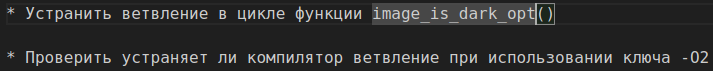
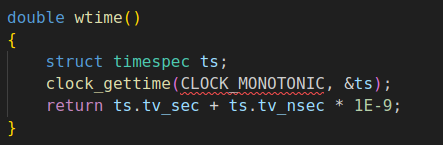
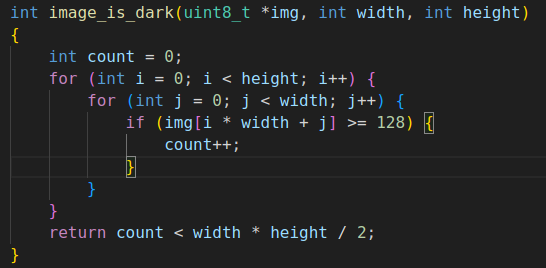
**Отчёт по практической работе №1**

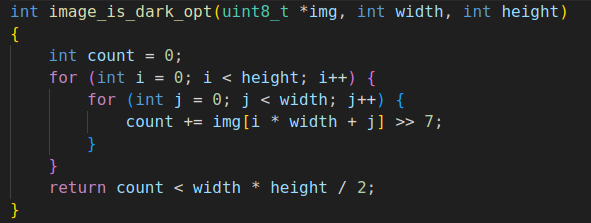
**1 Часть branch**

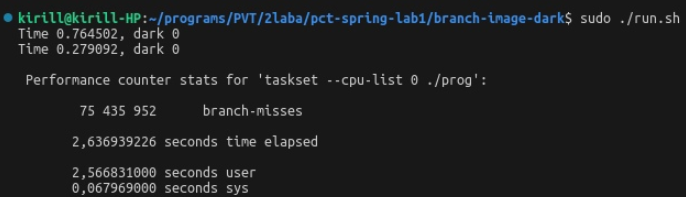
**Задание:**

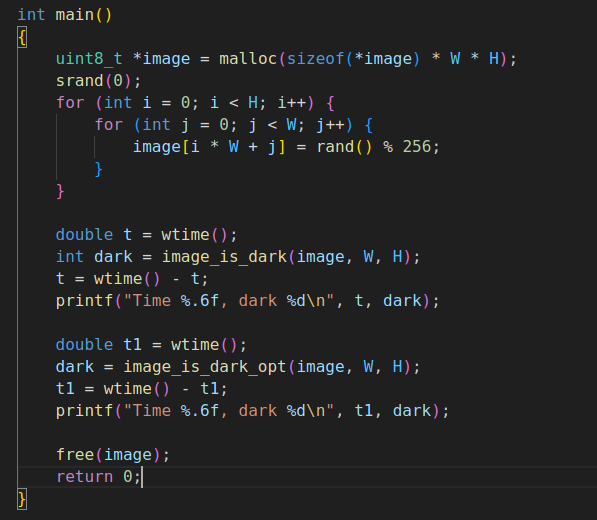
Рассмотрим код программы:

Эта функция используется для измерения времени выполнения других функций.

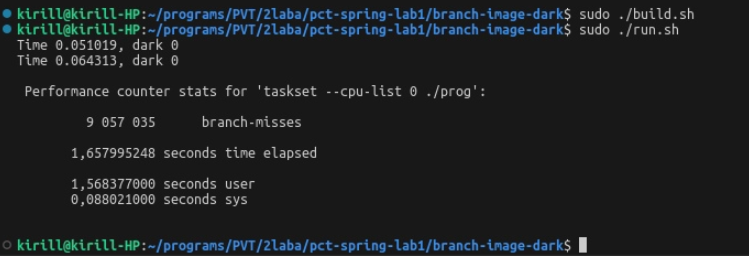
Эта функция анализирует изображение, представленное массивом img размером width на height, и определяет, является ли оно темным. Темным считается изображение, в котором менее половины пикселей имеют значение интенсивности 128 или выше. Проверка проходит с помощью двойного цикла который проходит по всем пикселям и считает количество ярких пикселей.

В функции image\_is\_dark\_opt() мы устраняем по заданию ветвления в цикле устраняем с помощью побитового сдвига он используется в качестве деления и в счётчик count либо прибавляется 1 либо 0 таким образом эта функция у нас выполняет те же самые действия что и image\_is\_dark() но в ней мы не используем ветвление и условия что помогает оптимизировать нашу программу и ускорить её выполнение в чём можно убедиться запустив её.

Первое время выполнения это функция с ветвлением, а второе это программа без ветвления. Также с помощью инструментария perf выводится информация о branch-misses -это количество промахов предсказания ветвлений.

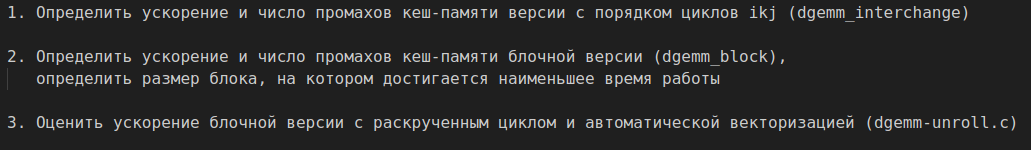
Функция main в которой в которой и происходят действия программы, в ней мы генерируем наше изображение и выполняем наши функции анализа изображения и считаем время их выполнения, выводим результат, освобождаем память.

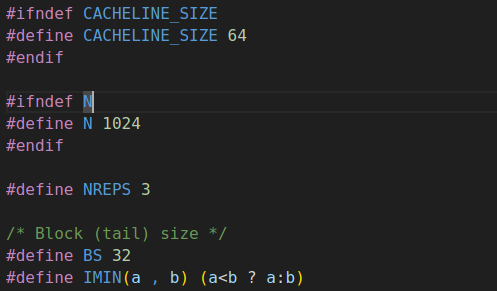
2 наша задача в этой части это проверить устраняет ли компилятор ветвление при использовании ключа -O2, компилируем с помощью этого ключа и снова запускаем программу:

Можем заметить значительное уменьшение времени выполнения функций и количество промахов по ветвлению. Потому что оптимизация -О2 включает различные техники улучшения производительности, такие как устранение избыточных ветвлений, предсказание ветвлений и другие оптимизации.

**2 Часть Cache-dgemm**

**Задание:**

Начнём с кода нашей программы

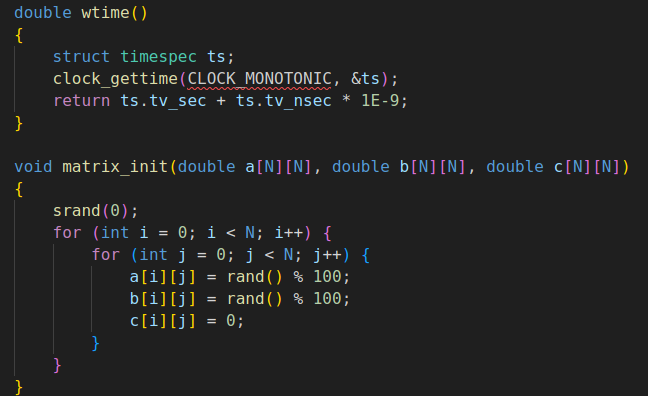
CACHELINE\_SIZE 64 определяет размер кэш линии

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

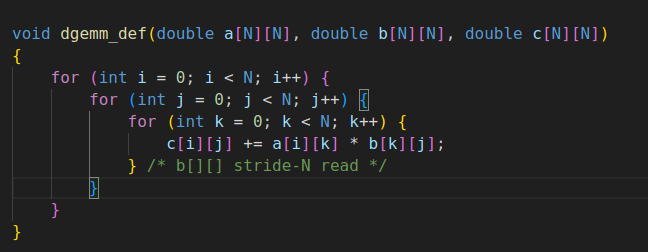
#define NREPS 3: Определяет количество повторений для измерения среднего времени выполнения.

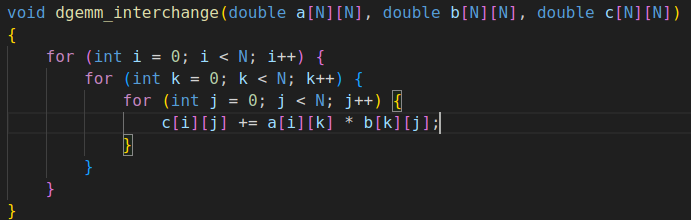
#define BS 32: Определяет размер блока для блочного умножения матриц.

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

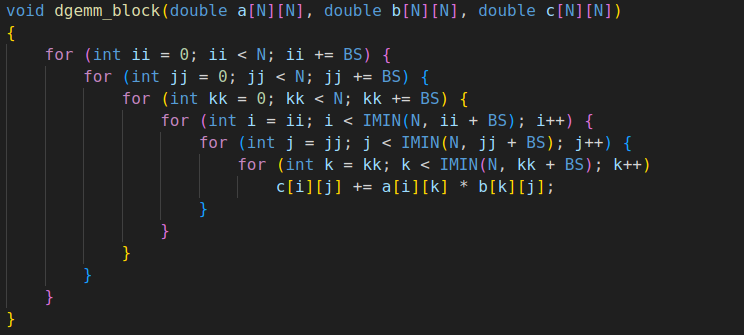


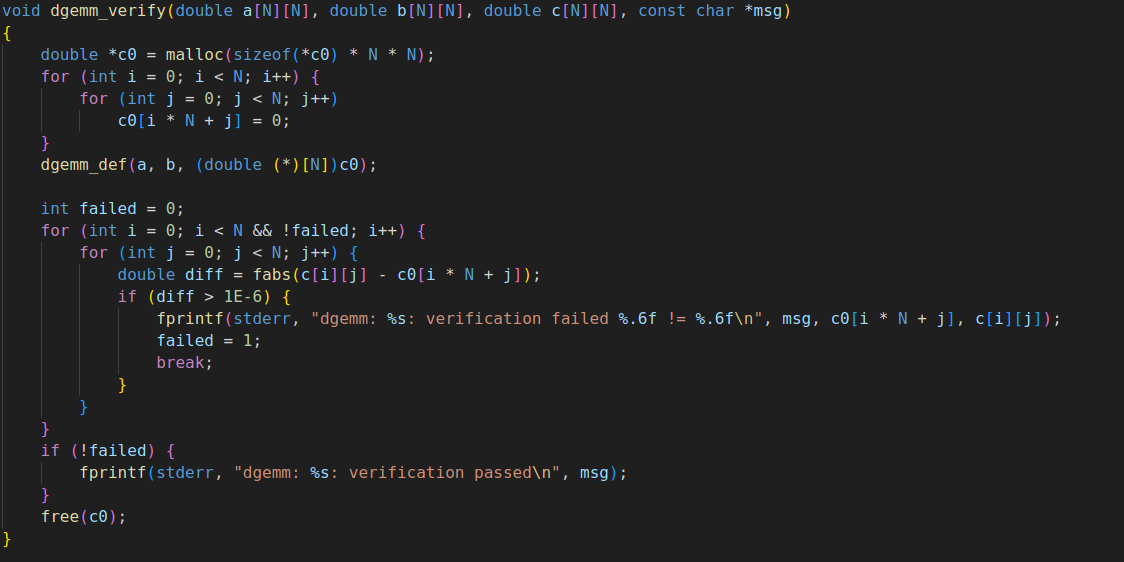
Wtime() используется для вычисления времени, а matrix\_init() Инициализирует матрицы a и b случайными значениями, а матрицу c — нулями.

  
Функция dgemm\_def() выполняет умножение двух матриц a и b размером N x N и сохраняет результат в матрице c. Это простой алгоритм умножения матриц с тройным вложенным циклом.

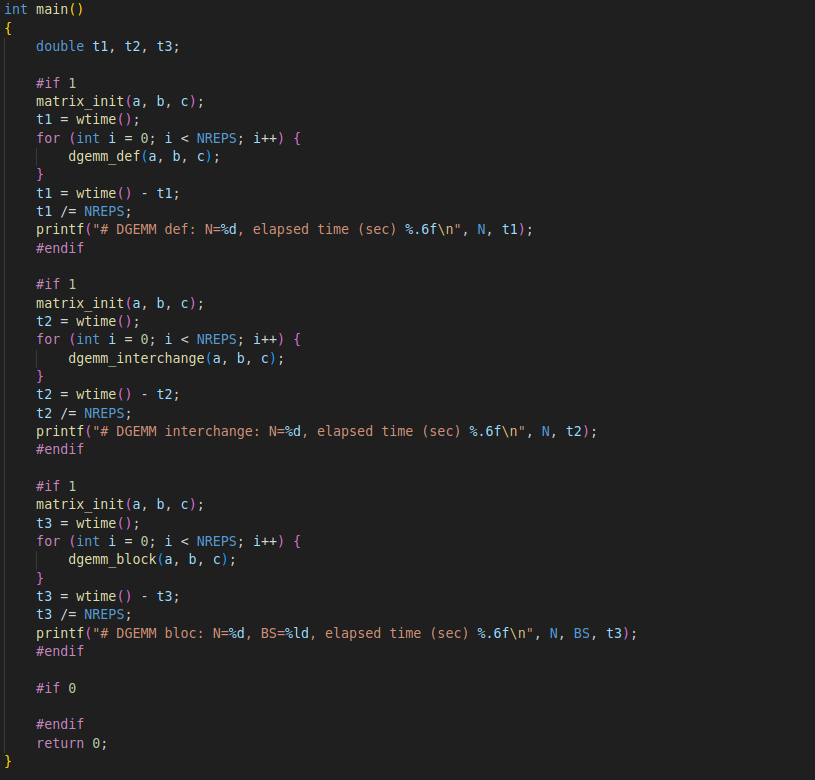


В функции dgemm\_interchange() выполняем перестановку вложенных циклов. В результате такой перестановки улучшается производительность умножения матриц за счет более последовательного доступа к памяти и снижения количества кеш-промахов.

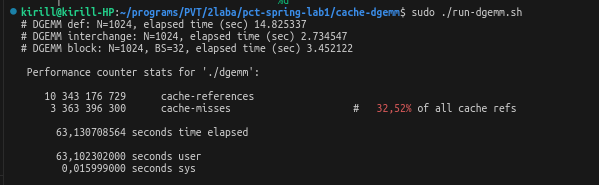
  
Функция dgemm\_block реализует блочное умножение матриц, разбивая большие матрицы на меньшие блоки. Матрицы разбиваются на подматрицы BSxBS rаждая подматрица (блок) полностью помещается в кеш-память.



Функция dgemm\_verify() проверяет корректность выполнения алгоритмов умножения матриц, сравнивая результаты их работы с классическим способом умнжения матриц.



В функции main() мы выполняем умножения матриц 3 способами: стандартным, с порядком циклов ikj и блочным когда разбиваем матрицу на подматрицы, кроме того вычисляет время выполнения для каждого способа теперь запустим эту программу.



Выполнив программу, можем заметить что версии умножения матрицы отличные от обычного умножения выполняются быстрее, как и предполагалось, теперь оценим ускорение модифицированных версий от обычной, для этого нужно время выполения обычной версии разделить на время выполнения той версии, ускорение которой мы хотим определить.

Для начала определим ускорение версии с порядком циклов ikj (dgemm\_interchange)

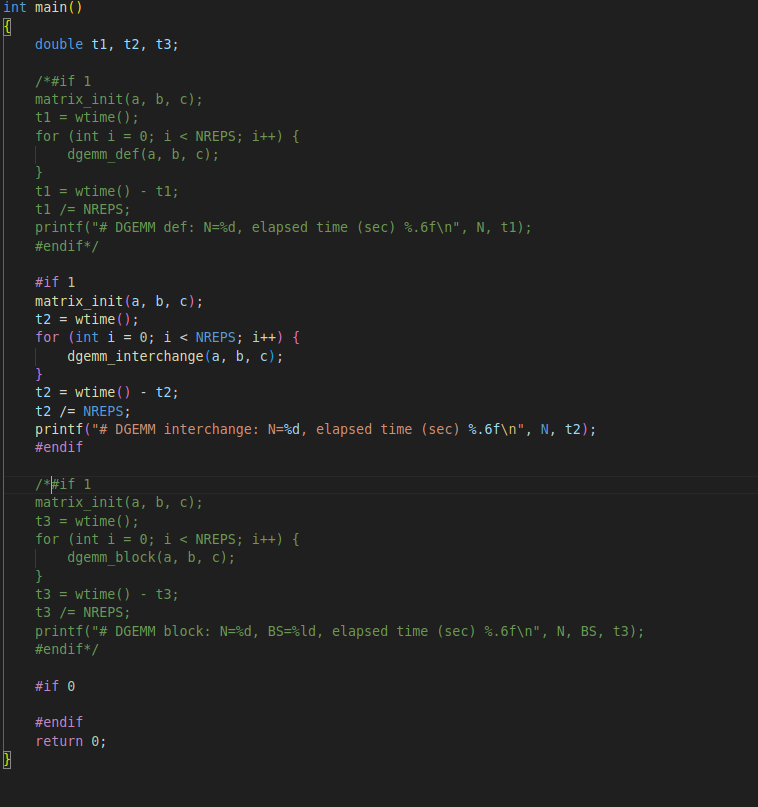
14.825337/2.734547=5,4215-ускорение версии dgemm\_interchange).

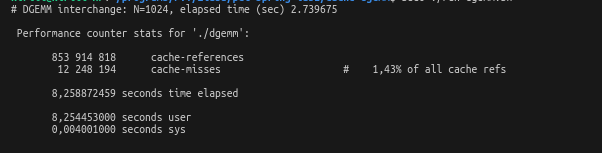
Теперь ускорение блочной версии (dgemm\_block)

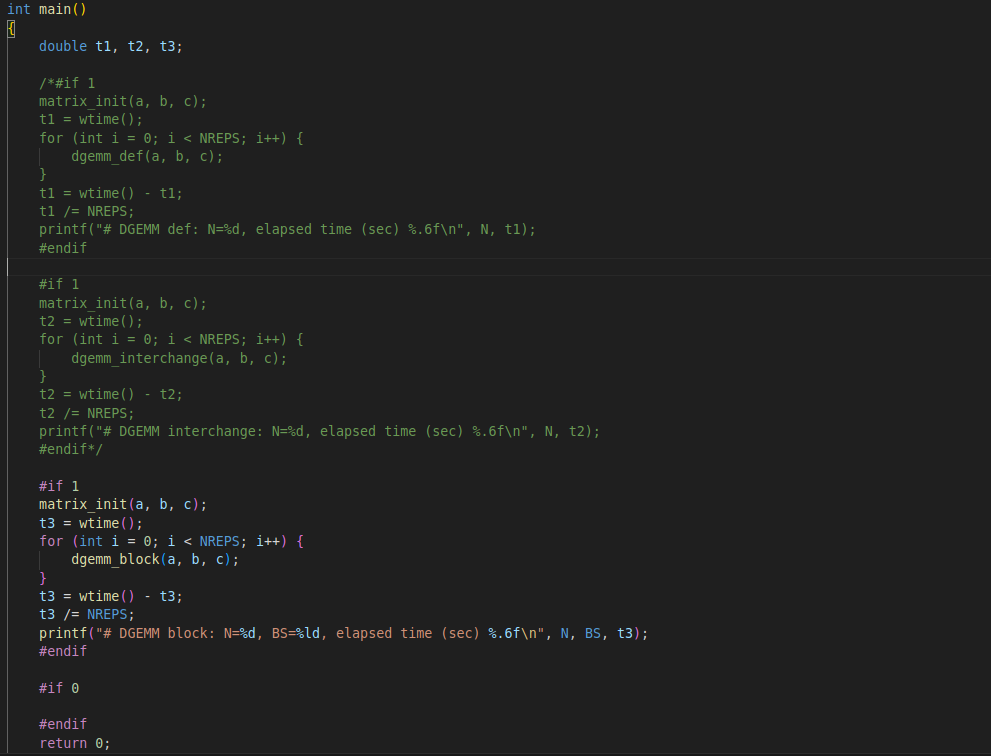
14.825337/3.452122=4,2946-ускорение блочной версии (dgemm\_block)

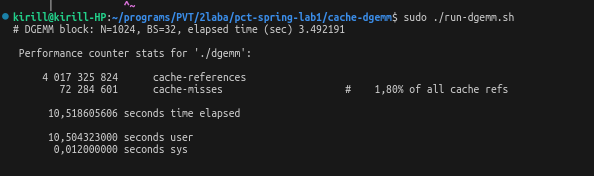
Также одной из задач этой части было определить размер блока на котором достигается наименьшее время работы для версии (dgemm\_block) путём тестирования был определён блок равный 32.

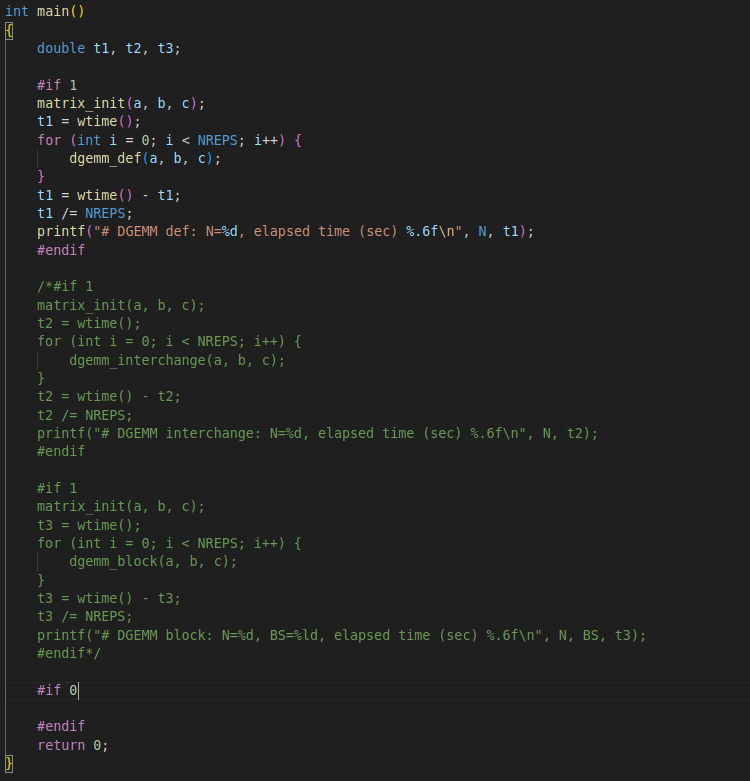
Кроме того нужно было определить число промахов кэш-памяти для версий dgemm\_interchange), (dgemm\_block), чтобы это сделать используем perf и выполним умножение матриц этими способами, запустив каждый способ отдельно для этого просто закомментируем ненужный участок кода.

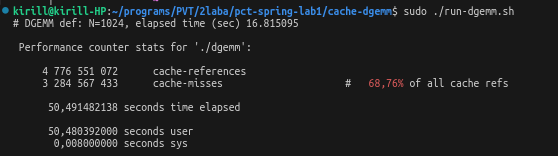
Результатом работы программы будет:

Теперь блочная версия :

 результат работы программы :

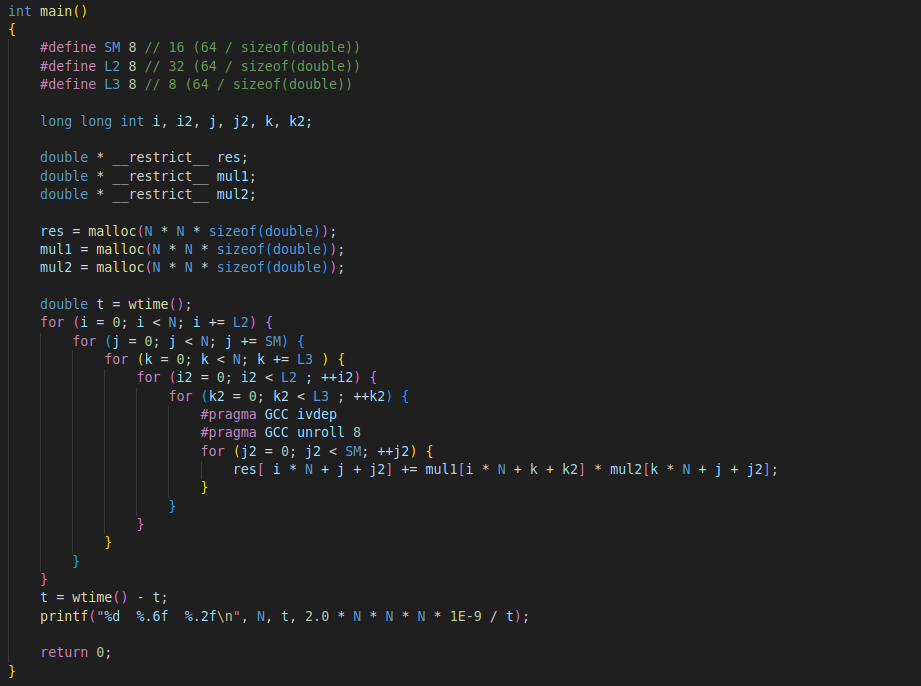
и для сравнения измерим пропуски кэша в обычной версии

результат работы программы:

Как мы можем видеть версии с другим порядком циклов и блочная показывают гораздо эффективное время работы, а также более эффективное время работы с кэш-памятью.

И последнее задание этой части : Оценить ускорение блочной версии с раскрученным циклом и автоматической векторизацией (dgemm-unroll.c).

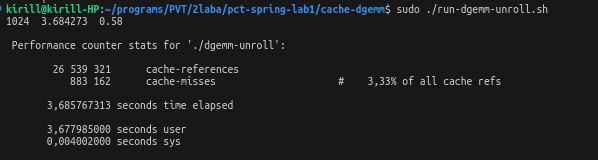
Используется блочный алгоритм умножения матриц для улучшения производительности за счет кэширования.

В этой версии также используется блочный алгоритм умножения матриц, а также:

#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, что также улучшает производительность.

Результат выполнения этой версии:

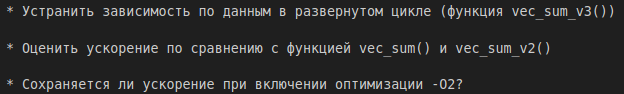


Оценим ускорение этой версии со стандартной:

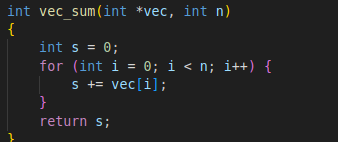
14.825337/3,684273=4,0234

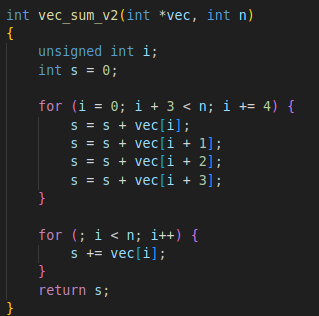
**3 Часть pipeline-loop-unrolling**

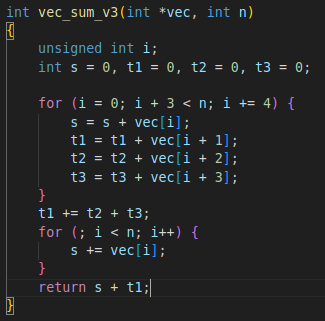
**Задача:**

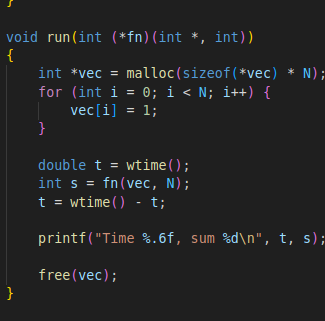
****

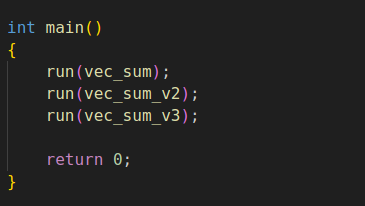
**Рассмотрим код**

Функция vec\_sum() принимает указатель на массив vec и его размер n. Она итерируется по всем элементам массива, добавляя каждый элемент к переменной s. Возвращает сумму всех элементов массива.

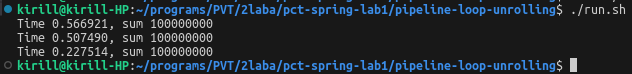
Эта функция выполняет те же действия, только в ней используется развёртывание цикла, то есть за одну итерацию в цикле суммируется 4 элемента массива. После выхода с основного цикла оставшиеся элементы, если их меньше 4 суммируются в оставшемся цикле.

Функция vec\_sum\_v3() как и vec\_sum\_v2() использует развёртывания цикла, но кроме этого в ней, элементы суммируются в отдельные переменные, что помогает устранить зависимость по данным.

Функция run() принимает указатель на функцию которую нужно выполнить, также в ней инициализируется массив и считается время выполнения функции.

В main() запускаем все функции.

Результат работы программы:

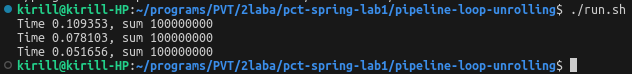


Как можно увидеть развёрнутый цикл в vec\_sum2() ускоряет выполнение программы в отличие от версии vec\_sum(), а в функции vec\_sum3() где устранена зависимость данных время выполнения ещё быстрее. По заданию оценим ускорение vec\_sum3() по сравнению с vec\_sum2() и vec\_sum()

0,566921/0,227514= 2,4918 ускорение vec\_sum3() по сравенению с vec\_sum()

0,507490/0,227514= 2,23 ускорение vec\_sum3() по сравенению с vec\_sum2()

Теперь проверим сохраняется ли ускорение при включении оптимизации -O2:



Сейчас посчитаем ускорение при компиляции с ключом -О2

0,109353/0,051656= 2,11 ускорение vec\_sum3() по сравнению с vec\_sum()

0,078103/0,051656= 1,511 ускорение vec\_sum3() по сравнению с vec\_sum2()

Ускорение при компиляции с ключом -О2, если сравнивать vec\_sum3() с vec\_sum() изменяется немного, а в сравнение vec\_sum3() vec\_sum2() прилично уменьшается.

**Ответы на контрольные вопросы**