МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики
Кафедра Математического обеспечения и суперкомпьютерных технологий

ОТЧЁТ
по предмету
“Анализ производительности и оптимизация программного обеспечения”

Выполнил:
студент группы 3824М1ПР1
Булгаков Д.Э.
Подпись

Проверил:
к.т.н., доц.
И.Б. Мееров
Подпись

Нижний Новгород
2024

1. Введение 3

2. Описание алгоритма 4

2.1 Инициализация 4

2.2 Итерационный процесс 4

2.3 Проверка сходимости 4

2.4 Преимущества метода 5

3. Способы ускорения 6

3.1 Использование технологии OpenMP 6

3.2 Распараллеливание независимых секций 6

3.3 Оптимизация численных операций 6

3.4 Эффективная работа с памятью 7

4. Результаты 8

4.1 Intel Advisor 8

4.2 AMDuProf 9

5. Заключение 11

6. Приложение 12

6.1 Оригинальный метод 12

6.2 Оптимизированный метод 16

1. Введение

Анализ производительности и оптимизация программного обеспечения (ПО) являются важными аспектами разработки масштабируемых и эффективных решений. Выявление узких мест, создание эффективного кода и использование современных инструментов способствуют значительному улучшению качества ПО и удовлетворению потребностей конечных пользователей.

В данном отчете рассматривается задача решения систем линейных алгебраических уравнений (СЛАУ) с использованием метода коньюгированных градиентов. Данная задача имеет высокую востребованность в численных методах и широко используется в различных областях науки и инженерии.

Цель работы — анализ различных методов оптимизации на основе готового решения из открытого репозитория на GitHub и оценка их влияния на производительность алгоритма.

2. Описание алгоритма

Метод сопряженных градиентов (Conjugate Gradient Method) — это численный алгоритм, предназначенный для решения систем линейных алгебраических уравнений вида

$$A \cdot x = b,$$

где A является симметричной положительно определённой матрицей. Этот метод является итерационным, что делает его эффективным для решения задач с большой размерностью, особенно когда матрица A разрежена.

Алгоритм включает следующие основные шаги:

2.1. Инициализация

Задаются начальное приближение x_0, начальный вектор остатка $r_0 = b - A \cdot x_0$ и вспомогательный вектор $p_0 = r_0$.

2.2. Итерационный процесс

На каждом шаге итерации вычисляются:

- $\alpha_k = \frac{r_k^T \cdot r_k}{p_k^T \cdot A \cdot p_k}$ — коэффициент для обновления решения;
- $x_{k+1} = x_k + \alpha_k \cdot p_k$ — новое приближение решения;
- $r_{k+1} = r_k - \alpha_k \cdot A \cdot p_k$ — новый вектор остатка;
- $\beta_k = \frac{r_{k+1}^T \cdot r_{k+1}}{r_k^T \cdot r_k}$ — коэффициент для обновления направления поиска;
- $p_{k+1} = r_{k+1} + \beta_k \cdot p_k$ — новое направление поиска.

2.3. Проверка сходимости

Итерации продолжаются до тех пор, пока норма вектора остатка $\|r_k\|$ не станет меньше заданного порога (показателя точности).

2.4. Преимущества метода

Метод сопряженных градиентов обладает следующими преимуществами:

- Экономия памяти, так как он не требует хранения всей матрицы A, а только её умножения на вектор;
- Быстрая сходимость для хорошо обусловленных систем.

Однако для достижения высокой производительности при реализации алгоритма важно учитывать аспекты, связанные с доступом к памяти, балансировкой вычислений и возможностями параллельной обработки. В следующем разделе будут рассмотрены методы ускорения алгоритма, включая применение технологий параллелизма.

3. Способы ускорения

Для повышения производительности алгоритма решения системы линейных уравнений методом сопряженных градиентов были использованы следующие подходы:

3.1. Использование технологии OpenMP

Для распараллеливания вычислений в коде применяется технология OpenMP, которая позволяет задействовать многопоточность. Это обеспечивает ускорение выполнения следующих операций:

- Векторно-матричное произведение (`omp_matrix_vec`) распараллеливается по строкам матрицы с использованием директивы `#pragma omp parallel for`.
- Скалярное произведение векторов (`omp_vec_vec`) выполняется с применением директивы `#pragma omp parallel for reduction(+ : res)` для обеспечения корректного суммирования результатов из разных потоков.

3.2. Распараллеливание независимых секций

Для повышения эффективности в цикле алгоритма сопряженных градиентов используется директива `#pragma omp parallel sections`, которая позволяет выполнять независимые вычисления, такие как обновление решения (x) и остатка (r), параллельно.

3.3. Оптимизация численных операций

Для предотвращения деления на ноль и улучшения численной устойчивости введено использование параметра `SMOL`, минимального значения, ниже которого дробь обнуляется:

$$d = \frac{r^T r}{\max(p^T (A \cdot p), \texttt{SMOL})}.$$

Это позволяет избежать ошибок вычислений при работе с малыми числами.

3.4. Эффективная работа с памятью

- Матрица и векторы передаются по ссылке, что минимизирует копирование данных.
- Векторы обновляются на месте, что уменьшает количество выделений и освобождений памяти.

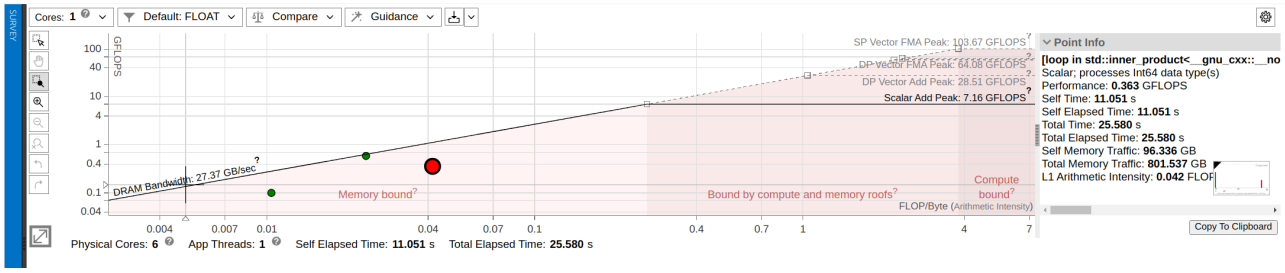
4. Результаты

Испытания проведены на матрице размера 1000×1000 и векторе размера 1000.

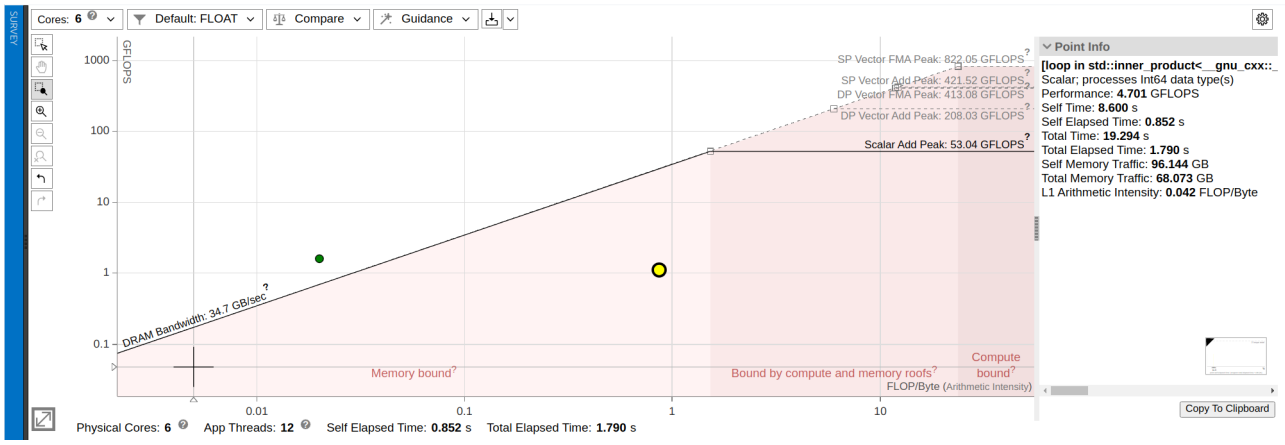
4.1. Intel Advisor

Первым инструментом для сравнения производительности был Intel Advisor. На нем получились следующие результаты:

Оригинальная версия



Оптимизированная версия



На основе этих данных можно сделать следующие выводы:

1. Производительность:
   - Оригинальная версия: 0.363 GFLOPS.
   - Оптимизированная версия: 4.701 GFLOPS.

Ускорение составляет примерно 13 раз.

2. Время выполнения:

   - Оригинальная версия: общее время выполнения 25.580 секунд.
   - Оптимизированная версия: общее время выполнения 1.790 секунд.

   Ускорение примерно в 14 раз.

3. Затраты времени на ключевые операции (Self Time):

   - Оригинальная версия: 11.051 секунд.
   - Оптимизированная версия: 0.852 секунды.

   Значительное сокращение времени выполнения.

4. Объём операций с памятью (Memory Traffic):

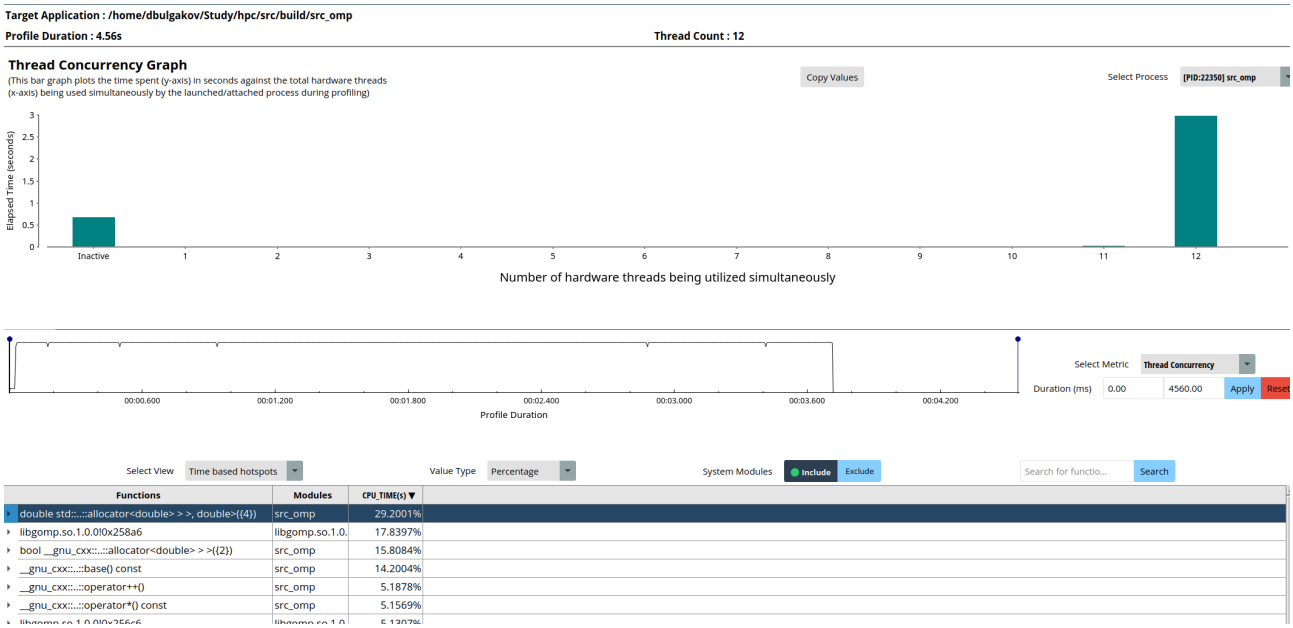
   - Оригинальная версия: общий трафик памяти 801.537 GB.
   - Оптимизированная версия: общий трафик памяти 68.073 GB.

   Оптимизация позволила уменьшить объём операций с памятью приблизительно в 11 раз.

4.2. AMDuProf

Дополнительным инструментом был использован AMDuProf

Оригинальная версия



Оптимизированная версия

На основе метрик и графиков собранных с данном профилировщика, можно сделать следующие выводы:

Оригинальная версия

- Общее количество активных потоков: 1 (Thread Count = 1). Все вычисления выполняются последовательно.
- Время выполнения составляет 6.6385 секунд, что отражает работу одного потока без использования параллелизма.

Оптимизированная версия

- Общее количество потоков, участвующих в вычислениях, увеличилось до 12.
- Общее время выполнения алгоритма существенно снизилось: главный поток выполняет работу за 2.9801 секунды (из общего времени), что демонстрирует успешное использование параллелизма.

5. Заключение

В данной работе был рассмотрен метод сопряженных градиентов для решения систем линейных уравнений, а также проведён анализ его производительности и реализация с использованием различных подходов к оптимизации. Основное внимание было уделено использованию технологии OpenMP для распараллеливания вычислений, что позволило эффективно задействовать ресурсы многопроцессорных систем.

Были выделены ключевые аспекты оптимизации:

- Распараллеливание операций векторно-матричного умножения и скалярного произведения.
- Разделение независимых вычислительных секций для ускорения итеративного процесса.
- Предотвращение ошибок численной стабильности с использованием параметра `SMOL`.
- Уменьшение накладных расходов на выделение памяти за счёт обновления данных на месте.

Таким образом, удалось подтвердить значительное ускорение работы параллельной версии по сравнению с последовательной. Полученные результаты демонстрируют эффективность подходов, использованных в реализации.

6. Приложение

6.1. Оригинальный метод

slau_gradient_orig.hh

```
#ifndef SRC_SLAU_GRADIENT_ORIG_HH_
#define SRC_SLAU_GRADIENT_ORIG_HH_

#include <vector>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <random>

using dvec = std::vector<double>;
using dmat = std::vector<dvec>;

/**
 * @brief generate random simmetric positive defined (spd) matrix
 *
 * @param size size of matrix
 * @param seed seed
 * @return matrix (dmat, std::vector<std::vector<double>>)
 */
dmat generateMatrix(int size, unsigned int seed);

/**
 * @brief generate random vector
 *
 * @param size vector size
 * @param seed seed
 * @return vector
 */
dvec generateVector(int size, unsigned int seed);
```

```
/**
 * @brief scalar multiplication of vectors
 *
 * @param a first vector
 * @param b second vector
 * @return result value
 */
double vec_vec(const dvec &a, const dvec &b);

/**
 * @brief matrix-vector multiplication
 *
 * @param a matrix (dmat, std::vector<std::vector<double>>)
 * @param b vector
 * @return result vector
 */
dvec matrix_vec(const dmat &a, const dvec &b);

/**
 * @brief linear combination of vectors
 *
 * @param ma scalar value mult for first vector
 * @param a first vector
 * @param mb scalar value mult for second vector
 * @param b second vector
 * @return result vector
 */
dvec vec_vec_comb(double ma, const dvec& a, double mb, const dvec& b);

/**
 * @brief vector normalizing
 *
 * @param a vector
 * @return magnitude
 */
double vec_norm(const dvec& a);
```

```
/**
 * @brief conjagurate gradient method solver
 *
 * @param a matrix (dmat, std::vector<std::vector<double>>)
 * @param b vector
 * @return result
 */
dvec solve(const dmat &a, const dvec& b);

#endif  // SRC_SLAU_GRADIENT_ORIG_HH_
```

slau_gradient_orig.cpp

```
#include "slau_gradient_orig.h"

dmat generateMatrix(int size, unsigned int seed) {
    dmat res = dmat(size, std::vector<double>(size));
    dvec v = generateVector(size, seed);
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            res[i][j] = v[i] * v[j];
        }
        res[i][i] += size;
    }
    return res;
}
dvec generateVector(int size, unsigned int seed) {
    dvec res = dvec(size);
    std::mt19937 mt(seed);
    std::uniform_real_distribution<double> urd(-5, 5);
    for (int i = 0; i < size; i++) {
        res[i] = urd(mt);
    }
    return res;
}
```

```
double vec_vec(const dvec &a, const dvec &b) {
    return std::inner_product(a.begin(), a.end(), b.begin(), 0.0);
}

dvec matrix_vec(const dmat &a, const dvec &b) {
    dvec res(a.size());
    for (int i = 0; i < a.size(); i++) {
        res[i] =  vec_vec(a[i], b);
    }
    return res;
}

dvec vec_vec_comb(double ma, const dvec& a, double mb, const dvec& b) {
    dvec res(a.size());
    for (int i = 0; i < a.size(); i++) {
        res[i] = ma * a[i] + mb * b[i];
    }
    return res;
}

double vec_norm(const dvec& a) {
    return sqrt(vec_vec(a, a));
}


dvec solve(const dmat &a, const dvec& b) {
    dvec res(a.size(), 0.0);

    dvec r(b);
    dvec p(r);

    int k = 0;

    while (k++ < a.size()) {
        dvec r_prev;
```

```
        dvec mv = matrix_vec(a, p);
        r_prev = r;

        double d = vec_vec(r, r) / std::max(vec_vec(p, mv), SMOL);
        res = vec_vec_comb(1.0, res, d, p);
        r = vec_vec_comb(1.0, r, -d, mv);

        if (vec_norm(r) < SMOL) break;

        double s = vec_vec(r, r) / std::max(vec_vec(r_prev, r_prev),
           SMOL);
        p = vec_vec_comb(1.0, r, s, p);
    }
    return res;
}
```

6.2. Оптимизированный метод

slau_gradient.hh

```
// Copyright 2024 Bulgakov Daniil

#ifndef SRC_SLAU_GRADIENT_HH_
#define SRC_SLAU_GRADIENT_HH_

#include <algorithm>
#include <cmath>
#include <iostream>
#include <numeric>
#include <random>
#include <vector>

const double SMOL = 1.0e-10;

using dvec = std::vector<double>;
using dmat = std::vector<dvec>;
```

```
// Sequential Computing Methods

/**
 * @brief scalar multiplication of vectors
 *
 * @param a first vector
 * @param b second vector
 * @return result value
 */
double
vec_vec(const dvec &a, const dvec &b);

/**
 * @brief matrix-vector multiplication
 *
 * @param a matrix (dmat, std::vector<std::vector<double>>)
 * @param b vector
 * @return result vector
 */
dvec
matrix_vec(const dmat &a, const dvec &b);

/**
 * @brief linear combination of vectors
 *
 * @param ma scalar value mult for first vector
 * @param a first vector
 * @param mb scalar value mult for second vector
 * @param b second vector
 * @return result vector
 */
dvec
vec_vec_comb(double ma, const dvec &a, double mb, const dvec &b);

/**
```

```
 * @brief vector normalizing
 *
 * @param a vector
 * @return magnitude
 */
double
vec_norm(const dvec &a);

// Parallel Computing Methods

/**
 * @brief parallel matrix-vector multiplication
 *
 * @param a matrix (dmat, std::vector<std::vector<double>>)
 * @param b vector
 * @return result vector
 */
dvec
omp_matrix_vec(const dmat &a, const dvec &b);

/**
 * @brief parallel scalar multiplication of vectors
 *
 * @param a first vector
 * @param b second vector
 * @return result value
 */
double
omp_vec_vec(const dvec &a, const dvec &b);

// Conjugate Method Algorithm

/**
 * @brief parallel conjugate gradient method solver
 *
 * @param a matrix (dmat, std::vector<std::vector<double>>)
```

```
 * @param b vector
 * @return result
 */
dvec
omp_solve(const dmat &a, const dvec &b);

#endif  // SRC_SLAU_GRADIENT_HH_
```

slau_gradient.cpp

```
// Copyright 2024 Bulgakov Daniil

#include "slau_gradient.hh"

// Sequential Computing Methods ------------------

double
vec_vec(const dvec &a, const dvec &b)
{
    return std::inner_product(a.begin(), a.end(), b.begin(), 0.0);
}

dvec
matrix_vec(const dmat &a, const dvec &b)
{
    dvec res(a.size());
    for (int i = 0; i < a.size(); i++) {
        res[i] = vec_vec(a[i], b);
    }
    return res;
}

dvec
vec_vec_comb(double ma, const dvec &a, double mb, const dvec &b)
{
    dvec res(a.size());
```

```
    for (int i = 0; i < a.size(); i++) {
        res[i] = ma * a[i] + mb * b[i];
    }
    return res;
}

double
vec_norm(const dvec &a)
{
    return sqrt(vec_vec(a, a));
}

// Parallel Computing Methods

dvec
omp_matrix_vec(const dmat &a, const dvec &b)
{
    dvec res(a.size());
#pragma omp parallel for
    for (int i = 0; i < a.size(); i++) {
        res[i] = vec_vec(a[i], b);
    }
    return res;
}

double
omp_vec_vec(const dvec &a, const dvec &b)
{
    double res = 0.0;
#pragma omp parallel for reduction(+ : res)
    for (int i = 0; i < a.size(); i++) {
        res += a[i] * b[i];
    }
    return res;
}
```

```
// Conjugate Method Algorithm

dvec
omp_solve(const dmat &a, const dvec &b)
{
    dvec res(a.size(), 0.0);

    dvec r(b);
    dvec p(r);

    for (int i = 0; i < a.size(); i++) {
        dvec r_prev;
        dvec mv = omp_matrix_vec(a, p);
        r_prev = r;

        double d = omp_vec_vec(r, r) / std::max(omp_vec_vec(p, mv),
           SMOL);

#pragma omp parallel sections
        {
#pragma omp section
            {
                res = vec_vec_comb(1.0, res, d, p);
            }
#pragma omp section
            {
                r = vec_vec_comb(1.0, r, -d, mv);
            }
        }

        if (vec_norm(r) < SMOL)
            break;

        double s = omp_vec_vec(r, r) / std::max(omp_vec_vec(r_prev,
           r_prev), SMOL);
```

```
            p = vec_vec_comb(1.0, r, s, p);
        }

        return res;
    }
```