МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра вычислительной техники

Отчет по лабораторной работе №1

по дисциплине

«Параллельные алгоритмы и системы»

Тема: «Умножение матриц»

Студент гр. 9306 _____ Смирнов А.В.

Преподаватель _____ Пазников А.А.

Санкт-Петербург
2023

СОДЕРЖАНИЕ

Цель

Практическое закрепление понимания общих идей оптимизации алгоритмов.

Постановка задачи

Требуется оптимизировать программу, проанализировать полученные результаты, построить графики. При оптимизации использовать распараллеливание OpenMP. Эксперименты провести на двух компиляторах: Clang, GCC. Для выполнения лабораторной работы был выбран язык программирования C.

Описание алгоритма

Листинг 1 – Код алгоритма без оптимизации

```
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>
#include <limits.h>

#define M_SIZE 512
#define OUT_EN 0
#define RAND_INIT 1
#define BLOCK_S 16

float tdiff(struct timeval *start, struct timeval *end){
    return (end -> tv_sec - start -> tv_sec) + 1e-6*(end -> tv_usec - start ->
tv_usec);
};

int main()
{

    int m1[M_SIZE][M_SIZE] = {{8,4,3,7},{3,2,4,8},{5,7,7,7},{1,8,9,5}};
    int m2[M_SIZE][M_SIZE] = {{6,4,8,9},{2,6,3,2},{0,1,3,7},{8,7,0,2}};
    int res[M_SIZE][M_SIZE] = {0};
    struct timeval start, end;
    srand(time(NULL));

    if(RAND_INIT == 1){
        for(int i = 0; i < M_SIZE; ++i){
            for(int j = 0; j < M_SIZE; ++j){
                m1[i][j] = 0 + rand()%(10);
                m2[i][j] = 0 + rand()%(10);
            }
        }
    }

    if(OUT_EN == 1){
        printf("A:\n");
        for(int i = 0; i < M_SIZE; ++i){
            for(int j = 0; j < M_SIZE; ++j){
               printf("%2d ", m1[i][j]);
            }
            printf("\n");
        }
        printf("\nB:\n");
        for(int i = 0; i < M_SIZE; ++i){
            for(int j = 0; j < M_SIZE; ++j){
               printf("%2d ", m2[i][j]);
            }
            printf("\n");
        }}

    printf("START...\n");
    gettimeofday(&start, NULL);

    for(int i = 0; i < M_SIZE; ++i){
        for(int j = 0; j <M_SIZE; ++j){
            for(int k = 0; k < M_SIZE; k+=4){
                res[i][j] += m1[i][k]*m2[k][j];
            }
        }
    }

    gettimeofday(&end, NULL);
    printf("STOP | execution time: %0.6f\n", tdiff(&start, &end));

    if(OUT_EN == 1){
        printf("\nA*B:\n");
        for(int i = 0; i < M_SIZE; ++i){
            for(int j = 0; j < M_SIZE; ++j){
               printf("%2d ", res[i][j]);
            }
            printf("\n");
        }
    }

    return 0;
}
```

Ниже представлены замеры времени выполнения программы для матриц размерности 512×512.

Оптимизация алгоритма

Рассмотрим некоторые методы оптимизации программы с использованием компиляторов GCC и Clang. Каждый последующий метод будет добавляться к успешно выполненному предыдущему.

1. GCC

Без каких-либо мероприятий по оптимизации программы среднее время выполнения программы составило 1.197152 секунд.

1.1 Кэш оптимизация

Рассмотрим все варианты комбинаций последовательности циклов в программе. Тестирование проводилось на матрице размерности 512×512. Полученные результаты приведены в таблице 2.

Таблица 2 – Время выполнения

i	j	k	1.197152
i	k	j	0.833544
j	i	k	1.323416
j	k	i	1.694051
k	i	j	0.867555
k	j	i	1.584708

Программа стала работать быстрее примерно на 0,3 секунды. Полученный результат обуславливается тем, что в таком варианте данные в памяти расположены ближе друг к другу, благодаря чему количество кэш-промахов незначительно, но меньше.

1.2 Ключи оптимизации

Рассмотрим все варианты работы программы с использованием ключей оптимизации. Тестирование проводилось на матрице размерности 512×512. Полученные результаты приведены в таблице 3.

Таблица 3 – Время выполнения

-O1	-O2	-O3
0,6334	0,8577	0,9202

При использовании ключа оптимизатора O1 программа стала работать быстрее.

1.3 OpenMP

Используем директиву OpenMP для распараллеливания циклов. Рассмотрим некоторые варианты комбинаций ее использования. OMP1 – распараллеливание внешнего цикла, OMP2 – распараллеливание внешнего и второго внутреннего циклов, OMP3 – распараллеливание всех трех циклов. Полученные результаты приведены в таблице 4.

Таблица 4 – Время выполнения

OMP1	OMP2	OMP3
0,1699	0,1299	1,1300

Программа стала работать быстрее при распараллеливании внешнего и второго внутреннего циклов, это обусловлено тем, что при распараллеливании вложенных циклов возрастают издержки по созданию потоков, для создания потоков тратится больше ресурсов. Ниже представлен фрагмент кода с добавлением OMP2.

Листинг 2 – Использование OpenMP

```
...
#pragma omp parallel for
    for(int i = 0; i < M_SIZE; ++i){
        #pragma omp parallel for
        for(int j = 0; j <M_SIZE; ++j){
            for(int k = 0; k < M_SIZE; k+=4){
                res[i][j] += m1[i][k]*m2[k][j];
            }
        }
    }
...
```

1.4 Векторизация

Используем векторизацию для параллельного умножения каждых четырех элементов строки матрицы на соответствующие им четыре элемента столбца второй матрицы. Но для упрощения задачи векторизации пришлось отказаться от измененного порядка циклов (см. п. 1), вместо этого вторая матрица предварительно транспонируется и умножается не строка на столбец, а строка на строку, что в некоторой степени уменьшает количество кэш-промахов. Полученные результаты приведены в таблице 5.

Таблица 5 – Время выполнения

Векторизация	Векторизация + OMP2
0,2899	0,0899

Ниже представлен фрагмент кода с добавлением векторизации и OpenMP.

Листинг 3 – Использование векторизации

```
...
#pragma omp parallel for
```

```
    for(int i = 0; i < M_SIZE; ++i){
        #pragma omp parallel for
        for(int j = 0; j <M_SIZE; ++j){
            __m128i a, b, c, tmp;
            c = _mm_setzero_si128();
            for(int k = 0; k < M_SIZE; k+=4){
                a = _mm_loadu_si128((__m128i*)&m1[i][k]);
                b = _mm_loadu_si128((__m128i*)&m2[j][k]);
                tmp = _mm_mullo_epi32(a, b);
                c = _mm_add_epi32(c, tmp);
            }
            int *result = (int*)&c;
            res[i][j] = result[0] + result[1] + result[2] + result[3];
        }
    }
...
```

1.5 Блочное умножение

Используем блочное умножение для разбиения исходных матриц на блоки, что влечет за собой более частое обращение к данным, расположенных рядом. В ходе тестирования подобрали оптимальный размер блока – 16×16. Результат выполнения данной программы с добавлением векторизации и OpenMP – 0,0801 с. Ниже представлен фрагмент кода с использованием блочного умножения.

Листинг 4 – Использование блочного умножения

```
...
#pragma omp parallel for
    for(int ih = 0; ih < M_SIZE; ih += BLOCK_S) {
        #pragma omp parallel for
        for(int jh = 0; jh < M_SIZE; jh += BLOCK_S) {
            __m128i a, b, c, tmp;
            for(int kh = 0; kh < M_SIZE; kh += BLOCK_S){
                for(int il = ih; il < ih + BLOCK_S; ++il){
                    for(int jl = jh; jl < jh + BLOCK_S; ++jl){
                        c = _mm_setzero_si128();
                        for(int kl = kh; kl < kh + BLOCK_S; kl+=4){
                            a = _mm_loadu_si128((__m128i*)&m1[il][kl]);
                            b = _mm_loadu_si128((__m128i*)&m2[jl][kl]);
                            tmp = _mm_mullo_epi32(a, b);
                            c = _mm_add_epi32(c, tmp);
                        }
                        int *result = (int*)&c;
                        res[il][jl] = result[0] + result[1] + result[2] +
result[3];
                    }
                }
            }
        }
    }
...
```

2. Clang

Те же самые коды программ были запущены с использованием компилятора Clang, и были получены следующие показатели времени исполнения программы

2.1 Кэш оптимизация: 0,9751 с;

2.2 Ключи оптимизации: 0,9374 с;

2.3 OpenMP: 0,1874 с;

2.4 Векторизация: 0,1519 с;
2.5 Блочное умножение: 0,09 с.

Сравнительный анализ

Ниже представлена таблица с информацией о производительности методов с использованием различных компиляторов.

Таблица 6 – Сравнение производительности методов оптимизации

Метод оптимизации	GCC		
	Running time	Прирост относительно прошлого шага	Прирост относительно варианта без оптимизации
Без оптимизации	1,1971	1,00	1,00
+ Кэш оптимизация	0,8335	1,44	1,44
+ Ключ оптимизации O1	0,6334	1,32	1,89
+ Использование OpenMP	0,1299	4,88	9,22
+ Векторизация	0,0899	1,44	13,32
+ Блочное умножение	0,0801	1,12	14,95

Таблица 7 – Сравнение производительности методов оптимизации

Метод оптимизации	Clang		
	Running time	Прирост относительно прошлого шага	Прирост относительно варианта без оптимизации
Без оптимизации	1,5323	1,00	1,00
+ Кэш оптимизация	0,9752	1,57	1,23
+ Ключ оптимизации O1	0,9374	1,04	1,28
+ Использование OpenMP	0,1827	5,13	6,55
+ Векторизация	0,1025	1,78	11,68
+ Блочное умножение	0,0913	1,12	13,11

По полученным результатам были построены графики производительности программы (см. рис. 1-2).

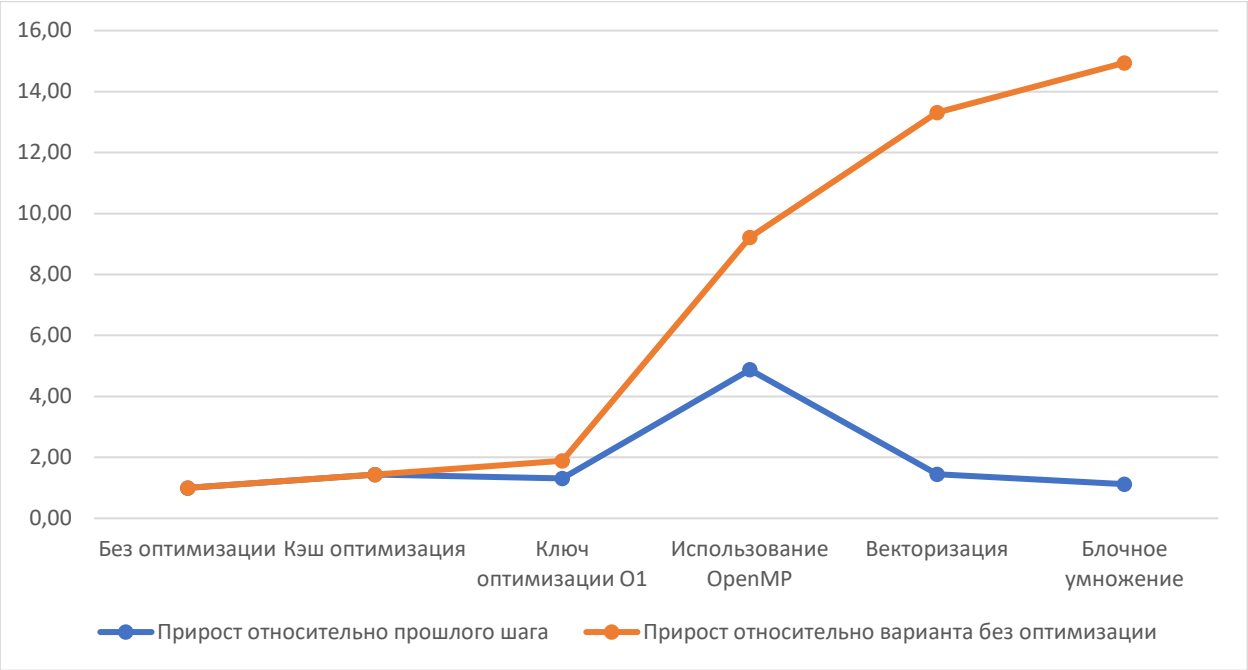

Рисунок 1 – График производительности программы с использованием компилятора GCC

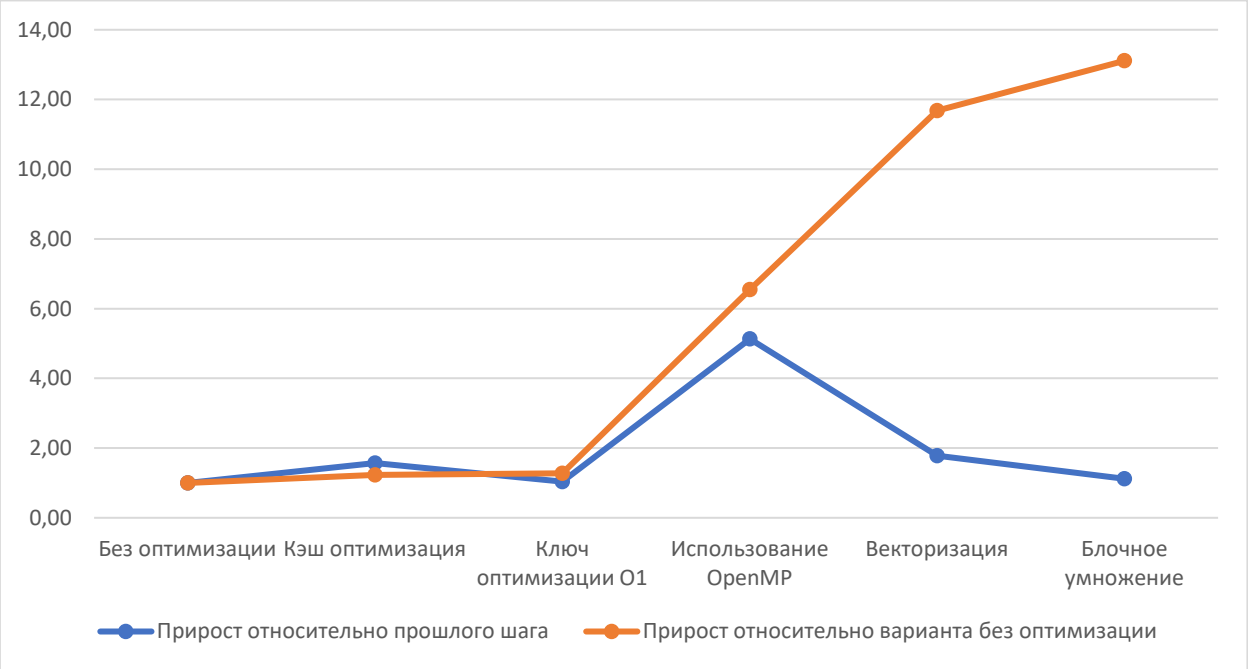

Рисунок 2 – График производительности программы с использованием компилятора Clang

Вывод

В результате выполнения лабораторной работы были получены знания о процессах оптимизации, а также навыки их практического применения. Были реализованы разные методы оптимизации программы, выполняющей умножение матриц, размером 512x512, которые суммарно дали прирост производительности в 13-14 раз.