Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации Сибирский Государственный Университет Телекоммуникаций и Информатики СибГУТИ

Кафедра вычислительных систем

Лабораторная работа №4

по дисциплине “Архитектура вычислительных систем”

Выполнил:

Студент группы ИП-912

Лёвкин Игорь Андреевич

Работу проверил: Токмашева Е.И.

Новосибирск, 2021

Лабораторная работа № 4. Оптимизация доступа к памяти.

1. На языке С/С++/C# реализовать функцию DGEMM BLAS последовательное умножение двух

квадратных матрицс элементами типа double. Обеспечить возможность задавать

размерностиматриц в качестве аргумента командной строки при запуске программы.

Инициализировать начальные значения матриц случайными числами.

2. Провести серию испытаний и построить график зависимости времени выполнения

программы от объёма входных данных. Например, для квадратных матриц с числом

строк/столбцов 1000, 2000, 3000, … 10000.

3. Оценить предельные размеры матриц, которые можно перемножить на вашем

вычислительном устройстве.

4. Реализовать дополнительную функцию DGEMM\_opt\_1, в которой выполняется

оптимизация доступа к памяти, за счет построчного перебора элементов обеих матриц.

5. \* Реализовать дополнительную функцию DGEMM\_opt\_2, в которой выполняется

оптимизация доступа к памяти, за счет блочного перебора элементов матриц. Обеспечить

возможность задавать блока, в качестве аргументафункции.

6. \*\* Реализовать дополнительную функцию DGEMM\_opt\_3, в которой выполняется

оптимизация доступа к памяти, за счет векторизации кода.

7. Оценить ускорение умножения для матриц фиксированного размера, например,

1000х1000, 2000х2000, 5000х5000, 10000х10000.

\* Для блочного умножения матриц определить размер блока, при котором достигается

максимальное ускорение.

8. С помощью профилировщика для исходной программы и каждого способа оптимизации

доступа к памяти оценить количество промахов при работе к КЭШ памятью (cache-misses).

9. Подготовить отчет отражающий суть, этапы и результаты проделанной работы.

Результат работы программы:

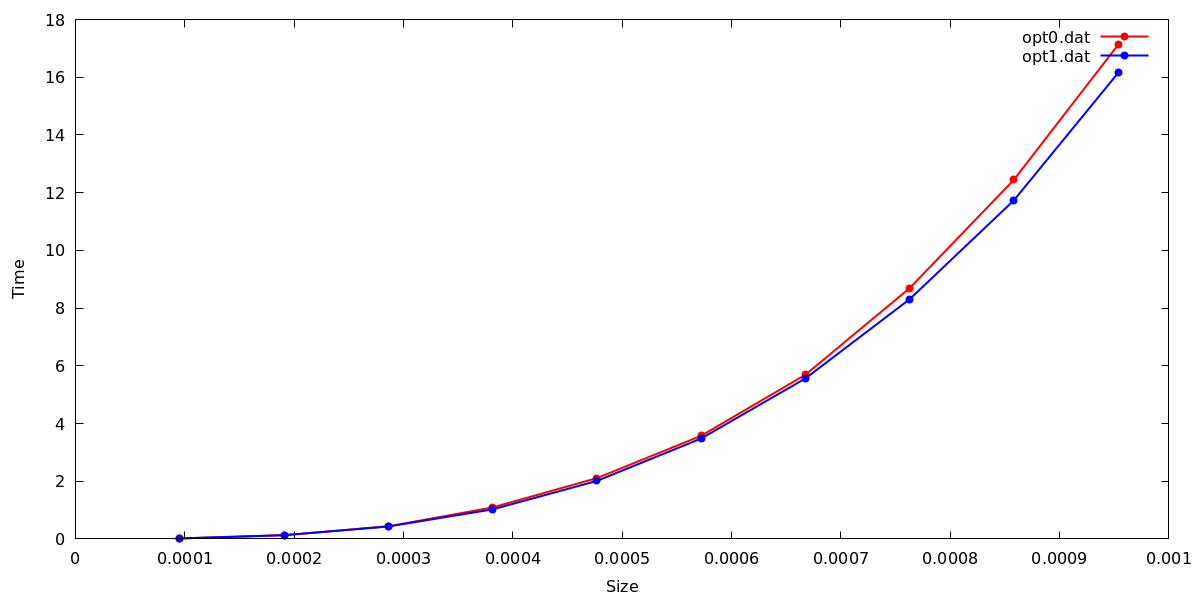


График . Зависимость времени от размера матрицы.

На графике можно пронаблюдать экспоненциальную зависимость времени, затраченного на умножение матриц от их размера.

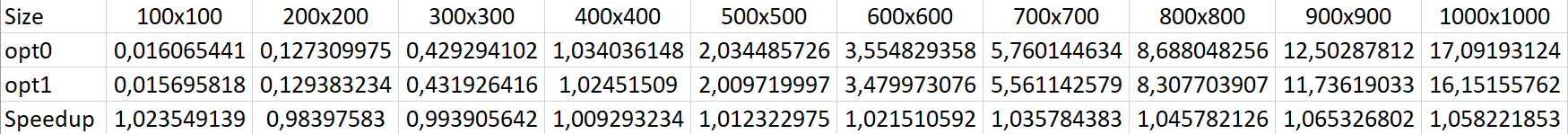
Предельные размеры матриц равны: 50000x50000. Опытным путём было выяснено что этом и большем значениях процесс завершается с сообщением killed, по причине потребления слишком большого количества ресурсов.  
 

Таблица 1. Ускорение умножения для матриц фиксированного размера.

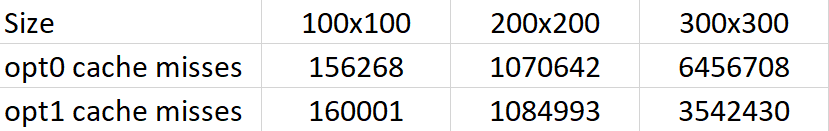


Таблица 2. Количество промахов при работе с КЭШ памятью

Листинг:

#include <iostream>

#include <vector>

#include <cstring>

#include <chrono>

#include <iomanip>

using namespace std;

void fillRandMatrix(vector<vector<double>> &*matrix*)

{

    for (size\_t i = 0; i < *matrix*.size(); i++)

    {

        for (size\_t j = 0; j < *matrix*[i].size(); j++)

        {

*matrix*[i][j] = (double)(rand()) / RAND\_MAX \* (*matrix*.size());

        }

    }

}

double dgemmBlas(const vector<vector<double>> &*matrixA*,

                 const vector<vector<double>> &*matrixB*,

                 vector<vector<double>> &*matrixC*)

{

    if (*matrixA*.size() != *matrixB*.size())

    {

        return 0;

    }

    size\_t matrixSize = *matrixA*.size();

*matrixC*.clear();

*matrixC*.shrink\_to\_fit();

*matrixC*.resize(matrixSize, vector<double>(matrixSize, 0));

    auto timerStart = chrono::high\_resolution\_clock::now();

    auto timerStop = chrono::high\_resolution\_clock::now();

    timerStart = chrono::high\_resolution\_clock::now();

    for (int i = 0; i < matrixSize; i++)

    {

        for (int j = 0; j < matrixSize; j++)

        {

            for (int k = 0; k < matrixSize; k++)

            {

*matrixC*[i][j] += *matrixA*[i][k] \* *matrixB*[k][j];

            }

        }

    }

    timerStop = chrono::high\_resolution\_clock::now();

    return chrono::duration<long double>(timerStop - timerStart).count();

}

double dgemmOpt1(const vector<vector<double>> &*matrixA*,

                 const vector<vector<double>> &*matrixB*,

                 vector<vector<double>> &*matrixC*)

{

    if (*matrixA*.size() != *matrixB*.size())

    {

        return 0;

    }

    size\_t matrixSize = *matrixA*.size();

*matrixC*.clear();

*matrixC*.shrink\_to\_fit();

*matrixC*.resize(matrixSize, vector<double>(matrixSize, 0));

    auto timerStart = chrono::high\_resolution\_clock::now();

    auto timerStop = chrono::high\_resolution\_clock::now();

    timerStart = chrono::high\_resolution\_clock::now();

    for (int i = 0; i < matrixSize; i++)

    {

        for (int k = 0; k < matrixSize; k++)

        {

            for (int j = 0; j < matrixSize; j++)

            {

*matrixC*[i][j] += *matrixA*[i][k] \* *matrixB*[k][j];

            }

        }

    }

    timerStop = chrono::high\_resolution\_clock::now();

    return chrono::duration<long double>(timerStop - timerStart).count();

}

int main(int *argc*, char \**argv*[])

{

    size\_t matrixSize = 0;

    for (size\_t i = 0; i < static\_cast<size\_t>(*argc*); i++)

    {

        if (!strcmp(*argv*[i], "-s") || !strcmp(*argv*[i], "--matrix-size"))

        {

            matrixSize = atoll(*argv*[i + 1]);

        }

    }

    vector<vector<double>> matrixA(matrixSize, vector<double>(matrixSize, 0));

    vector<vector<double>> matrixB(matrixSize, vector<double>(matrixSize, 0));

    vector<vector<double>> matrixC(matrixSize, vector<double>(matrixSize, 0));

    fillRandMatrix(matrixA);

    fillRandMatrix(matrixB);

    double elapsedTime = 0;

    for (size\_t i = 0; i < static\_cast<size\_t>(*argc*); i++)

    {

        if (!strcmp(*argv*[i], "opt0"))

        {

            elapsedTime = dgemmBlas(matrixA, matrixB, matrixC);

        }

        else if (!strcmp(*argv*[i], "opt1"))

        {

            elapsedTime = dgemmOpt1(matrixA, matrixB, matrixC);

        }

    }

    cout << fixed << setprecision(16) << matrixSize << "x" << matrixSize << ";" << elapsedTime << endl;

    return EXIT\_SUCCESS;

}