МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский нижегородский
государственный университет им. Н.И. Лобачевского»
(ННГУ)**

**Институт информационных технологий, математики и механики
Кафедра Алгебры, геометрии и дискретной математики**

Направление подготовки: «Фундаментальная информатика и информационные технологии»
Магистерская программа: «Когнитивные системы»

ОТЧЕТ

по курсу:

«Анализ производительности и оптимизация программного обеспечения»

Выполнил: студент группы 382006-3м
Вихрев И. Б.

Нижний Новгород
2021

Содержание

Определения

- Секвенирование ДНК — определение нуклеотидной последовательности. В результате секвенирования получают формальное описание первичной структуры линейной макромолекулы в виде последовательности мономеров в текстовом виде.
- Короткие прочтения, риды (англ. Reads) — короткие последовательности нуклеотидов, получаемые в процессе секвенирования.
- k-mer — последовательность длины k, содержащаяся в геноме.

1 ПОСТАНОВКА ЗАДАЧИ

Рассмотрим распространненую задачу из области биоинформатики — картирование ридов на известный геном. После проведения секвенирования неизвестного генома, необходимо картировать полученные риды на уже известный геном, т.е определить позиции в референсном геноме или транскриптоме, откуда с наибольшей вероятностью могло быть получено каждое конкретное короткое прочтение. Обычно это является первой стадией в обработке данных.

Формально, можно сказать, что отображение коротких прочтений на известный геном — это проблема поиска подстроки в строке. Допустим, дан референсный геном

ATATGTTAGTCAAGTTATTAAGACTATGTTAG

И список ридов:

1. TCAAG
2. TGTA

Тогда, в результате работы алгоритма мы хотим получить список ридов и их наиболее вероятные позиции в геноме:

ATATGTTAGTCAAGTTATTTATGTCAG

1. TCAAG : 10 (идеальное совпадение)
2. TGTA: 3, 18, 26 (1 несовпадение)

Для определения степени различия двух строк вводится расстояние Хэмминга — число позиций, в которых соответствующие символы двух слов одинаковой длины различны.

Приведем упрощенную реализацию данного алгоритма для поиска идеального расположения рида на референсном геноме, основываясь на подходе с использованием хэширования. Такой алгоритм состоит из следующих шагов:

1. Строим хэш-таблицу всех k-меров референсного генома. Ключом в такой таблице будет являться k-mer, а значением – список позиций всех его вхождений в геном.

2. Для каждого короткого прочтения выделяем seed – некоторый k-mer, обычно из начала рида и ищем его в хэш-таблице.
3. Если k-mer присутствует в хэш-таблице, то идем по всем его позиция и считаем расстояния Хэмминга для подстроки генома и обрабатываемого рида. Для идельного совпадения выбираем подстроку из генома, для которой метрика равна 0. Найденную позицию сохраняем. Во всех остальных случаях сохраняем значение -1 – признак отсутствия идеального совпадения.

2 ОПИСАНИЕ БАЗОВОЙ ВЕРСИИ КОДА

2.1 Описание программной реализации

Исходная реализация алгоритма содержит несколько функций:

1. **ham_dist** — функция для вычисления расстояния Хэмминга. Принимает на вход две строки одинаковой длины, возвращает число отличных символов на соответствующих позициях.
2. **build_kmers_map** — функция, отвечающая за построение хэш-таблицы kmer'ов и их позиций в геноме. Принимает на вход строку генома и длину k-mer'а.
3. **simple_reads_mapping** — функция, содержащая реализацию самого алгоритма картирования ридов на референсный геном. Принимает на вход строку с геномом, список ридов и длину k-mer'а

Полный код программы находится в приложении.

2.2 Тестовая инфраструктура

В процессе проведения экспериментов использована инфраструктура, параметры которой приведены ниже:

CPU	Intel® Core™ i5-7300 CPU @ 2.60 GHz
Операционная система	Windows 10
Оперативная память	8 GB
Используемые компиляторы	MSVC C++ Compiler, Intel C++ Compiler 2022
Конфигурация:	Release with Debug Info, x64

В качестве тестовых входных файлов используются:

1. genome.fasta — fasta-файл с текстовым представлением кусочка генома, содержит 29903 нуклеотида.
2. reads.txt — текстовый файл, содержащий 10000 ридов различных длин.

3 АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ

Для анализа производительности воспользуемся инструментом Intel VTune Profiler. Он позволяет провести анализ разных аспектов приложения, из них наиболее релевантными в процессе оптимизации данной программы будут:

1. Hotspot анализ — нахождение участка кода в программе, на который приходится боольшая часть исполняемых инструкций процессора.
2. Memory access анализ — выявление проблем в производительности программы, связанных с памятью, определения числа обращений к памяти (Loads and Stores), определения промахов кэша последнего уровня (LLC Miss Count).
3. Threading анализ — определение того, насколько эффективно приложение использует доступные вычислительные ядра процессора.

3.1 Hotspot анализ

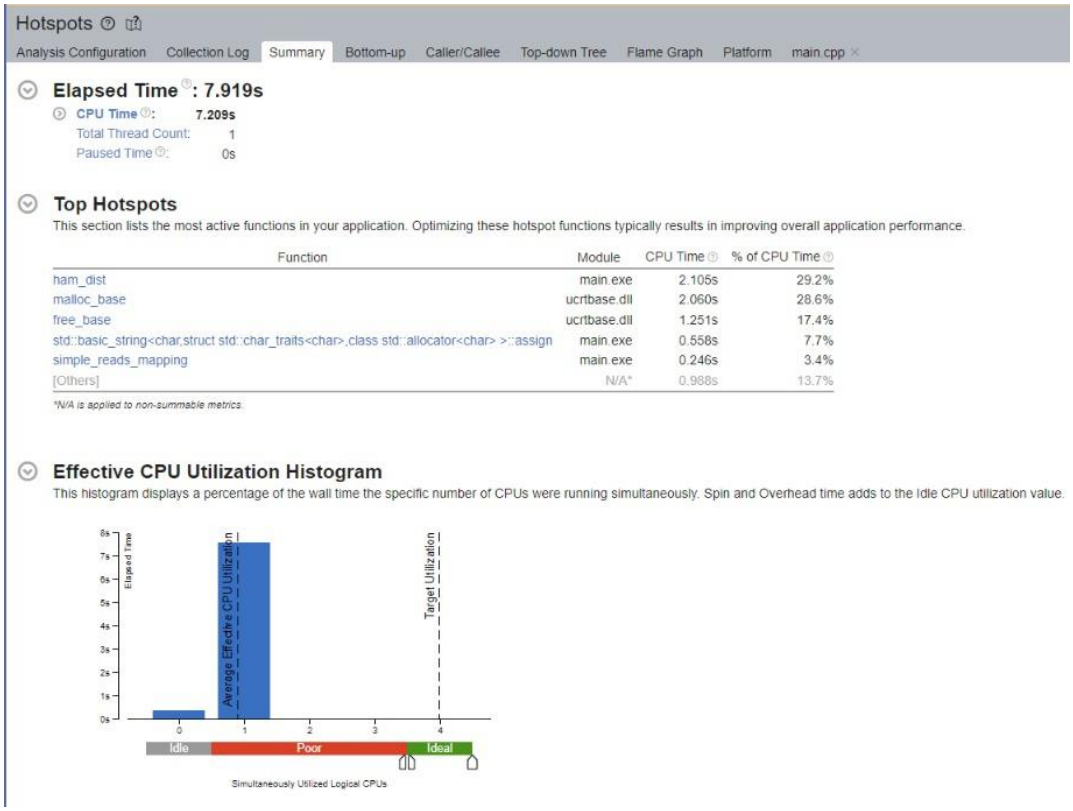

Рис. 1.Результаты hotspot анализа исходной реализации.

Hotspot анализ показал, что самой затратной функцией, с точки зрения использования процессорного времени, является **ham_dist**. На втором и третьем местах идут функции **malloc** и **free,** которые вызываются в процессе создания и уничтожения копий подстроки в методе **string::substr**.

В исходной реализации используется структура данных из стандартной бибилиотеки C++: **std::string**. Она предоставляет удобный интерфейс для работы со строками, однако ее использование сопровождается некоторыми накладными расходами. Один из возможных путей оптимизации — использовать в этой части программы C-строки в формате **char***.

3.2 Memory Access анализ

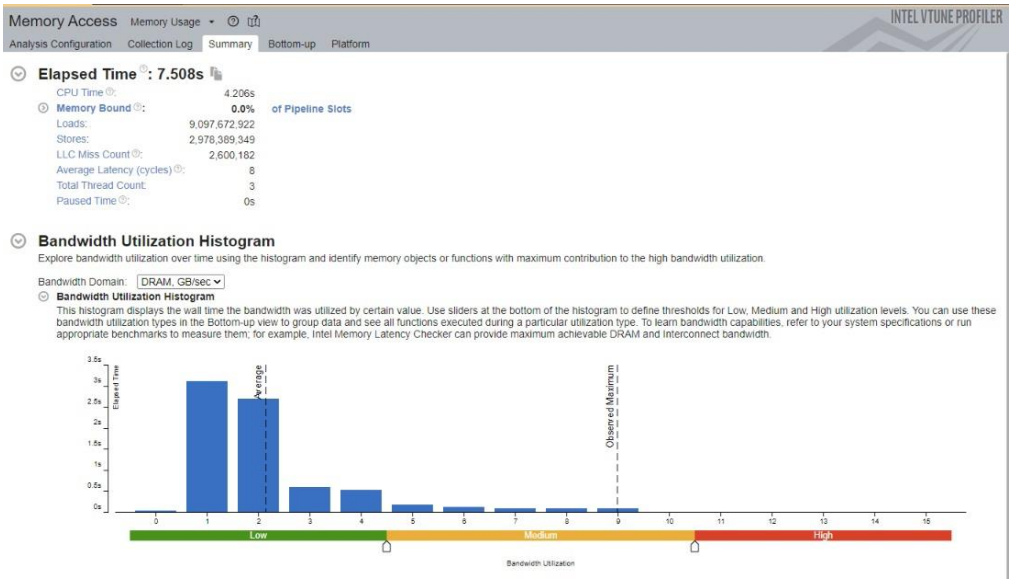

Рис. 2.Результаты memory access анализа исходной реализации.

Программа не сталкивается с ограничениями при работе с памятью. Число промахов кэша умеренно низкое, по сравнению с числом чтения из памяти.

3.3 Threading анализ

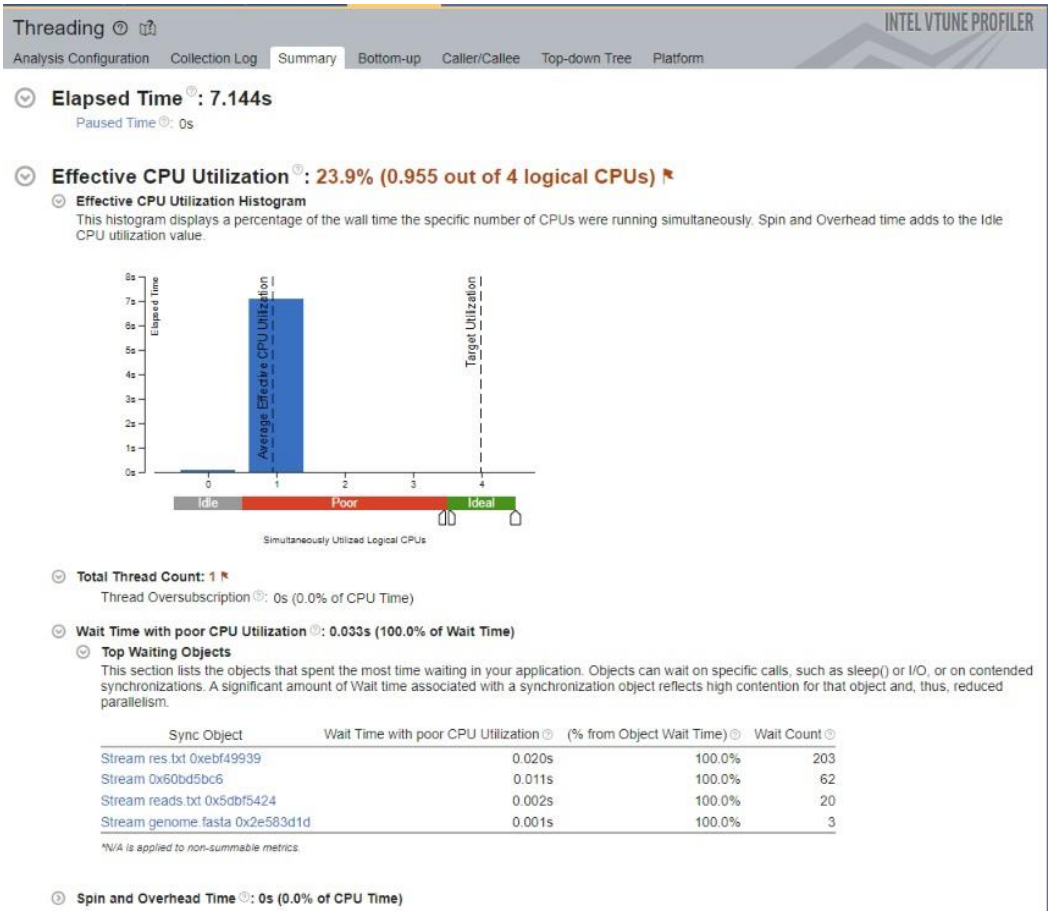

Рис. 3.Результаты threading анализа исходной реализации

Программа не утилизирует все доступные ядра процессора и выполняется последовательно.

4 ОПТИМИЗАЦИЯ

4.1 Первоначальная оптимизация: использование char* вместо std::string

После замены затратных операций из **std::string** (выделение подстроки из строки (**string::substr**) и вызова оператора доступа к элементу строки (**string::operator[]**)) на работу со строками в формате **char***, общее время работы программы уменьшилось с 7 до 2 секунд, т.е получили ускорение в 3.5 раза. Время выполнение функции **ham_dist** уменьшилось почти в 2 раза (с 2.105 с. до 1.123 с.), однака она по-прежнему остается самой вычислительно затратной.

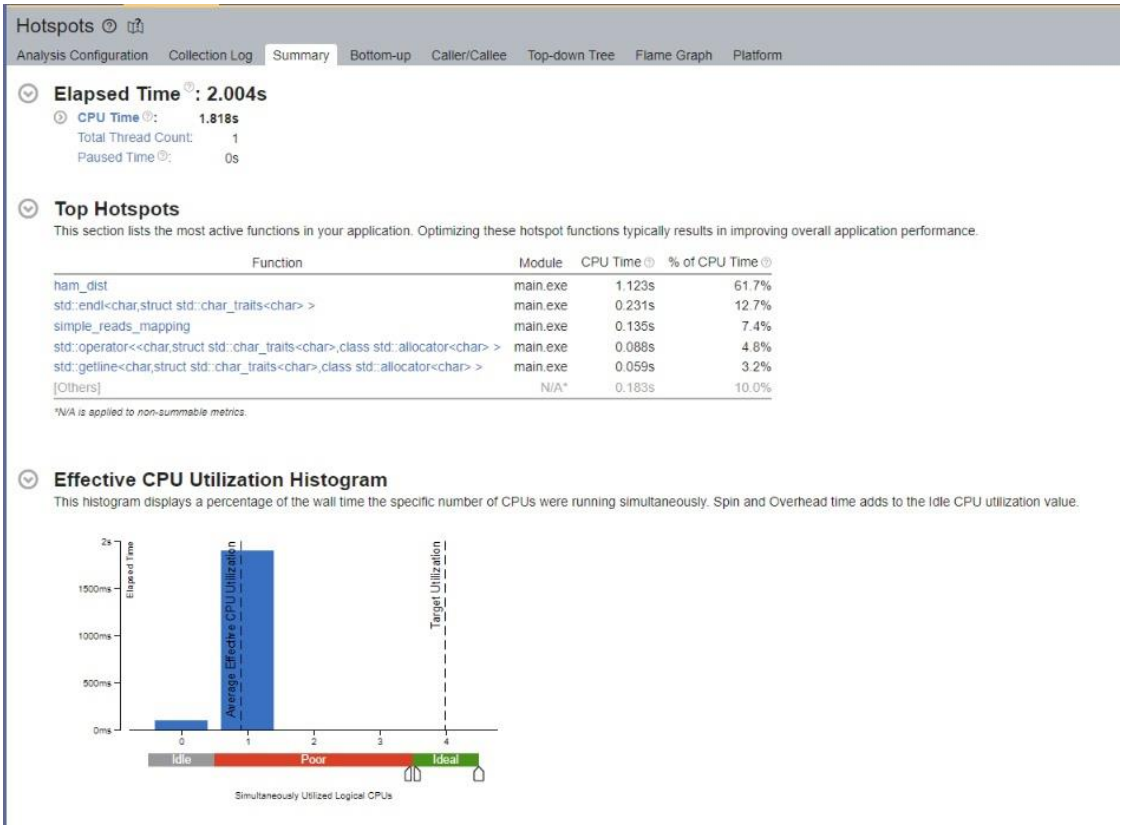

Рис. 4.Результаты hotspot анализа после замены **std::string** на **char*** в функции **ham_dist**.

Также анализ показал, что на вызов **std::endl**, используемой в записи результатов, тратится достаточно много времени. Заменим **std::endl** на символ пересноса строки ‘**\n**’.

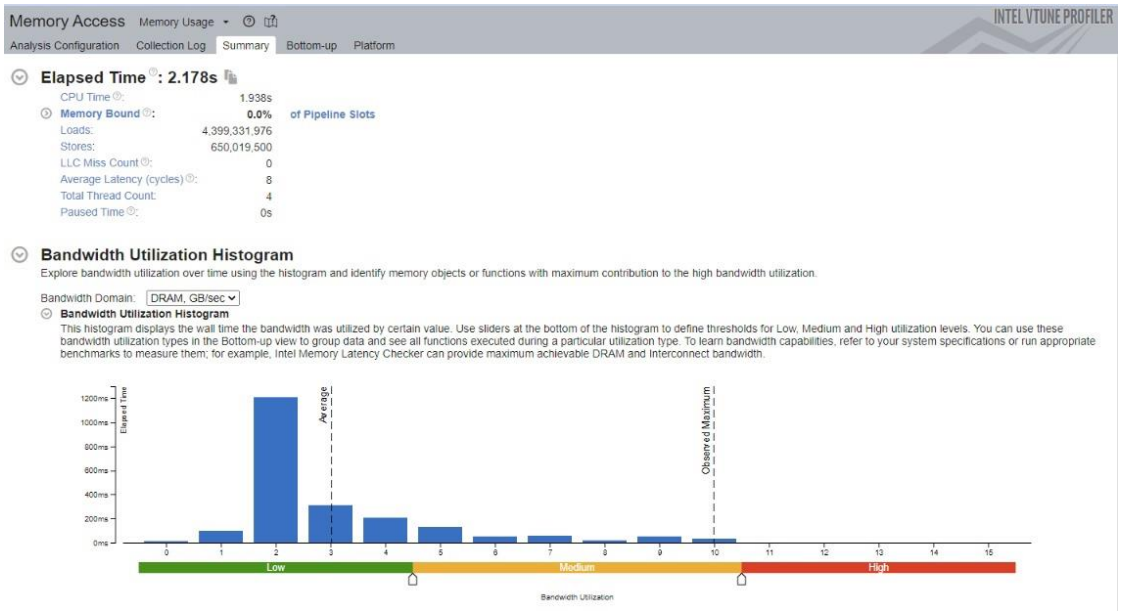

Рис. 5.Результаты memory access анализа исходной реализации

Обаращаясь повторно к анализу доступа к памяти, видим что уменьшилось общее число обращений к памяти (чтений-записи), а также удалось совсем избавиться от промахов кэша последнего уровня.

4.2 Использование оптимизирующего компилятора

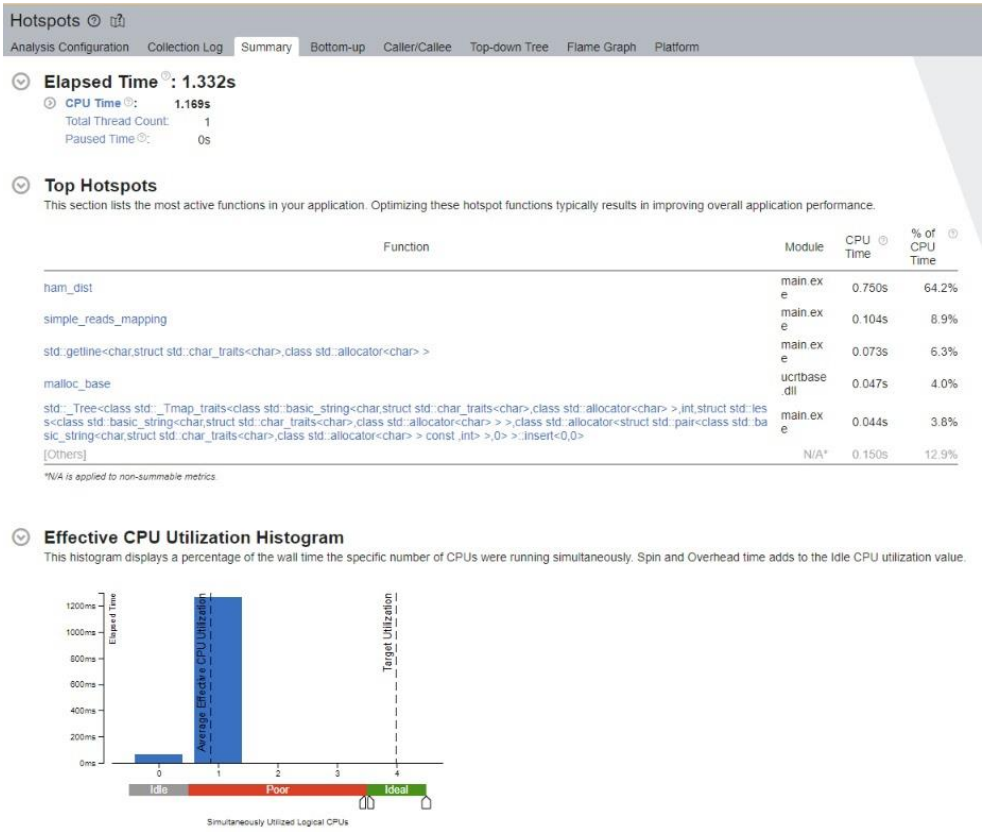

Рис. 6.Результаты hotspot анализа после перехода на оптимизирующий компилятор Intel C++ Compiler 2022.

После замены **std::endl** на ‘**\n**’ в записи результатов и перехода на оптимизирующий компилятор Intel, в релизной конфигурации включилась автовекторизация кода (функция **ham_dist** с использованием **char*** может быть векторизована). Получили общее ускорение работы программы с 2.004 с. до 1.332 с. Время выполнение функции **ham_dist** вследствие автовекторизации уменьшилось с 1.123 с до 0.75 с.

4.3 Параллельная реализация

Пытаться распараллеливать цикл внутри функции **ham_dist** смысла нет, посколькку риды это короткие последовательности нуклеотидов, и мы лишь получим рост накладных расходов от параллельного исполнения, вместо прироста производительности. С оптимизацией данной функции отлично справилась автовекторизация.

Характер алгоритма в целом таков, что он плохо поддается распараллеливанию в его изначальном виде, поскольку мы заранее не знаем в какой части генома могут располагаться риды и нельзя разделить референсный геном на эти части и отдать на обработку каждому отдельному потоку.

Если же попытаться распараллелить цикл по обработке ридов в фунции **simple_reads_mapping** с помощью директивы **#pragma omp parallel for** и провести threading анализ**:**

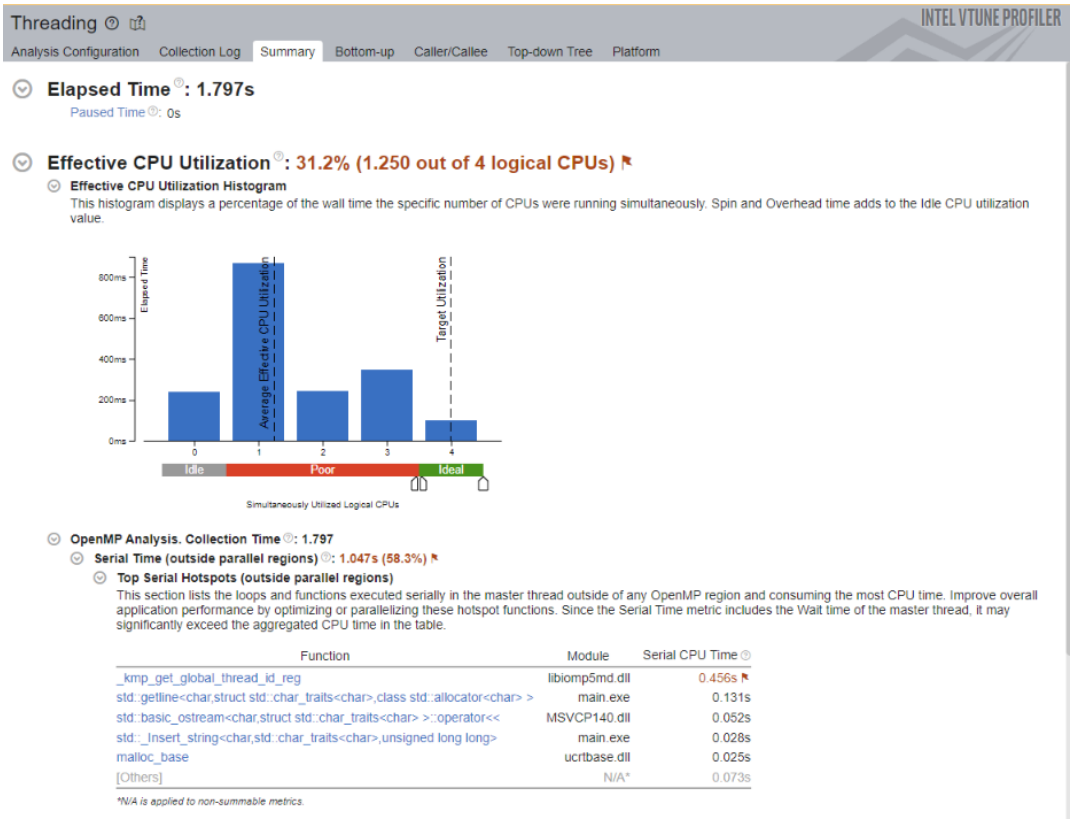

Рис. 7.Результаты threading анализа при попытке распараллелить исходный алгоритм.

То получим ухудшение производительности от накладных расходов на вызов функции **_kmp_get_global_thread_id_reg.** Без значительной переработки исходного алгоритма и применения специальных схем распараллеливания, получить прирост производительности не получилось.

4.4 Применение дополнительных упрощений

В данной задаче, с учетом условия поиска идеальных совпадений, можно пойти на одно упрощение — заменить **ham_dist** на функцию **std::strncmp**, используемую для быстрого сравнения С-строк. Однако в таком случае становится невозможным расширить алгоритм на поиск наиболее близких подстрок, что не подходит для практического применения.

Если в полученном результате не важна упорядоченность данных и доступ за **log(n)**, вместо **std::map**, возможно использование **std::vector** для сохранения найденных позиций ридов в геноме. Но, поскольку на данных небольшого размера, это не является узким местом, значительного прироста производительсноти от такой оптимизации в данный момент не получить.

После всех выше описанных упрощений, удалось снизить общее время выполнения программы до 0.871 с:

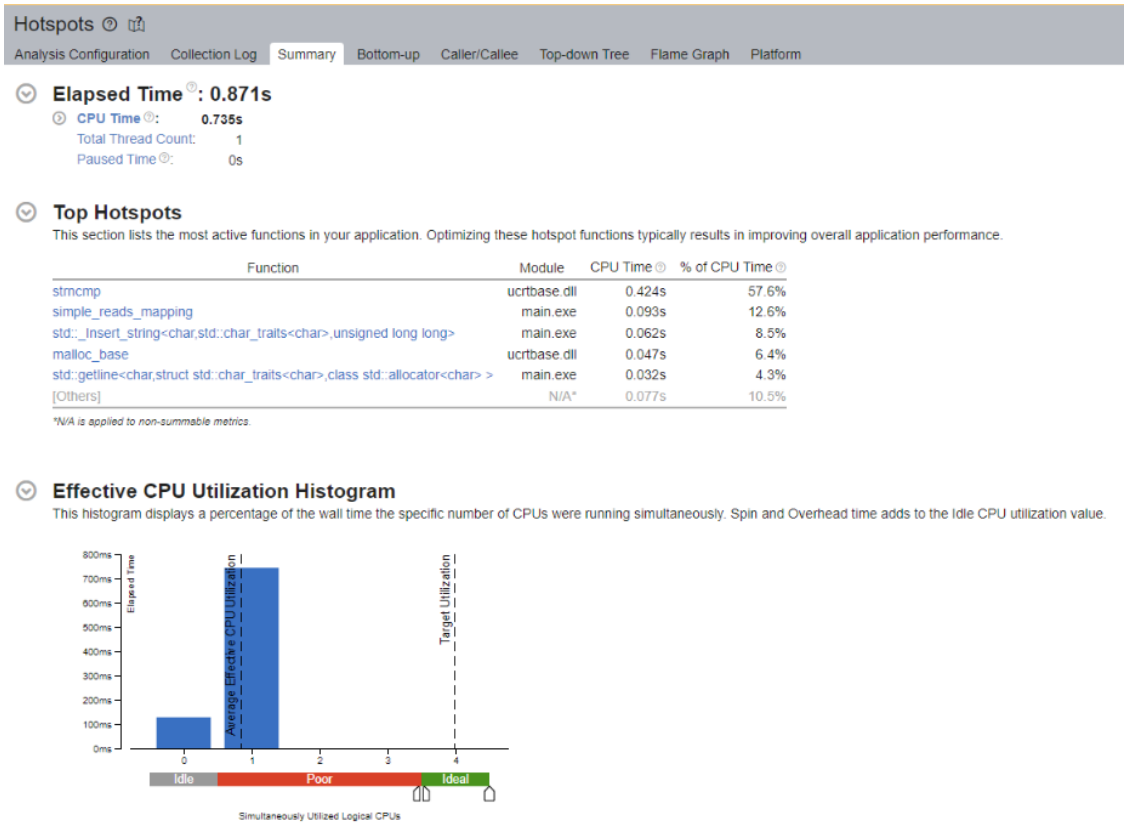

Рис. 8.Результаты hotspot анализа после замены **ham_dist** на функцию из стандартной библиотеки **std::strncmp,** и **std::map** на **std::vector.**

ЗАКЛЮЧЕНИЕ

Изначально данный алгоритм был реализован на языке Python, со стандартными структурами данных (встроенными типами данных **list** и **str**), где на тех же самых тестовых файлах он работал около 2.5 минут. Переписав программу на язык C++, уже был получен значительный прирост производительности, в первоначальной реализации программа уже работала 7.9 с, а применив все описанные оптимизации и упрощения, удалось ускорить общее время выполнения программы до 0.8 с.

Программу также можно оптимизировать в части чтения файлов, сконвертировав все входные данные в бинарный формат.

ПРИЛОЖЕНИЯ

Исходный код доступен по ссылке: https://github.com/ivanvikhrev/software-optimization-course

Исходная реализация, unopt_reads_mapping.cpp

```
    #include <fstream>
    #include <iostream>
    #include <sstream>

    #include <map>
    #include <string>
    #include <vector>

    size_t ham_dist(const std::string& s1, const std::string& s2) {
        if (s1.size() != s2.size()) {
            throw std::logic_error("Hamming distance assumes that string
lengths are equal!");
        }

        size_t dist = 0;
        for (size_t i = 0; i < s1.size(); ++i) {
            if (s1[i] != s2[i]) {
                ++dist;
            }
        }

        return dist;
    }

    std::map<std::string, std::vector<size_t>> build_kmers_map(const
std::string& genome, size_t kmer_len) {
        std::map<std::string, std::vector<size_t>> kmers;
        size_t pos = 0;
        for (size_t i = 0; i < genome.size() - kmer_len + 1; ++i) {
            std::string kmer = genome.substr(i, kmer_len);
            if (kmers.find(kmer) == kmers.end()) {
                kmers.insert({kmer, {i}});
            } else {
                kmers[kmer].push_back(i);
            }
        }

        return kmers;
    }

    std::map<std::string, int> simple_reads_mapping(const std::string&
genome, const std::vector<std::string>& reads, size_t kmer_len) {
        std::map<std::string, int> mapped_reads;
        std::map<std::string, std::vector<size_t>> kmers_map =
build_kmers_map(genome, kmer_len);
        for (const auto& read : reads) {
            std::string seed = read.substr(0, kmer_len);
            bool found = false;
            auto seed_it = kmers_map.find(seed);
```

```
            std::vector<size_t> poss = {};

            if (seed_it != kmers_map.end()) {
                poss = seed_it->second;
            }

            for (size_t pos : poss) {
                if (pos + read.size() < genome.size()) {
                    size_t d = ham_dist(read, genome.substr(pos,
read.size()));
                    if (d == 0) {
                        mapped_reads.insert({read, static_cast<int>(pos)});
                        found = true;
                    }
                }
            }

            if (!found) {
                mapped_reads.insert({ read, -1 }); // no perfect match for
read in genome
            }
        }

        return mapped_reads;
    }

    int main() {
        std::ifstream genome_file;
        std::ifstream reads_file;

        //std::string genome_file_name = "test_example/genome.fasta";
        std::string genome_file_name = "genome.fasta";
        genome_file.open(genome_file_name );
        if (!genome_file) {
            throw std::runtime_error("Unable to open file " +
genome_file_name);
        }

        //std::string reads_file_name = "test_example/reads.txt";
        std::string reads_file_name = "reads.txt";
        reads_file.open(reads_file_name );
        if (!reads_file) {
            throw std::runtime_error("Unable to open file " +
reads_file_name);
        }

        std::string line;
        std::string genome = "";
        std::getline(genome_file, line); // skip header
        while(std::getline (genome_file, line)) {
          genome += line;
        }
        genome_file.close();

        std::vector<std::string> reads;
        while (std::getline(reads_file, line)) {
          reads.push_back(line);
        }
        reads_file.close();
        auto mapped_reads = simple_reads_mapping(genome, reads, 3);

        std::ofstream res("res.txt");
        for (const auto& item : mapped_reads) {
```

```
            res << item.first << " " << item.second << std::endl;
        }

        res.close();
    }
```

После оптимизации, opt_reads_mapping.cpp

```
    #include <fstream>
    #include <iostream>
    #include <sstream>

    #include <map>
    #include <string>
    #include <vector>
    #include <omp.h>

    size_t ham_dist(const char* s1, const char* s2, size_t len) {
        size_t dist = 0;
        for (size_t i = 0; i < len; ++i) {
            dist += (s1[i] != s2[i]);
        }

        return dist;
    }

    std::map<std::string, std::vector<size_t>> build_kmers_map(const
std::string& genome, size_t kmer_len) {
        std::map<std::string, std::vector<size_t>> kmers;
        for (size_t i = 0; i < genome.size() - kmer_len + 1; ++i) {
            std::string kmer = genome.substr(i, kmer_len);
            if (kmers.find(kmer) == kmers.end()) {
                kmers.insert({kmer, {i}});
            } else {
                kmers[kmer].push_back(i);
            }
        }

        return kmers;
    }

    std::vector<int> simple_reads_mapping(const std::string& genome, const
std::vector<std::string>& reads, size_t kmer_len) {
        std::vector<int> mapped_reads(reads.size());
        std::map<std::string, std::vector<size_t>> kmers_map =
build_kmers_map(genome, kmer_len);
        auto genome_cstr = genome.c_str();

        size_t count = 0;
        for (const auto& read : reads) {
            bool found = false;

            std::string seed = read.substr(0, kmer_len);
            auto seed_it = kmers_map.find(seed);
            std::vector<size_t> poss = {};

            if (seed_it != kmers_map.end()) {
                poss = seed_it->second;
            }

            for (size_t pos : poss) {
                if (pos + read.size() < genome.size()) {
```

```
                    if (std::strncmp(read.c_str(), genome_cstr + pos,
read.size()) == 0) {
                        mapped_reads[count] = int(pos);
                        found = true;
                    }
                }
            }

            if (!found) {
                mapped_reads[count] = -1; // no perfect match for read in
genome
            }
            ++count;
        }

        return mapped_reads;
    }

    int main() {
        std::ifstream genome_file;
        std::ifstream reads_file;

        //std::string genome_file_name = "test_example/genome.fasta";
        std::string genome_file_name = "genome.fasta";
        genome_file.open(genome_file_name );
        if (!genome_file) {
            throw std::runtime_error("Unable to open file " +
genome_file_name);
        }

        //std::string reads_file_name = "test_example/reads.txt";
        std::string reads_file_name = "reads.txt";
        reads_file.open(reads_file_name );
        if (!reads_file) {
            throw std::runtime_error("Unable to open file " +
reads_file_name);
        }

        std::string line;
        std::string genome = "";
        std::getline(genome_file, line); // skip header
        while(std::getline (genome_file, line)) {
          genome += line;
        }
        genome_file.close();

        std::vector<std::string> reads;
        while (std::getline(reads_file, line)) {
          reads.push_back(line);
        }
        reads_file.close();
        auto mapped_reads = simple_reads_mapping(genome, reads, 3);


        std::ofstream res("res.txt");
        for (size_t i = 0; i < mapped_reads.size(); ++i) {
            res << reads[i] << " " << mapped_reads[i] << '\n';
        }

        res.close();

        return 0;
    }
```

Код на python:

```
from timeit import default_timer as timer
from datetime import timedelta


def hamming_distance(s1, s2):
    dist = max(len(s1), len(s2))
    if len(s1) == len(s2):
        dist = 0
        for i in range(len(s1)):
            if s1[i] != s2[i]:
                dist += 1
    return dist


def build_kmers_map(genome, k):
    hash_table = {}
    for j in range(len(genome)-k+1):
        k_mer = genome[j:j+k]
           if k_mer in hash_table:
               hash_table[k_mer].append(j)
           else:
              hash_table[k_mer] = [j]
    return hash_table


def simple_reads_mapping(genome, reads, k):
    hash_table = build_kmers_map(genome, k)
    mapped_reads = {}
    count = 0
    print(len(reads))
    for read in reads:
        if (count % 10000 == 0):
            print(f"{count} reads were processed")
        seed = read[0:k]
        possible_poss = []
        if seed in hash_table:
            possible_poss = hash_table[seed]
        for pos in possible_poss:
            d = hamming_distance(read, genome[pos:pos+len(read)])
            if d == 0:
                mapped_reads[read] = pos
                break
        else:
            mapped_reads[read] = -1
        count += 1

    return mapped_reads


fasta_file = "genome.fasta"
genome = ""
with open(fasta_file) as file:
    for line in file:
        line = line.rstrip()
        if line[0] != ">":
            genome += line

reads_file = "reads.txt"
```

```
reads_lst = []
with open(reads_file) as reads:
    for read in reads:
        reads_lst.append(read.rstrip())

k = int(input())
start = timer()
mapped_reads = simple_reads_mapping(genome, reads_lst, k)
end = timer()
print("total time: ", timedelta(seconds=end-start))
```