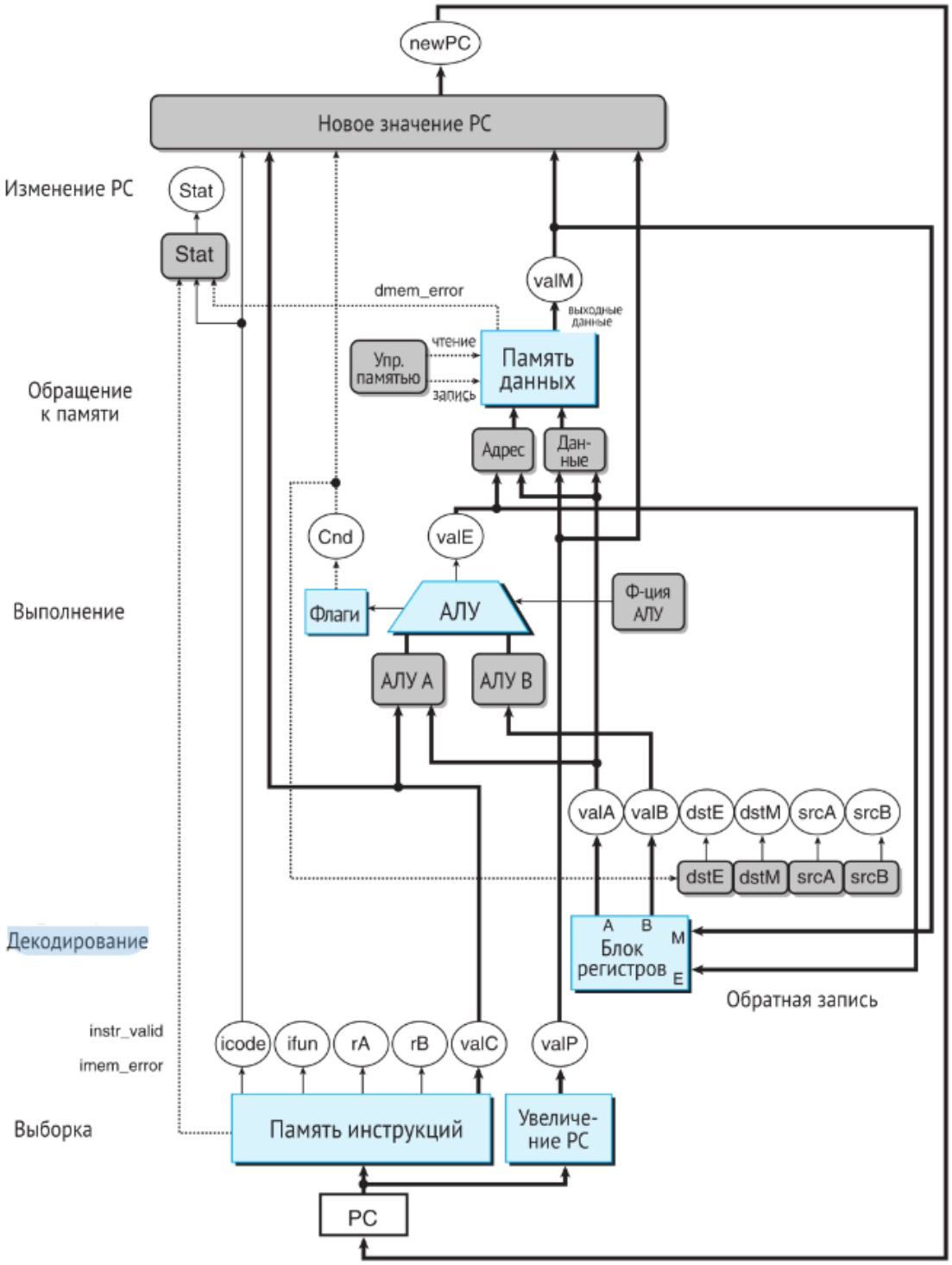


Рис. 1: Блок-схема последовательной обработки инструкций (SEQ)[2]

- Для арифметических операций - запись в регистр-приемник
- Для операций загрузки - запись данных из памяти в регистр

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.

1.2.2 Конвейерная модель выполнения (PIPE)

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

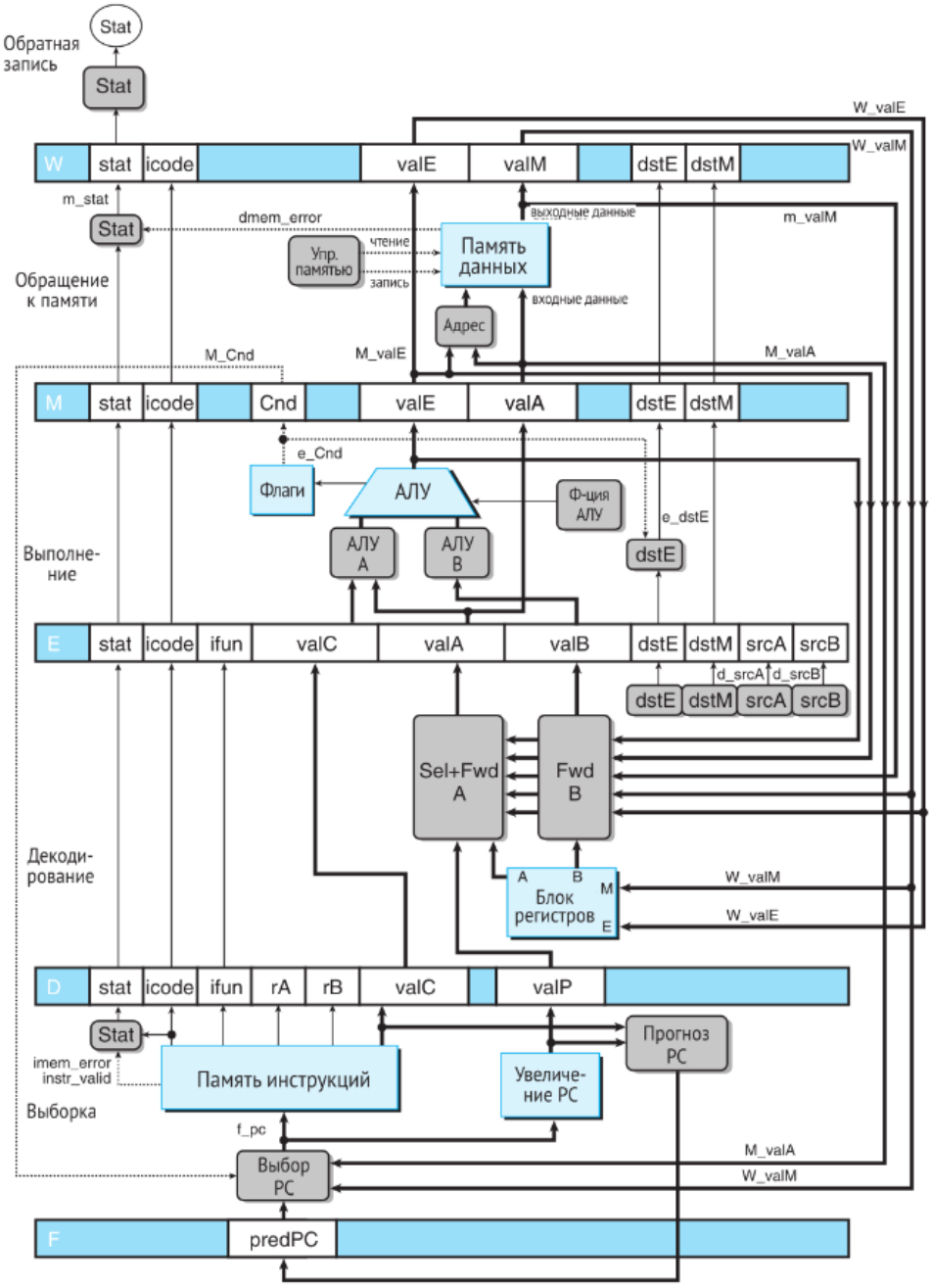


Рис. 2: Блок-схема конвейерной обработки инструкций (PIPE)[2]

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.

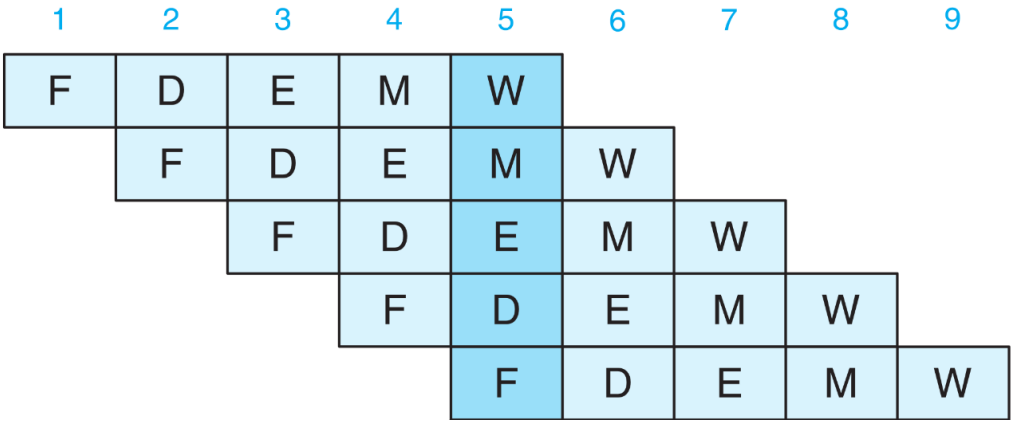


Рис. 3: Визуализация конвейерной обработки инструкций[2]

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 -** мы знаем адрес, на который нужно перейти в случаях, когда операндами являются литералы. Иначе мы не знаем куда прыгать.
- **Условный переход -** До проверки условия мы не знаем куда надо делать "прыжок".
- **RET -** также не знаем куда куда нужно возвращаться, так как адрес возврата в стеке можно менять.

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 (branch prediction), 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 (conditional move), которые могут улучшить производительност ветвлений.

Риски по данным (data hazards).

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 (No Operation):** 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" (bubble, Привет дед!):** Аппаратная приостановка конкретных этапов обработки инструкций, зависящих по данным (Рис. 5).

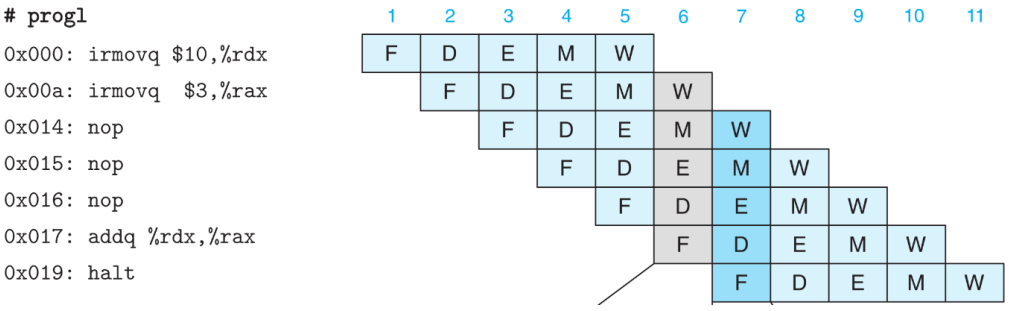


Рис. 4: Останова, реализованная через NOP для решения проблемы риска по данным[2]

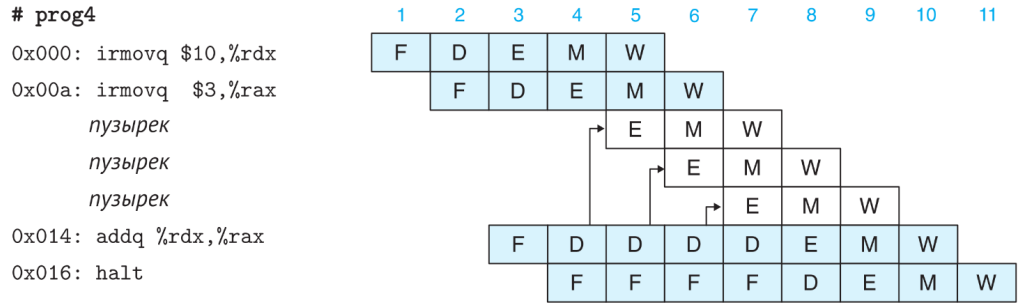


Рис. 5: Останова, реализованная через "пузырёк" для решения проблемы риска по данным[2]

- **Продвижение (forwarding):** 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. 6).

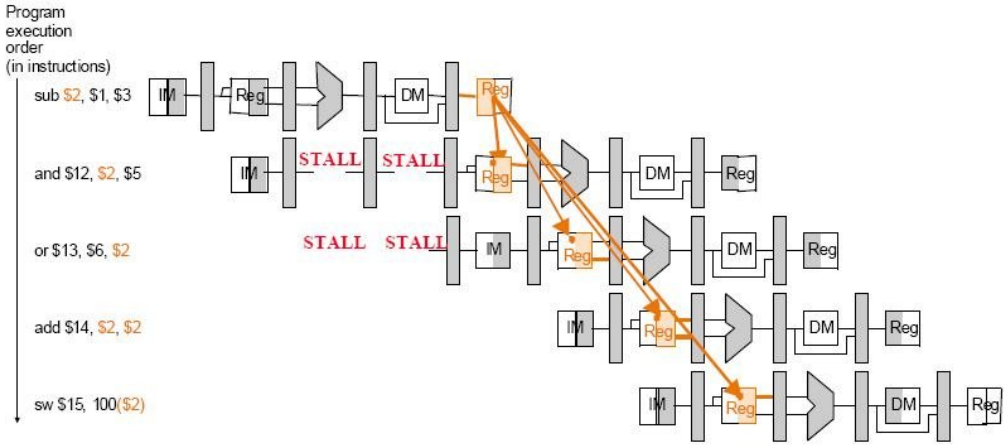


Рис. 6: Визуализация продвижения назад при зависимости по данным[3]

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 (superscalar, Рис 7) 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 упрлс (а стоп, это же не гайд на летку (ну ладно, пока (швабра))))))* (Instruction Control Unit). 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 (speculative execution)*.

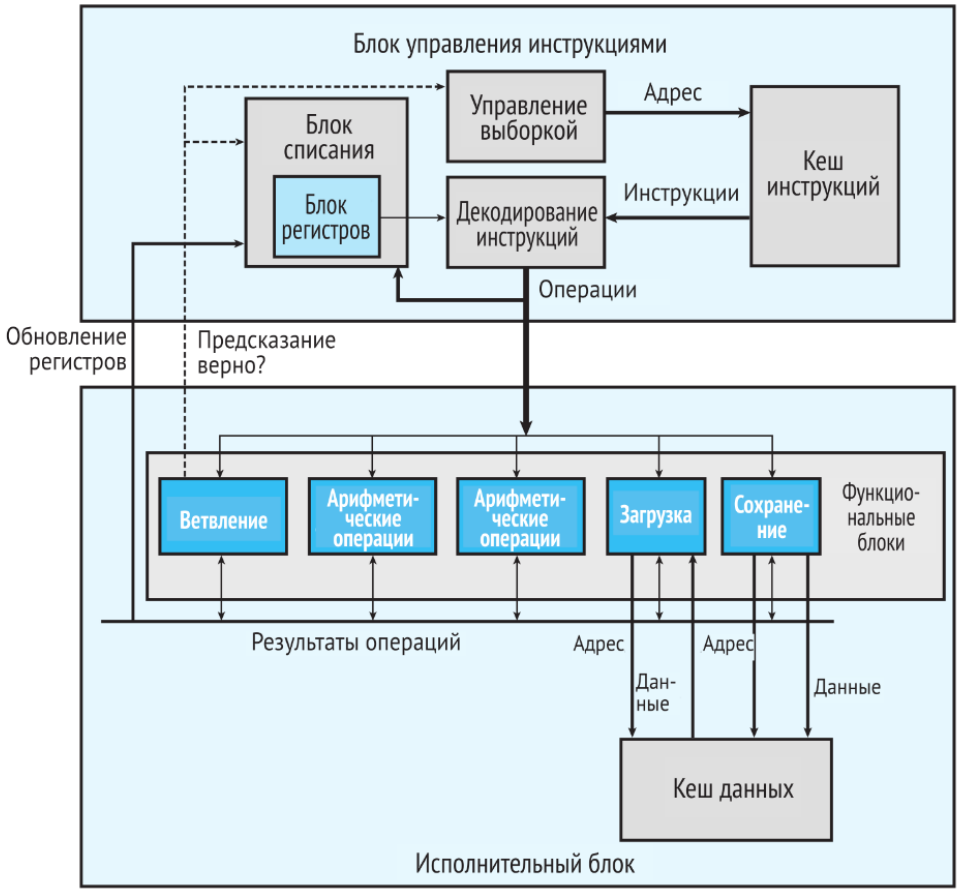


Рис. 7: Блок-схема суперскалярного процессора[2]

1.2.3 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

Вынос кода (code motion). 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.

```
// 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);
}
```

Уменьшение рисков по данным (data hazards reduction). 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.

```
// 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;
```

Комбинация развёртывания цикла (loop unrolling) и аккумулятора (accumulator).

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).

```
// 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;
```

Применение векторных инструкций (SIMD, Single Instruction Multiple Data). 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.

```
#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);
}
```

1.2.4 Измерение эффективности.

Наиболее точным способом измерения времени работы программы является ассемблерная инструкция *rdtsc (Read Time-Stamp Counter)*, которая возвращает количество пройденных тактов с момента запуска программы. Проблема состоит в том, что тактовая частота при нормальной работе машины постоянно меняется. Мы можем зафиксировать её специальными утилитами (далее для ОС Fedora).

```
# Установка нужных инструментов
$ sudo dnf install kernel-tools cpupowerutils

# Вывод информации о процессоре
$ cpupower frequency-info
analyzing CPU 0:
#...
hardware limits: 400 MHz - 3.60 GHz
#...
current CPU frequency: 2.23 GHz (asserted by call to kernel)
#...
# Несколько раз потыкав внешнюю программу можно понять какую максимальную
↪  частоту можно поставить, чтобы она не менялась. Я выбрал 1 GHz

$ sudo cpupower frequency-set -u 1.0GHz

# Проверям
$ cpupower frequency-info
```

После работы возвращаем всё на свои места, ставя ограничение на максимальную частоту больше, чем может быть максимум

```
$ sudo cpupower frequency-set -u 3.6GHz
```

Стоит уточнить, что измерение будет производиться на одном и том же "месте" множества мандельброта и при одинаковых параметрах экрана, чтобы расчёты были идентичными.

2 Оборудование

- 2 банки энергетика
- конспекты деда
- 18+ бутылка пива
- чипсы соседа
- третье издание БХ [2]
- слинки (всем с СДВГ советую)

- ноутбук (опционально) (Рис. 8)

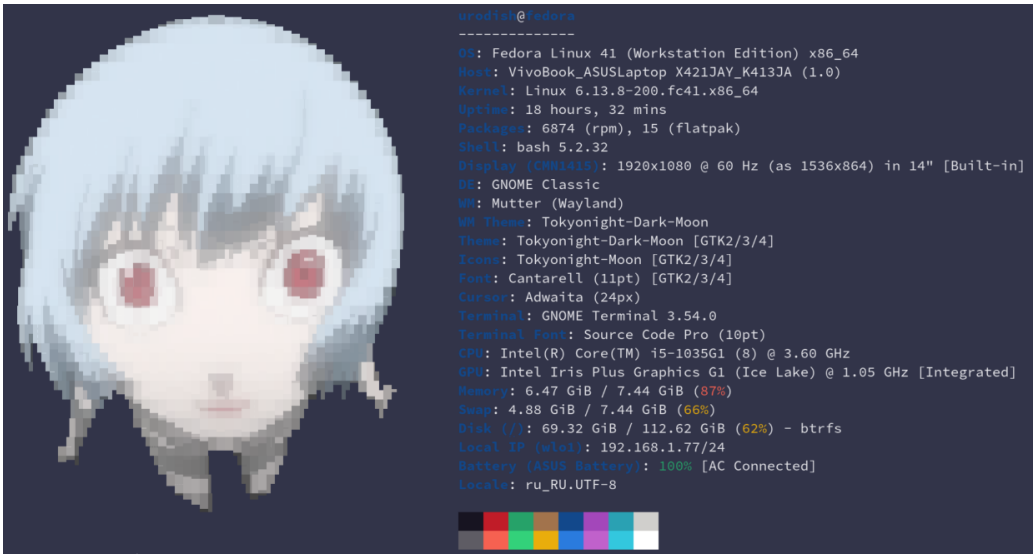


Рис. 8: Характеристики ноутбука, на котором выполняется работа

3 Ход работы

3.1 Зависимости

Для сборки проекта нужны Make и GCC (GNU Compiler Collection). Для отрисовки множества Мандельброта возьмём библиотеку SDL2 (Simple DirectMedia Layer 2) (SFML для лохов), также дополнительно установим модуль TTF (TrueType Font) для вывода FPS (Frames Per Second) на экран.

3.2 Начало работы. Поверхностная оценка производительности

Основным местом, которое мы будем оптимизировать является функция расчёта самого множества. Накладные расходы по типу инициализации SDL и логгера, обработки флагов и отрисовки множества мы не будем учитывать при измерении эффективности, так как это не цель исследования. Соответственно, максимально постараемся минимизировать их вклад в измеряемую производительность.

3.2.1 Нулевая версия

Напишем наивную версию программы и настроим на её основе:

- Обработку флагов, чтобы можно было изменять параметры отображения множества Мандельброта, отключать графику и увеличивать количество повторных обработок 1 кадра (для увеличения точности измерений)
- Движение по множеству, а также приближение или отдаление к центру координат, чтобы было удобнее рассматривать математические красоты
- Вывод текущего FPS, чтобы примерно оценить скорость расчёта кадров без точных времязатраных измерений, реализованный через функции SDL2.

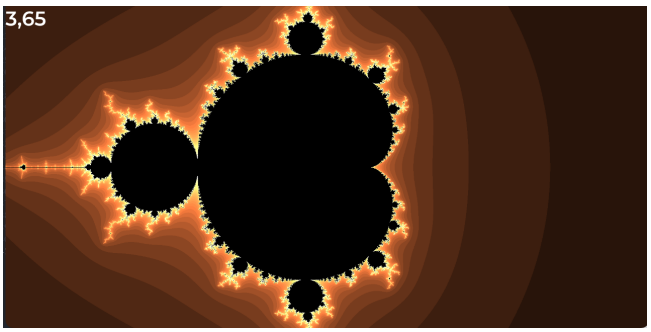


Рис. 9: Визуализация множества Мандельброта без оптимизаций

В среднем значение FPS варьируется в районе 3.5 FPS на изначальной позиции, то есть без сдвига системы координаты и без приближения. В дальнейшем мы будем использовать эту позицию для всех измерений, чтобы они проводились в одинаковых условиях.

3.2.2 Базовые оптимизации

Теперь мы можем начать оптимизировать программу. Для начала применим оптимизации, которые не требуют переписывания кода. А именно установим уровень оптимизации -O3, пропишем флаги, которые позволят компилятору применять дополнительные оптимизации, а также сделаем предварительный запуск программы, чтобы применить оптимизацию профилировщиком (Profile-guided optimization, PGO), далее будем называть PGO.

При первом запуске профилировщик будет "анализировать" как выполняется программа и предполагать какие решения могут её ускорить. К примеру это может сильно ускорить ветвления, так как будет заранее известно когда выполняется условие. Эта информация записывается в файл, который используется при втором запуске.

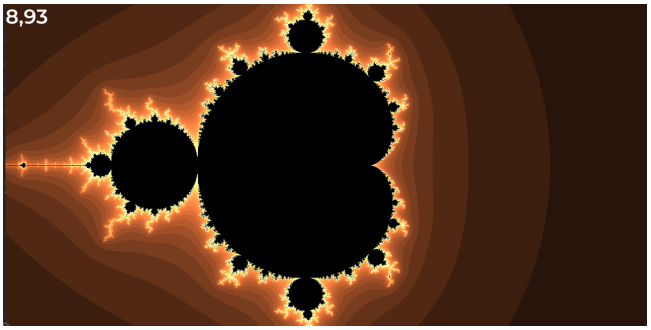


Рис. 10: Визуализация множества Мандельброта с наивными оптимизациями

Значение FPS находится в районе 7.5-8.

3.2.3 Векторные инструкции (AVX2)

Теперь начнём применять оптимизации, для которых придётся изменять код, и, к сожалению, приводить его к менее читаемому виду. Задействуем функциональный блок, который умеет обрабатывать SIMD-инструкции и перепишем код на интринсиках (intrinsics). Это специальные функции, которые вместо себя подставляют соответствующую ассемблерную инструкцию. Эти инструкции являются аппаратно-зависимыми, поэтому для различных процессоров будет свой список и свои названия. Характеристики машины, которая используется в данной работе, приведены в главе Оборудование.

Будем использовать тип `__m256`, который сможет хранить 8 значений типа `float`. Соответственно итоговый прирост к скорости должен быть примерно в 8 раз по сравнению с прошлой версией без учёта накладных расходов.

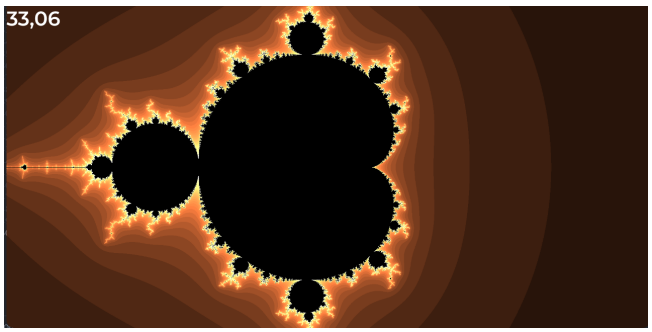


Рис. 11: Визуализация множества Мандельброта на интринсиках

FPS вырос примерно в 4 раза, а не в 8. Это можно объяснить тем, что отрисовка пикселей и обработка событий SDL2 сравнима по времени выполнения с исследуемыми расчётами. Следовательно, далее анализировать показатель FPS не имеет смысла, так как он некорректно показывает вклад применённых оптимизаций.

3.2.4 Развёртывание (loop unrolling)

Мы уже применили векторные инструкции, которые выполняются на отдельном функциональном блоке, но ведь в нашем коде невозможно полностью избежать зависимости по данным просто перестановкой операций. Поэтому, для того чтобы полностью загрузить конвейер, применим развёртывание цикла.

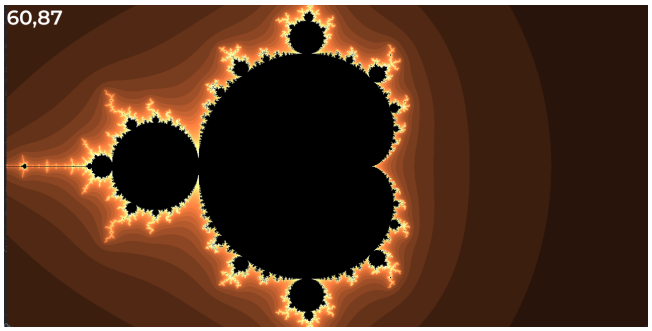


Рис. 12: Визуализация множества Мандельброта на интринсиках с развёртыванием

3.2.5 Кроссплатформенность

Как упоминалось ранее интринсики аппаратно-зависимы. Но компилятор может сам заменять циклы на векторные операции, если заметит возможность такой оптимизации. Тогда можно написать программу на массивах размером 8, которые с первого взгляда будут только ухудшать производительность, но при правильной организации кода будут заменяться соответствующими векторными инструкциями. В таком случае код будет переносимым и не должен сильно потерять в производительности.

FPS упал даже ниже, чем на версии с интринсиками без развёртывания (значение FPS версии программы на массивах без развёртывания было таким же). Скорее всего это произошло из-за того, что компилятор не смог заменить некоторые неочевидные аналоги векторных инструкций. Это наталкивает на идею оценить производительность при сборке разными компиляторами.

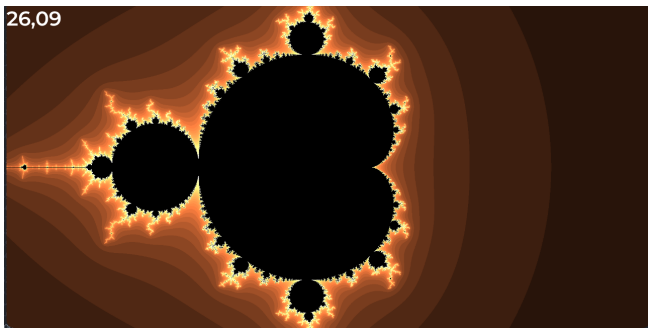


Рис. 13: Визуализация множества Мандельброта на массивах с развёртыванием

3.3 Точная оценка производительности

3.3.1 Проверка стационарности

Прежде чем проводить измерения нужно убедиться, что соблюдается стационарность процесса. Установим условия, при которых собираемся прогонять программу (частота процессора - 1Ghz, наивысший приоритет программы (nice -n -20), отключение графического режима) и построим график зависимости времени выполнения от итерации.

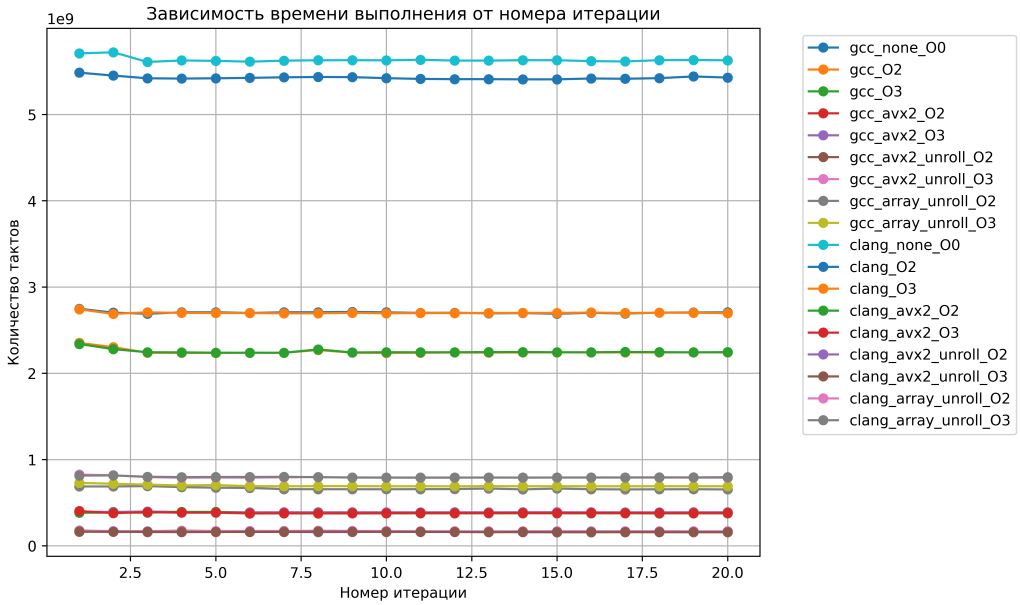


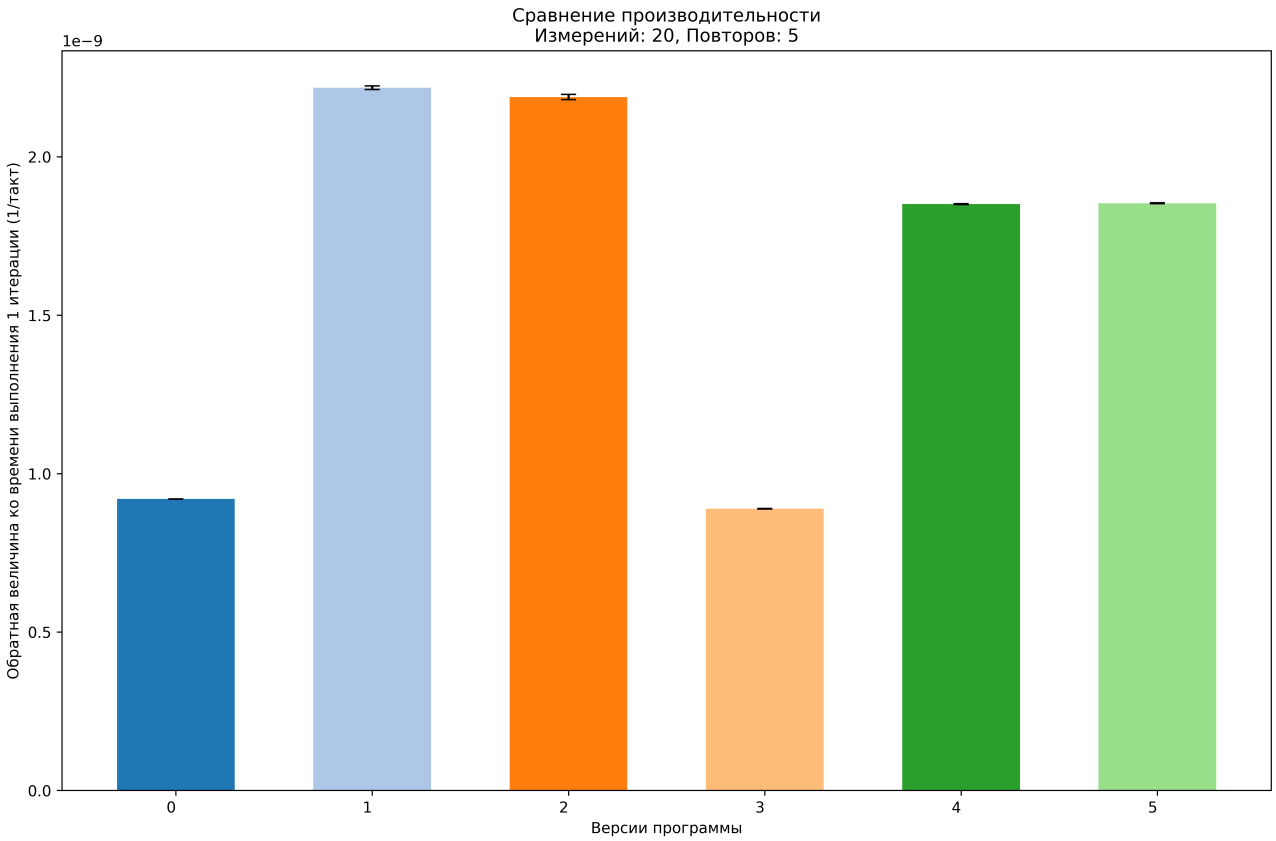
Рис. 14: График зависимости времени выполнения от итерации

Отклонения от нормы находится в пределах 1%, а значит измерениям можно будет доверять.

3.3.2 Измерения

Baseline. Имеет смысл оценивать прирост производительности оптимизаций, которые требуют переработки кода, так как оптимизирующие флаги компиляции и так используются в release-версии программы, поэтому выберем baseline, то есть версию, относительно которое будем смотреть прирост оптимизаций.

Над каждым столбиком гистограммы написано среднее время расчёта 1 кадра выраженное в тактах со среднеквадратичной погрешностью, а также прирост производительности относительно худшего результата. Как вы можете видеть, в лучшей конфигурации применялся



Номер	Компилятор	Флаг -O	Оптимизации	Производительность	Время $(\frac{\text{такт}}{\text{кадр}} \cdot 10^6)$
0	gcc	-O0	нет	1.034x ± 0.001x	1087.2 ± 0.6
1	gcc	-O2	PGO	2.494x ± 0.007x	450 ± 1
2	gcc	-O3	PGO	2.461x ± 0.009x	451 ± 1.2
3	clang	-O0	нет	1.000x ± 0.001x	1125 ± 0.9
4	clang	-O2	PGO	2.081x ± 0.002x	540.4 ± 0.4
5	clang	-O3	PGO	2.084x ± 0.002x	539.6 ± 0.4

Рис. 15: Сравнение производительности версий программы с базовыми оптимизациями

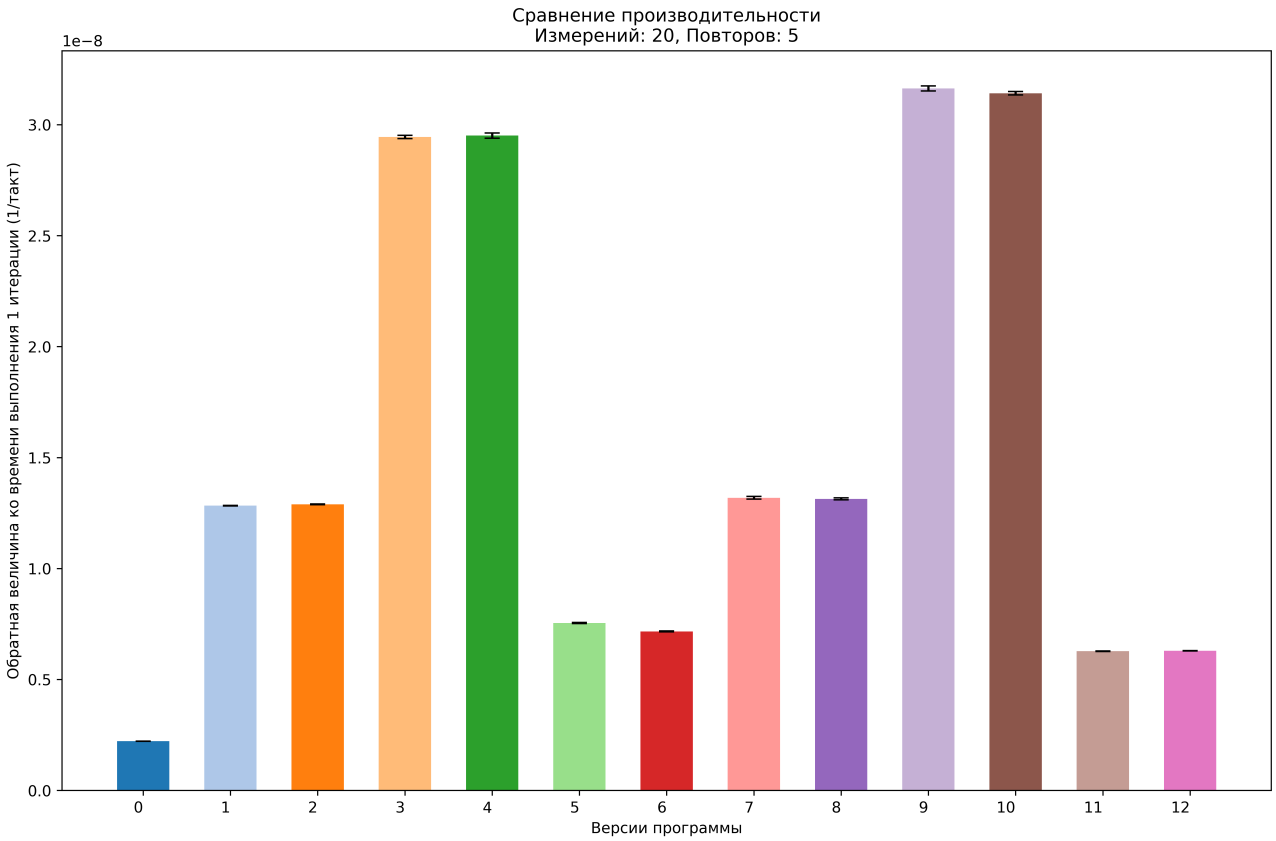
уровень оптимизации -O2 и использовался компилятор gcc. Далее прирост производительности будем смотреть относительно этой версии.

Ручные оптимизации. Теперь посмотрим ускорение оставшихся версий программы.

Как вы можете видеть самой лучшей версией по производительности является та, в которой использовались векторные инструкции, реализованные через интринсики, с флагов оптимизации -O2 развёртыванием цикла на 4 и компилятором clang. Версии с флагом оптимизации -O3 могут показывать себя немного хуже, так как применяют агрессивные методы, которые могут ухудшить производительность.

Версия на массивах в данной работе показала плохую эффективность, по сравнению с непереносимыми аналогами. Это может быть связано с недостаточно очевидным для компилятора заменами интринсиков на массивы.

Ожидаемый прирост лучшей версии (в $8 * 4 = 32$ раза) в более чем 2 раза больше того, что в итоге получился. Это можно объяснить тем, что базовые оптимизации обычной версии улучшали код именно в тех местах, которые мы заменили на векторные инструкции, при этом аналогичные оптимизации к векторам применить не удалось. При этом, если рассматривать прирост лучшей версии относительно той, в которой не было применено



Номер	Компилятор	Флаг -O	Оптимизации	Производительность	Время ($\frac{\text{такт}}{\text{кадр}} \cdot 10^6$)
0	gcc	-O2	PGO	1x ± 0.004x	451 ± 1.2
1	gcc	-O2	PGO, AVX2	5.786x ± 0.016x	77.9 ± 0.1
2	gcc	-O3	PGO, AVX2	5.813x ± 0.016x	77.6 ± 0.1
3	gcc	-O2	PGO, AVX2, unroll	13.27x ± 0.047x	34.0 ± 0.1
4	gcc	-O3	PGO, AVX2, unroll	13.3x ± 0.063x	33.9 ± 0.1
5	gcc	-O2	PGO, array, unroll	3.4x ± 0.015x	132 ± 0.5
6	gcc	-O3	PGO, array, unroll	3.23x ± 0.014x	139 ± 0.5
7	clang	-O2	PGO, AVX2	5.94x ± 0.032x	75.8 ± 0.4
8	clang	-O3	PGO, AVX2	5.92x ± 0.025x	76.1 ± 0.3
9	clang	-O2	PGO, AVX2, unroll	14.26x ± 0.063x	31.6 ± 0.1
10	clang	-O3	PGO, AVX2, unroll	14.16x ± 0.05x	31.8 ± 0.1
11	clang	-O2	PGO, array, unroll	2.828x ± 0.009x	159 ± 0.3
12	clang	-O3	PGO, array, unroll	2.837x ± 0.009x	159 ± 0.3

Рис. 16: Сравнение производительности версий программы с ручными оптимизациями

никаких оптимизаций, то прирост получается в $2.49 * 14.26 \approx 35.5$ раз, что коррелирует с предполагаемой эффективностью.

4 Вывод

В ходе выполнения данной работы была проведена комплексная оценка эффективности различных оптимизаций при вычислении множества Мандельброта. Основные этапы:

4.1 Теоретический анализ

- Рассмотрены базовые модели выполнения инструкций (SEQ и PIPE), а также современные суперскалярные и конвейерные архитектуры
- Изучены методы оптимизации, включая вынос кода, уменьшение рисков по данным, развёртывание циклов, использование аккумуляторов и SIMD-инструкций
- Описаны способы измерения производительности, включая фиксацию тактовой частоты процессора и использование инструкции `rdtsc`

4.2 Практическая реализация

- Разработана программа для визуализации множества Мандельброта с возможностью настройки параметров и отключения графики
- Последовательно применены различные оптимизации:
  - Базовые (флаги компилятора `-O2`, `-O3`, PGO)
  - Векторные инструкции (AVX2 через интринсики)
  - Развёртывание циклов
  - Кроссплатформенная реализация через массивы
- Проведены тесты на стационарность процесса для обеспечения достоверности измерений

4.3 Оценка производительности

- Наилучший результат показала версия с интринсиками, развёртыванием цикла на 4 порядка и компиляцией через `clang` с флагом оптимизации `-O2`
- Максимальный прирост производительности относительно базовой версии составил **∼35.5 раз**, что близко к теоретически ожидаемому ускорению (32x)
- Оптимизации компилятора (`-O3`) в некоторых случаях ухудшали производительность из-за агрессивных преобразований

4.4 Итоги

- SIMD-оптимизации обеспечили наибольший прирост производительности
- Развёртывание циклов и аккумуляторы дополнительно ускорили вычисления
- Кроссплатформенный подход на массивах показал более худшие результаты, но может быть полезен для переносимости

Заключение: Применение низкоуровневых оптимизаций (SIMD, развёртывание циклов) в сочетании с правильными флагами компиляции позволяет значительно ускорить вычисления. Однако эффективность зависит от архитектуры процессора и качества оптимизации компилятора. Для достижения максимальной производительности рекомендуется использовать аппаратно-зависимые инструкции (интринсики).

5 Полезные ссылки

1. Сюда звёздочки ставить
2. Компьютерные системы. Архитектура и программирование [2022] Брайант Р. Э., О'Халларон Д. Р.
3. (Визуализация продвижения)
4. Зеркало сайта с интрисинками на intel