Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра вычислительных систем

**ОТЧЕТ**

по лабораторной работе

по дисциплине «Параллельные вычислительные технологии»

Выполнил:

студент группы

ИС-241 Жур А. А.

Проверил:

ассистент кафедры ВС

Челканова Т. В.

Новосибирск 2024

**Оглавление**

[ПОСТАНОВКА ЗАДАЧИ 3](#_TOC_250003)

[ОПИСАНИЕ ТЕОРИИ МЕТОДА РЕШЕНИЯ ЗАДАЧИ 5](#_TOC_250002)

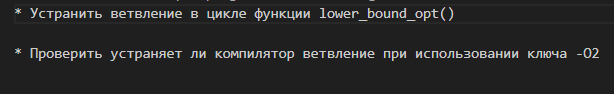
[СХЕМА АЛГОРИТМА РЕШЕНИЯ 6](#_TOC_250001)

ЛИСТИНГ ПРОГРАММЫ С КОММЕНТАРИЯМИ 7

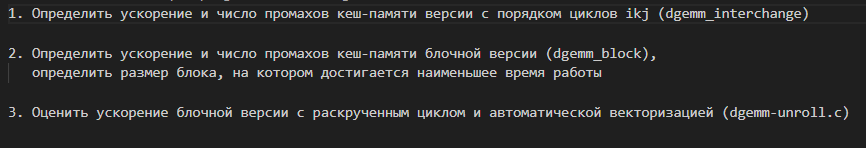
[РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ 9](#_TOC_250000)

# ПОСТАНОВКА ЗАДАЧИ

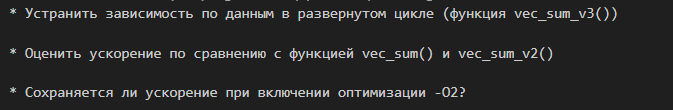
Практическая работа 1.1 (cache)



Практическая работа 1.2 (branch)

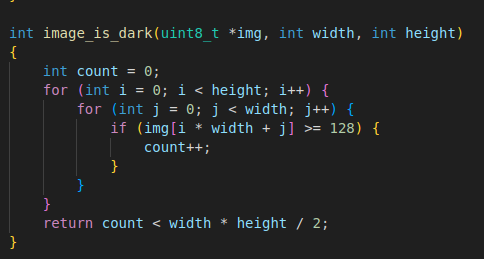


Практическая работа 1.3 (loop-unrolling)

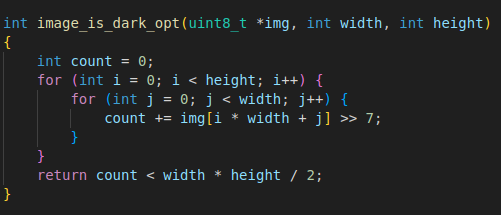


# Задание 1.1

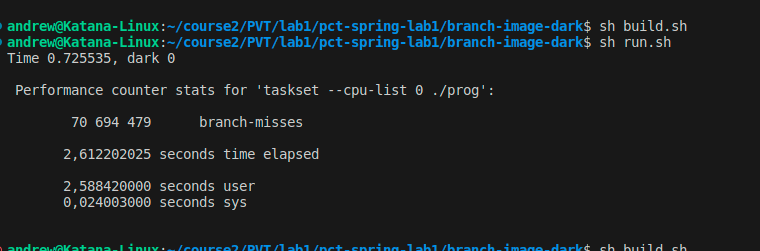
Рассмотрим код исходной функции:

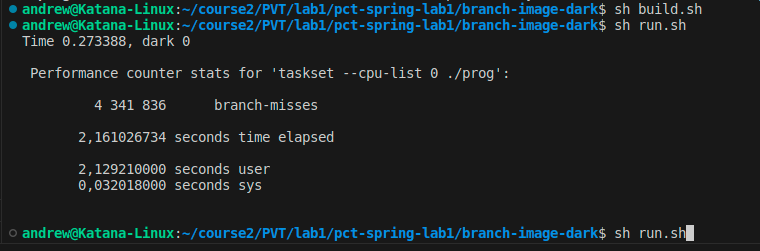


Эта функция проверяет тёмное ли это изображение. Тёмным считается изображение, в котором меньше половины пикселей имеют значение интенсивности >=128. У нас есть 2 цикла, проверяющих это условие. В них мы проходим по всем пикселям и считаем только яркие.

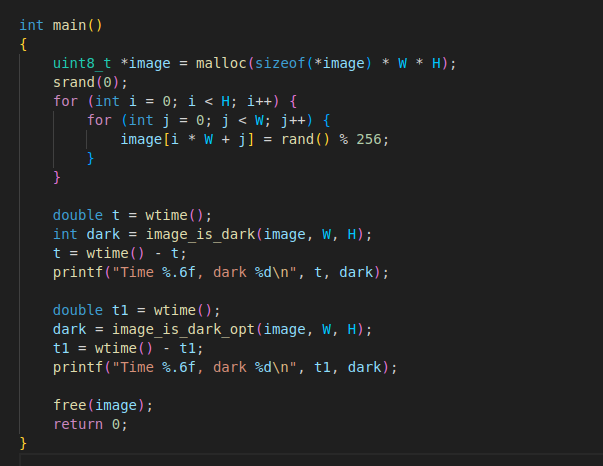


Это та же функция, но с устранённым ветвлением в циклах. Тут используется побитовый сдвиг в качестве деления, и к счётчику count прибавляется 0 или 1(сдвигаем на 7 т. к. 128 это 2^7 и в двоичном виде 1 стоит в 7 разряде).

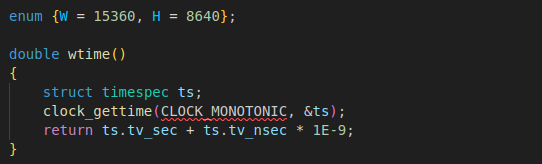




На первом скриншоте продемонстрирована работа программы с ветвлением, а на втором с его устранением. С помощью perf выводится информация о branch-misses -это количество промахов модуля предсказания переходов.

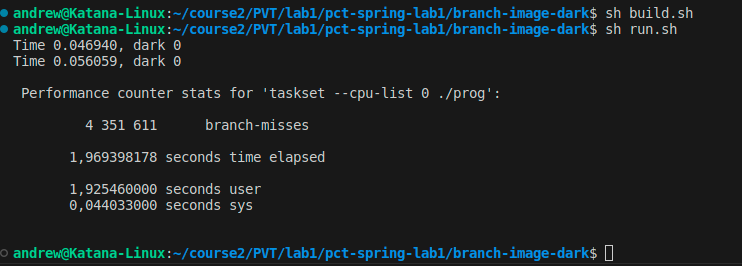


В функции main мы генерируем наше изображение а затем считаем и выводим время для версии с ветвлением и оптимизированной версии.



Тут представлены размер нашей картинки в пикселях и вспомогательная функция для подсчёта времени.

Дальше проверим, устраняет ли ключ -О2 ветвление:

Получается, что при ключе -О2 компилятор устраняет ветвление. Программа выполнилась ещё быстрее и с меньшем количеством промахов.

Ответы на контрольные вопросы по блоку 1.1 (cache):

1. Неэффективно используется кеш-память при обращении к массиву с большим шагом, превышающим размер строки кеша. Например, если шаг равен размеру строки кеша или более. Тогда каждая следующая выборка данных вызывает cache miss, что значительно замедляет выполнение программы.

2. Физический адрес разбивается на три части: индекс набора, тэг и смещение. 1. По полю Set index выбирается одно из S множеств 2. Среди E записей множества отыскивается строка с требуемым полем Tag и установленным битом Valid. Найдена – cache hit, не найдена – cache miss 3. Данные из блока считываются с заданным смещением Block offset.

3. Если все каналы множества заняты, кеш-память использует политику замещения для выбора линии кеша, которая будет заменена новыми данными. Алгоритм замещения строк кеш-памяти (replacement policy) — выбирает строку и удаляет из кеш-памяти для размещения новой записи.

Алгоритмы требуют хранения вместе с каждой строкой кеш-памяти специализированного поля флагов/истории (age bits).

LRU (Least Recently Used) – вытесняется наименее востребованную строку (2Q, LRU/K).

RR (Random Replacement) – вытесняет случайную строку.

Алгоритм L. Belady – вытесняет запись, которая с большой вероятностью не понадобиться в будущем (IBM Research, 1966).

4. Политика write-through (сквозная запись) – запись в кеш-память влечет за собой немедленное обновление данных в кеш-памяти и оперативной памяти (кеш “отключается”)

Политика write-back (отложенная запись, copy-back) – первоначально данные записываются только в кеш-память

Строки нет в кеш-памяти (write miss):

 В кеш-памяти выделяется строка

 Из оперативной памяти загружается строка, соответствующая адресу

 Необходимые байты изменяются в загруженной строке кеш-памяти

 Строка помечается как модифицированная (dirty)

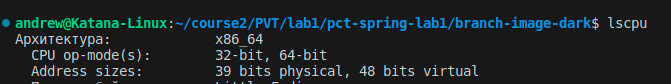
Строка присутствует в кеш-памяти (write hit):

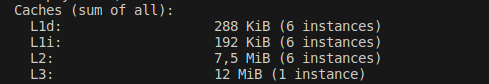
 Необходимые байты изменяются в строке кеш-памяти

 Строка помечается как модифицированная (грязная, dirty)

Запись в память модифицированных строк осуществляется при их замещении

5.





# Задание 1.2

# 

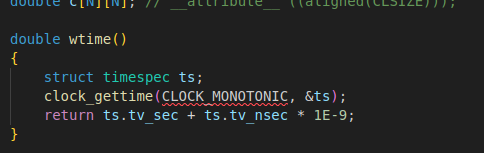
CACHELINE\_SIZE определяет размер кэш линии

#define N 1024: размер матриц (N x N).

#define NREPS 3: Количество повторений для измерения среднего времени.

#define BS 32: Определяет размер блока для блочного умножения матриц.

#define IMIN(a, b) (a < b ? a : b): вычисляет минимум из двух значений.



# Функция для измерения времени, используемая ранее.

# Далее идут функции для инициализации матриц и их умножения:

# 

# Матрицу А умножаем на матрицу В и результат записываем в матрицу С.

# 

# dgemm\_interchange — улучшенная версия обычного умножения путём перестановки циклов местами. Таким образом уменьшаются кеш-миссы. Доступ к памяти становится более последовательным .

# 

dgemm\_block - блочное умножение матриц, разбивает исходные матрицы на маленькие блоки. Матрицы разбиваются на подматрицы каждая подматрица полностью помещается в кеш.

# 

В функции dgemm\_verify() мы проверяем корректность выполнения алгоритмов умножения матриц, сравнивая результаты их работы с классическим способом умножения матриц.

# 

# 

# В функции main мы инициализируем матрицы и по очереди проверяем каждый способ для оценки времени выполнения.

# Работа обычной версии:

# 

# Работа с перестановками циклов:

# 

# Работа блочной версии:

# 

# Из результатов следует, что версии с другим порядком циклов и блочная показывают эффективное время работы, а также более эффективное время работы с кэш-памятью.

# Считаем ускорения:

# Перестановка: 9,911/3.498 =2,833

# Блочная: 9,911/4.522 = 2,192

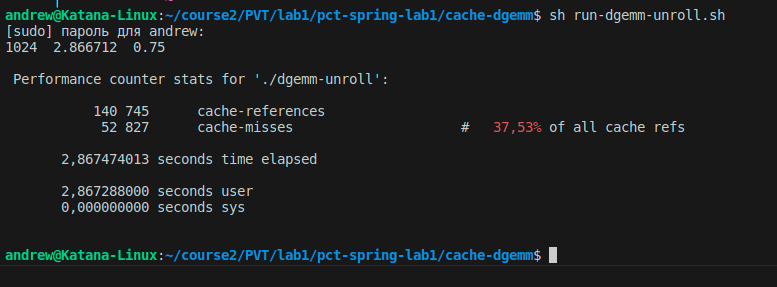
# Последний способ оптимизации в данном задании это блочная версия с раскрученным циклом и автоматической векторизацией:

# 

#pragma GCC ivdep указывает компилятору игнорировать зависимости между итерациями цикла, что позволяет компилятору лучше оптимизировать код.

#pragma GCC unroll 8 указывает компилятору развернуть внутренний цикл на 8 итераций.

Результат:



Ускорение: 9,911/2,867 = 3,457

Данная версия имеет наибольшее ускорение за счёт способов её реализации.

Ответы на контрольные вопросы по блоку 1.2 (branch):

1. Конфликт управления (control hazard) возникает при ветвлениях, когда процессор не знает, какое из двух возможных направлений (веток) выбрать, что может привести к простою конвейера.

2. На вход подаются адрес текущей инструкции и, возможно, информация о предыдущих переходах. Модуль предсказания переходов выдаёт предсказанный адрес следующей инструкции.

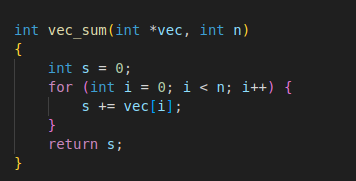
3. Двухбитный предсказатель использует два бита для каждой записи, которые хранят одно из четырёх состояний: "сильное не переходить", "слабое не переходить", "слабое переходить", "сильное переходить". В зависимости от результата, состояние изменяется.

4. Для Intel Core i5-11400H, как и для других процессоров на архитектуре Willow Cove, предсказатель переходов включает несколько усовершенствований и технологий:   
  
- TAGE является одним из самых передовых типов предсказателей переходов, использующих несколько таблиц с разными историческими данными для предсказания.   
Micro-op cache:   
- Кэш микроопераций (Micro-op Cache) сохраняет декодированные инструкции, что уменьшает необходимость повторного декодирования и ускоряет выполнение ветвлений.   
Return Stack Buffer (RSB)  
- Буфер возвратов, который отслеживает адреса возврата для инструкций вызова функции и возврата из функции, улучшая предсказание для этих часто встречающихся конструкций.

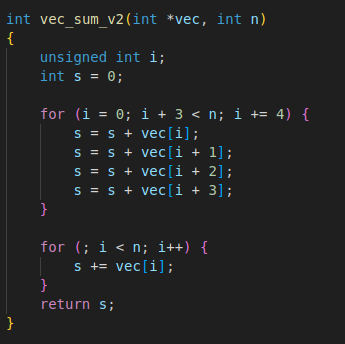
Задание 1.3

В этом задании считаем сумму элементов вектора.

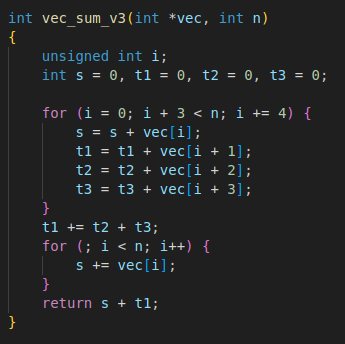
Стандартная функция для данной задачи:



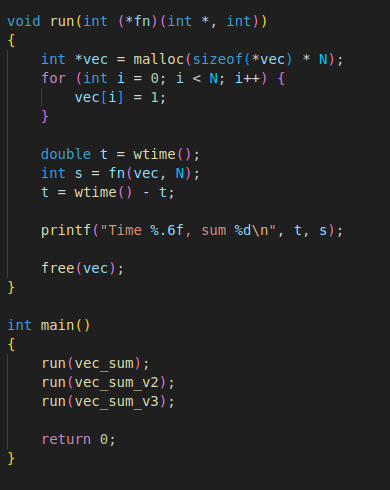
В переменную s складываем все элементы массива.

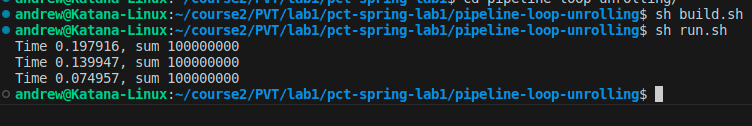


В vec\_sum\_v2 мы делаем развёртывание цикла и суммируем сразу несколько элементов массива, задействовав все alu нашего процессора. Однако возникает зависимость по данным, которую мы устраняем в vec\_sum\_v3 с помощью локальных переменных:



Далее мы запускаем наши функции и считаем время выполнения каждой:



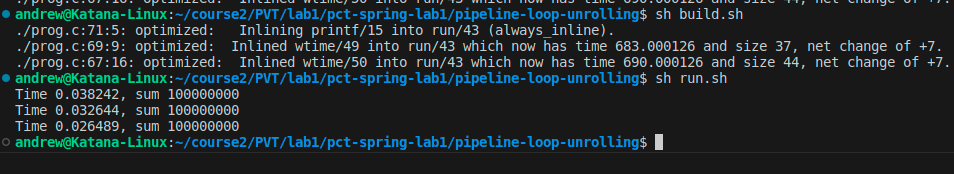


Оценим ускорение vec\_sum\_v3 по сравнению с другими:

с vec\_sum: 0.1979/0.0749 = 2.642

с vec\_sum\_v2:0.1399/0.0749 = 1.868

Проверка оптимизации с ключом -О2:



Время работы уменьшилось. Проверим ускорение:

с vec\_sum: 0.038242/ 0.026489 = 1.442

с vec\_sum\_v2: 0.032644/ 0.026489 = 1.232

Ускорение уменьшилось поскольку первая версия начала работать быстрей.

Ответы на контрольные вопросы по блоку 1.3 (loop-unrolling):

1. Развёрнутый цикл работает быстрее, потому что уменьшается количество итераций цикла, что позволяет сократить накладные расходы на развёртывание цикла, а также развёртывание позволяет выполнять польше команд параллельно

Что касается суперскалярных процессоров, то они могут выполнять несколько инструкций

за один такт, и развёртывание помогает им в том, что увеличивает количества доступных

независимых инструкций что позволяет использовать все возможности суперскалярному

процессору, а также снижает зависимости от ветвлений

2. это зависит от самого процессора и его характеристик то со сколькими данными

он может работать.

3.