Оптимизация алгоритма линейной регрессии

Лабораторная работа № 5

Выполнил

студент гр. 5130904/40003 Д. Р. Никищенко

Руководитель-преподаватель а.в. петров

**Содержание**

[**Цель работы:** 2](#_Toc184311868)

[**Задачи работы:** 2](#_Toc184311869)

[**Ход работы** 2](#_Toc184311870)

[Шаг 1 2](#_Toc184311871)

[Шаг 2 3](#_Toc184311872)

[Шаг 3 5](#_Toc184311873)

[Шаг 4 6](#_Toc184311874)

[Шаг 5 8](#_Toc184311875)

[Шаг 6 8](#_Toc184311876)

[**Вывод** 10](#_Toc184311877)

# **Цель работы:**

Перенести под целевую микроархитектуру ПО с оптимальными опциями оптимизации.

# **Задачи работы:**

1) Разработать требования к целевой и инструментальной платформе. Определить техническое задание.

2) Разработать ПО, которая выполняет поставленную задачу, на языке программирования C++14.

3) Написать скрипт, запускающий ПО с разными опциями оптимизации, оценить их влияние на данное ПО.

4) Оценить влияние системных методов оптимизации на эффективность разработанного приложения.

5) Оценить влияние векторизации на эффективность разработанного приложения.

6) Оценить влияние выравнивания адресов на эффективность разработанного приложения.

# **Ход работы**

## Шаг 1

В качестве инструментальной платформы выбрана ОС Ubuntu 22.04 (64-bit). Целевые платформы – Unix-подобные операционные системы. При выполнении работы использовалась библиотека для работы с векторами и матрицами – Eigen.

Техническое задание (далее – ТЗ) – написать алгоритм вычисления коэффициентов линейной регрессии аналитическим методом через OLS-оценку. Формула:

*где – вектор вычисляемых коэффициентов линейной регрессии, A – матрица наблюдений (признаков), – вектор целевого признака (переменной)*

Матрицу признаков и вектор целевой переменной заполним циклично, числа (данные) сгенерируем с помощью метода языка C++ std::rand().

## Шаг 2

Разработано ПО, выполняющее ТЗ.

Код ПО:

*﻿ #include <iostream>*

*#include <Eigen/Dense>*

*const size\_t ROWS = 3000000;*

*const size\_t FEATURES = 30;*

*using namespace Eigen;*

*void linear\_regression(const MatrixXd& X, const VectorXd& y, VectorXd& w) {*

*w = (X.transpose() \* X).inverse() \* X.transpose() \* y;*

*}*

*int main() {*

*MatrixXd X(ROWS, FEATURES);*

*for (int i=0; i < ROWS; ++i) {*

*for (int j=0; j < FEATURES; ++j) {*

*X(i, j) = static\_cast<double>(std::rand()\*100 / RAND\_MAX;*

*}*

*}*

*VectorXd y(ROWS);*

*for (int i = 0; i < ROWS; ++i) {*

*y(i) = 1 + 10/(static\_cast<double>(std::rand()) / RAND\_MAX + 9);*

*}*

*VectorXd w(10);*

*linear\_regression(X, y, w);*

*return 0;*

*}*

Рассмотрим время выполнения кода без оптимизации и размер исполняемого кода:

Примечание: для корректного подключения библиотеки Eigen приходится указывать его путь через ключ -I

*user@user:~$ g++ -I/usr/include/eigen3 mse.cpp -o mse*

*user@user:~$ time ./mse*

*real 0m33.636s*

*user 0m33.221s*

*sys 0m0.372s*

*user@user:~$ du -b mse*

*741152*

*user@user:~$ wc -l mse.cpp*

*48 mse.cpp*

Эффективность кода вычисляется по формуле:

P = ,

где P – эффективность;

Perf – обратное время работы кода, ;

Comp – размер кода (в байтах).

## Шаг 3

В файл *test\_opt.sh* записан скрипт, запускающий ПО с указанными методами оптимизации:

*#!/bin/bash*

*FILE="main.cpp"*

*OPTIONS=("-O0" "-Os" "-O1" "-O2" "-O3" "-O2 -march=native" "-O3 -march=native" "-O2 -march=native -funroll-loops" "-O3 -march=native -funroll-loops")*

*for opt in "${OPTIONS[@]}"; do*

*echo "Building with $opt"*

*g++ $FILE -o linear\_regression $opt*

*size=$(du -b linear\_regression | cut -f1)*

*time=$( { /usr/bin/time -f "%e" ./linear\_regression > /dev/null; } 2>&1 )*

*echo "Optimization: $opt | Size: $size bytes | Time: $time s"*

*done*

Запускаем скрипт с помощью команды tee, которая делает вывод в консоль и записывает в файл *opt\_results.txt*:

*user@user:~$ bash test\_opt.sh | tee opt\_results.txt*

Результаты оптимизации ПО:

*Optimization: -O0 | Size: 741152 bytes | Time: 33.59 s*

*Optimization: -Os | Size: 123168 bytes | Time: 3.19 s*

*Optimization: -O1 | Size: 97536 bytes | Time: 1.73 s*

*Optimization: -O2 | Size: 109128 bytes | Time: 1.72 s*

*Optimization: -O3 | Size: 117432 bytes | Time: 1.71 s*

*Optimization: -O2 -march=native | Size: 109288 bytes | Time: 1.58 s*

*Optimization: -O2 -march=native -funroll-loops | Size: 137960 bytes | Time: 1.37 s*

*Optimization: -O3 -march=native | Size: 113496 bytes | Time: 1.40 s*

*Optimization: -O3 -march=native -funroll-loops | Size:146264 | Time: 1.36 s*

По результатам вычисления метрики эффективности P самой эффективной оказалась опция -O3 -march=native (с показателем эффективности 629), которая сократила время с 33.59 секунд до 1.4 секунд, а также уменьшила размер исполняемого кода с 741152 байтов до 113496.

## Шаг 4

Запустим ПО с лучшим методом оптимизации -O3 -march=native и добавим системные методы оптимизации. Используем межпроцедурную оптимизацию (методы -fipa), оптимизацию времени компоновки (-flto), а также профильную оптимизацию (оптимизацию с обратной связью, -fprofile-generate/-fprofile-use)

*user@user:~$ g++ -I/usr/include/eigen3 mse.cpp -o mse\_sys -O3 -march=native -fipa-ra -fipa-cp -fipa-vrp -flto -fpofile-generate*

*user@user:~$ time ./mse\_sys*

*real 0m1.533s*

*user 0m1.235s*

*sys 0m0.296s*

*user@user:~$ du -b mse\_sys*

*437472 mse\_sys*

*user@user:~$ g++ -I/usr/include/eigen3 mse.cpp -o mse\_sys -O3 -march=native -fipa-ra -fipa-cp -fipa-vrp -flto -fpofile-use*

*user@user:~$ time ./mse\_sys*

*real 0m1.300s*

*user 0m0.966s*

*sys 0m0.332s*

*user@user:~$ du -b mse\_sys*

*108120 mse\_sys*

Даже без использования подсчетов видно, что по результатам использования системных методов вместе с лучшим методом оптимизации (-O3 -march=native) время выполнения кода сократилось до 1.3 секунды, а размер до 108120. Показатель эффективности P стал 711.

С этого момента будем считать данный вариант оптимальным.

## Шаг 5

Дополнительно к опциям оптимизации применим векторизацию кода. Несмотря на то, что -O3 уже включает в себя авто-векторизацию, команда выполняет это явно и делает векторизацию более агрессивной.

Время выполнения кода с агрессивной векторизацией и размер кода:

*user@user:~$ g++ -I/usr/include/eigen3 mse.cpp -o mse\_sys\_vec -O3 -march=native -ftree-vectorize -fvect-cost-model=unlimited*

*user@user:~$ time ./mse\_sys\_vec*

*real 0m1.315s*

*user 0m1.033s*

*sys 0m0.280s*

*user@user:~$ du -b mse\_sys\_vec*

*117592 mse\_sys\_vec*

По результатам использования агрессивной векторизации видно, что время и размер файла увеличились по сравнению с оптимальным вариантом.

## Шаг 6

Попробуем применить выравнивание адресов дополнительно к векторизации. Возможно, это улучшит результат, полученный на прошлом шаге.

Чтобы определить оптимальную ширину для выравнивания, воспользуемся командой:

*user@user:~$ cat /proc/cpuinfo | grep flags | grep avx*

Среди флагов обнаружились avx и avx2. Значит, при выравнивании адресов данных, связанных с циклами, функциями используем ширину регистров AVX2, то есть 32 байта:

*user@user:~$ g++ -I/usr/include/eigen3 mse.cpp -o mse\_sys\_vec -O3 -march=native -ftree-vectorize -falign-loops=32 -falign-functions=32 -falign-jumps=32 -falign-labels=32*

*user@user:~$ time ./mse\_sys\_vec*

*real 0m1.323s*

*user 0m1.025s*

*sys 0m0.296s*

*user@user:~$ du -b mse\_sys\_vec*

*129880 mse\_sys\_vec*

По результатам видно, что показатели времени выполнения кода и его размера исполняемого файла еще больше увеличились по сравнению с векторизацией.

Возможно, причина в излишнем выравнивании: в коде используется мало меток и функций, а вот циклов – много. Попробуем оставить выравнивание только по циклам.

*user@user:~$ g++ -I/usr/include/eigen3 mse.cpp -o mse\_sys\_vec -O3 -march=native -ftree-vectorize -falign-loops=32*

*user@user:~$ time ./mse\_sys\_vec*

*real 0m1.309s*

*user 0m1.067s*

*sys 0m0.240s*

*user@user:~$ du -b mse\_sys\_vec*

*117592 mse\_sys\_vec*

Как видим, теперь время выполнения кода незначительно уменьшилось по сравнению с векторизацией, а размер остался прежний.

# **Вывод**

Результаты работы приведен в таблице и на графиках:

|  | **Оптимизация** | **Время (с)** | **Размер (байт)** | **Эффективность** |
| --- | --- | --- | --- | --- |
| 0 | -O0 | 33.59 | 741152 | 4.02 |
| 1 | -Os | 3.19 | 123168 | 254.51 |
| 2 | -O1 | 1.73 | 97536 | 592.64 |
| 3 | -O2 | 1.72 | 109128 | 532.76 |
| 4 | -O3 | 1.71 | 117432 | 497.99 |
| 5 | -O2 -march=native | 1.58 | 109288 | 579.12 |
| 6 | -O2 -march=native -funroll-loops | 1.37 | 137960 | 529.09 |
| 7 | -O3 -march=native | 1.40 | 113496 | 629.35 |
| 8 | -O3 -march=native -funroll-loops | 1.36 | 146264 | 502.72 |
| 9 | -O3 -march-native + sys | 1.30 | 108120 | 711.46 |
| 10 | -O3 -march=native + vector | 1.32 | 117592 | 644.24 |
| 11 | -O3 -march=native + vectorize + align | 1.31 | 117592 | 649.16 |

*Таблица 1 Результаты оптимизаций*

*Примечание: для улучшения восприятия, показатель времени был прологарифмирован.*

*Изображение выглядит как текст, снимок экрана

Автоматически созданное описание*

*Рис. 1 Диаграмма зависимости времени выполнения кода от метода оптимизации*

Изображение выглядит как текст, снимок экрана, Прямоугольник, Параллельный

Автоматически созданное описание

*Рис. 2 Диаграмма зависимости размера исполняемого файла от метода оптимизации*

Изображение выглядит как текст, снимок экрана, диаграмма, Параллельный

Автоматически созданное описание

*Рис. 3 Показатели эффективности методов оптимизаций*

Оптимизация значительно ускорила выполнение программы. Наиболее эффективной оказалась опция -O3 -march=native, которая включает агрессивные оптимизации, такие как разворачивание циклов, а также использует специфичные для архитектуры процессора методы. Кроме того, уменьшение размера программы дополнительно повысило её производительность.

Системные методы оптимизации позволили добиться ещё большего ускорения работы программы и сокращения размера исполняемого файла, что сделало их самыми подходящими для моего приложения.

Векторизация показала себя полезной для моего случая, так как приложение выполняет множество вычислений с большим объёмом данных. Однако авто-векторизации, предоставляемой опцией -O3, оказалось достаточно. Использование явной агрессивной векторизации лишь увеличивало размер кода без заметного выигрыша в скорости.

Выравнивание адресов не дало существенного улучшения. Хотя время выполнения кода уменьшилось, увеличение размера исполняемого файла привело к потенциальным потерям в производительности.