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 ИВ-921
Ярошев Р. А..

Работу проверил:
Ассистент кафедры ВС
Петухова Я.В.

Новосибирск 2021

## Задание

1. На языке C/C++/C# реализовать функцию DGEMM BLAS последовательное умножение двух
квадратных матриц с элементами типа double. Обеспечить возможность задавать
размерности матриц 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 аргумента функции.
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,
1000x1000, 2000x2000, 5000x5000, 10000x10000.
* 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.

## Результаты работы

В ходе данной работы были проведены несколько тестов.

1) Сперва опытным путем вычисляем зависимость времени выполнения от размера матрицы с типом значений - double.

Тесты проведены для матриц, размером 400, 800, 1200, 1600 и 2000 элементов.

| multiplication type | launch count | matrix size | block size | timer | average time | absError | relError |
|---|---|---|---|---|---|---|---|
| usual | 3 | 400 | 64 | clock() | 4.608213e-01 | 4.738849e-02 | 4.873188e-01% |
| usual | 3 | 800 | 64 | clock() | 3.767132e+00 | 2.696565e-01 | 1.930238e+00% |
| usual | 3 | 1200 | 64 | clock() | 1.788878e+01 | 7.210071e-01 | 2.906019e+00% |
| usual | 3 | 1600 | 64 | clock() | 6.314992e+01 | 3.341893e+00 | 1.768530e+01% |
| usual | 3 | 2000 | 64 | clock() | 1.224532e+02 | 5.341880e-01 | 2.330334e-01% |

Таблица 1. csv — файл.

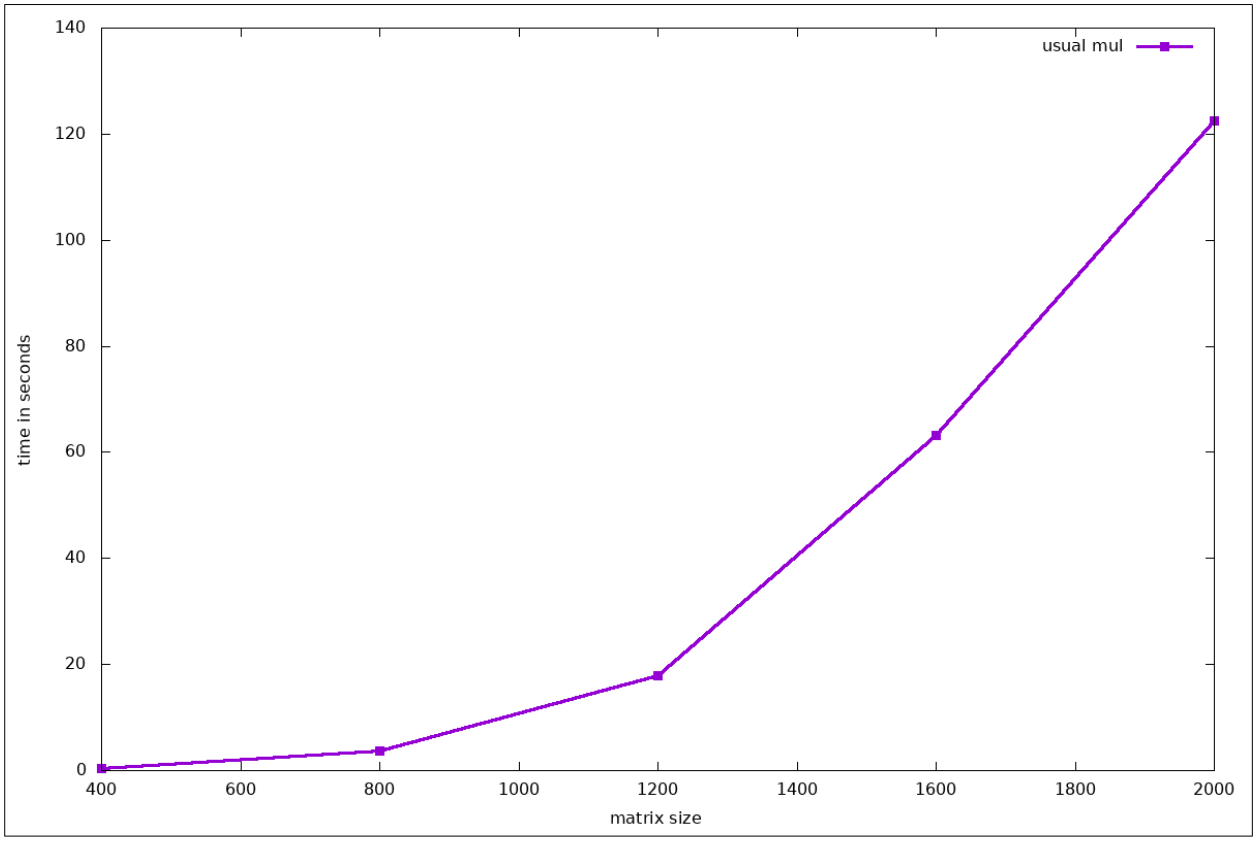


График 1. Зависимость времени выполнения от размера данных.

Из данного графика видно, что при увеличении размера матрицы время выполнения также растет. При чем данная зависимость — нелинейная, т. е. При увеличении числа данных в 2 раза, время выполнения растет более чем в 2 раза.

Также при увеличении объема данных скачкообразно увеличивается и абсолютная погрешность.

2) При умножении матриц размером 6400 элементов время выполнения составило порядка суток.

Оптимальный размер — порядка 4000 элементов.

| 1 | multiplication type | launch count | matrix size | block size | timer | average time | absError | relError |
|---|---|---|---|---|---|---|---|---|
| 2 | usual | 3 | 400 | 64 | clock() | 4.936367e-01 | 3.159530e-02 | 2.022263e-01% |
| 3 | usual | 3 | 800 | 64 | clock() | 4.489526e+00 | 2.966090e-01 | 1.959603e+00% |
| 4 | usual | 3 | 1200 | 64 | clock() | 2.030827e+01 | 3.906991e-01 | 7.516436e-01% |
| 5 | usual | 3 | 1600 | 64 | clock() | 6.291539e+01 | 1.833092e+00 | 5.340862e+00% |
| 6 | usual | 3 | 2000 | 64 | clock() | 1.248913e+02 | 1.097373e+01 | 9.642203e+01% |
| 7 | usual | 3 | 2400 | 64 | clock() | 2.090520e+02 | 1.728660e+01 | 1.429436e+02% |
| 8 | usual | 3 | 2800 | 64 | clock() | 2.946034e+02 | 3.557273e+00 | 4.295331e+00% |
| 9 | usual | 3 | 3200 | 64 | clock() | 4.611623e+02 | 1.486745e+00 | 4.793129e-01% |
| 10 | usual | 3 | 3600 | 64 | clock() | 6.864195e+02 | 3.145944e+00 | 1.441825e+00% |
| 11 | usual | 3 | 4000 | 64 | clock() | 9.545873e+02 | 4.841552e+01 | 2.455576e+02% |
| 12 | usual | 3 | 4400 | 64 | clock() | 1.065937e+03 | 2.217732e+01 | 4.614097e+01% |
| 13 | usual | 3 | 4800 | 64 | clock() | 1.368039e+03 | 1.392352e+01 | 1.417097e+01% |
| 14 | usual | 3 | 5200 | 64 | clock() | 2.317450e+03 | 9.229993e+01 | 3.676143e+02% |
| 15 | usual | 3 | 5600 | 64 | clock() | 2.143021e+03 | 8.817485e+01 | 3.627964e+02% |
| 16 | usual | 3 | 6000 | 64 | clock() | 2.558349e+03 | 1.769625e+01 | 1.224059e+01% |
| 17 | usual | 3 | 6400 | 64 | clock() | 3.191064e+03 | 3.578476e-01 | 4.012922e-03% |

Таблица 2. csv — файл. Оптимальный размер матрицы

3) проведена оценка зависимости времени на умножение матриц тремя способами от количества элементов матрицы (256, 512, 768, 1024 элементов).

| 1 | multiplication type | launch count | matrix size | block size | timer | average time | absError | relError |
|---|---|---|---|---|---|---|---|---|
| 2 | usual | 3 | 256 | 64 | clock() | 7.330500e-02 | 7.071223e-03 | 6.821116e-02% |
| 3 | row_by_row | 3 | 256 | 64 | clock() | 6.560333e-02 | 1.685788e-02 | 4.331914e-01% |
| 4 | block | 3 | 256 | 64 | clock() | 3.900533e-02 | 1.907128e-03 | 9.324715e-03% |
| 5 | usual | 3 | 512 | 64 | clock() | 5.879697e-01 | 7.057347e-03 | 8.470871e-03% |
| 6 | row_by_row | 3 | 512 | 64 | clock() | 4.054763e-01 | 6.359102e-03 | 9.973007e-03% |
| 7 | block | 3 | 512 | 64 | clock() | 2.982063e-01 | 9.149388e-03 | 2.807160e-02% |
| 8 | usual | 3 | 768 | 64 | clock() | 1.996105e+00 | 4.155119e-02 | 8.649352e-02% |
| 9 | row_by_row | 3 | 768 | 64 | clock() | 1.354763e+00 | 2.804585e-02 | 5.805958e-02% |
| 10 | block | 3 | 768 | 64 | clock() | 9.314463e-01 | 1.997853e-02 | 4.285182e-02% |
| 11 | usual | 3 | 1024 | 64 | clock() | 5.900832e+00 | 4.161277e-01 | 2.934540e+00% |
| 12 | row_by_row | 3 | 1024 | 64 | clock() | 3.239475e+00 | 5.401925e-02 | 9.007877e-02% |
| 13 | block | 3 | 1024 | 64 | clock() | 2.385500e+00 | 1.031206e-01 | 4.457702e-01% |

Таблица 3. csv — файл.

По данной таблице построен график:

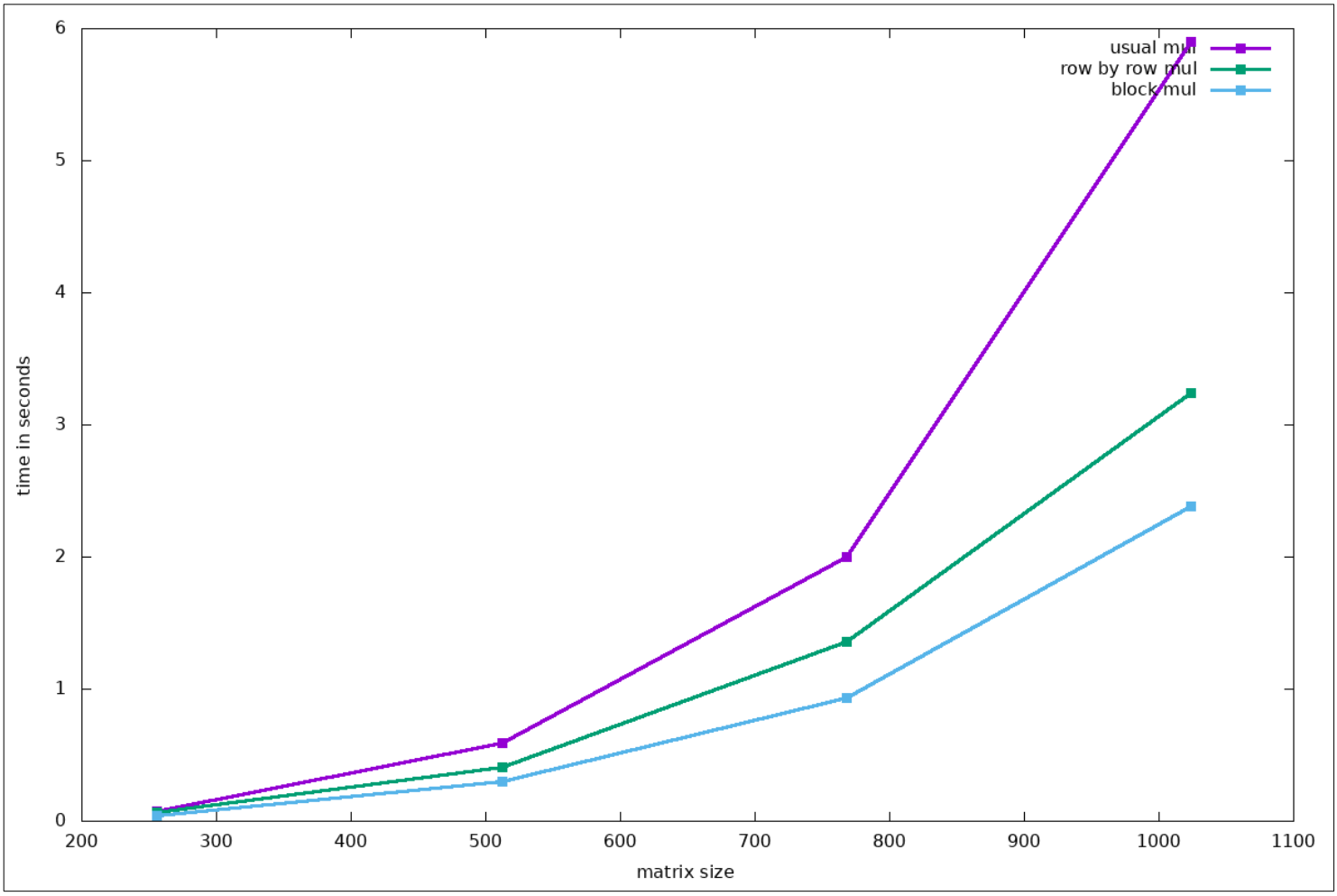


График 2. Способы умножения матриц

Из данного графика следует, что наиболее эффективно блочное умножение матриц.

Чуть менее эффективно построчное умножение.

Наименее  эффективным будет стандартное умножение — строка на столбец.

4) Был построен график ускорения для матриц фиксированного размера

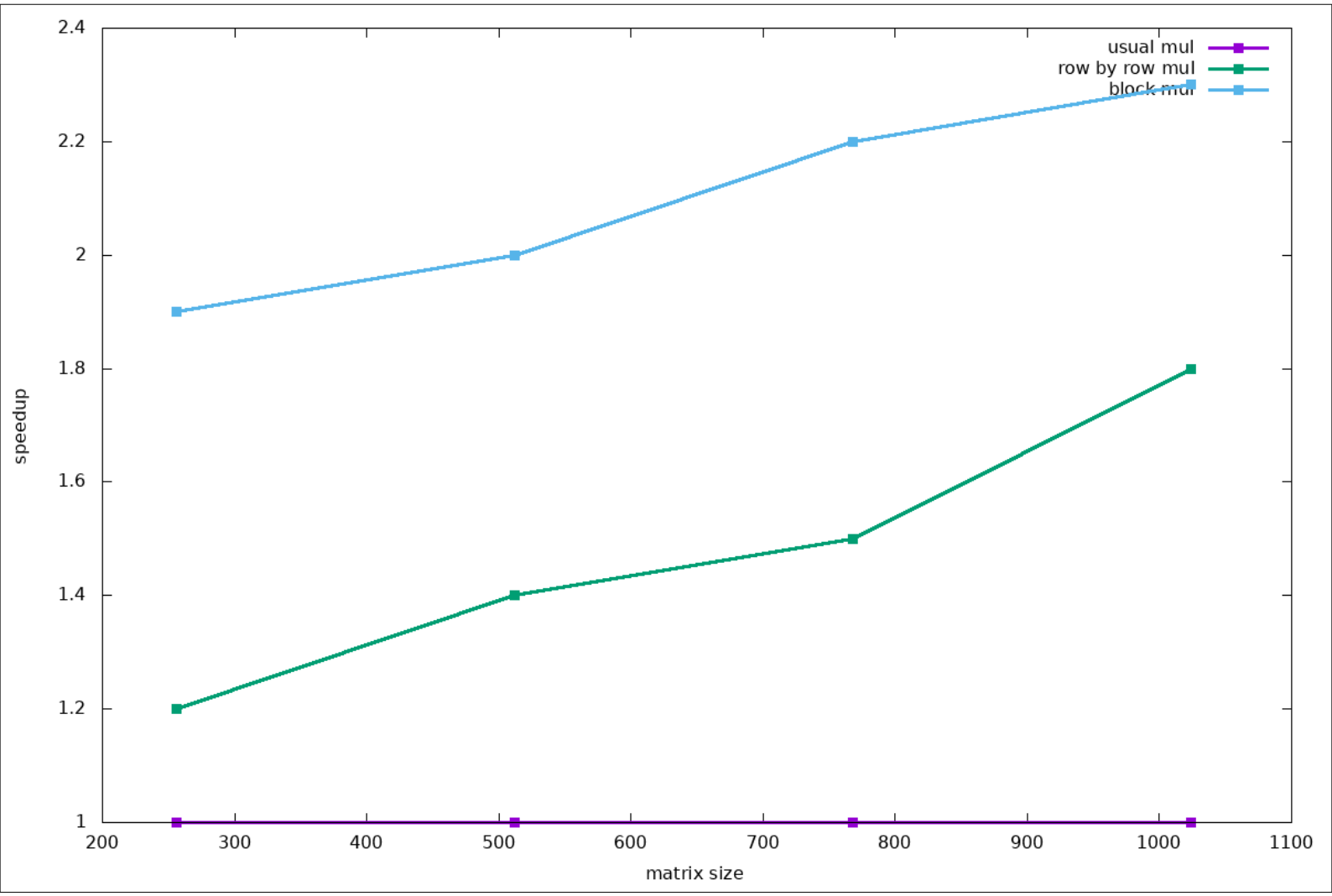


График 3. Ускорение умножения для матриц фиксированного размера.

| | A | B | C | D |
|---|---|---|---|---|
| 1 | multiplication type | matrix size | block size | speedup |
| 2 | usual | 256 | 64 | 1 |
| 3 | row_by_row | 256 | 64 | 1.2 |
| 4 | block | 256 | 64 | 1.9 |
| 5 | usual | 512 | 64 | 1 |
| 6 | row_by_row | 512 | 64 | 1.4 |
| 7 | block | 512 | 64 | 2.0 |
| 8 | usual | 768 | 64 | 1 |
| 9 | row_by_row | 768 | 64 | 1.5 |
| 10 | block | 768 | 64 | 2.2 |
| 11 | usual | 1024 | 64 | 1 |
| 12 | row_by_row | 1024 | 64 | 1.8 |
| 13 | block | 1024 | 64 | 2.3 |

Таблица 4. Ускорение умножения для матриц фиксированного размера.

5) Произведена оценка промахов при работе с кеш-памятью (cache-misses) для четырех видов оптимизации (O0, O1, O2, O3):

```
1  COMP = g++
2  FLAGS = -O0 -Wall -g -o
3  OBJS = main.cpp foo.cpp
```

```
Performance counter stats for './main':

        1 749 919      cache-misses

       3,005531056 seconds time elapsed

       2,979188000 seconds user
       0,004004000 seconds sys
```

Рисунок 1.  cache-misses при O0

```
COMP = g++
FLAGS = -O1 -Wall -g -o
OBJS = main.cpp foo.cpp
```

```
Performance counter stats for './main':

        1 662 083      cache-misses

      1,665518385 seconds time elapsed

      1,636021000 seconds user
      0,012029000 seconds sys
```

Рисунок 2.  cache-misses при O1

```
COMP = g++
FLAGS = -O2 -Wall -g -o
OBJS = main.cpp foo.cpp
```

```
Performance counter stats for './main':

        1 304 045      cache-misses

      0,942141996 seconds time elapsed

      0,908749000 seconds user
      0,012063000 seconds sys
```

Рисунок 3. cache-misses при O2

```
COMP = g++
FLAGS = -O3 -Wall -g -o
OBJS = main.cpp foo.cpp
```

```
Performance counter stats for './main':

        1 308 312      cache-misses

      0,876183507 seconds time elapsed

      0,842558000 seconds user
      0,012036000 seconds sys
```

Рисунок 3. cache-misses при O3

Видим, что с усилением оптимизации уменьшается число промахов.

- -O0 (O ноль) - это самые простые и примитивные оптимизации.
- -O1 - более сильные оптимизации.
- -O2 - оптимизировать все, что можно, но только проверенные и надежные оптимизации.
- -O3 - жесткая и насильная оптимизация, применяются экспериментальные методы.

## Листинг

### Foo.cpp

```
#include "foo.h"

int ProcessParameters(int argc, char *argv[],
                      long long &matrixSize, char* mulType,
                      long long &launchCnt, bool &bCheck,
                      long long &blockSize)
{
    int i;
    for (i = 1; i < argc; i++)
    {

        if (strcmp("-s", argv[i]) == 0 ||
            strcmp("--matrix-size", argv[i]) == 0)
        {
            i++;
            matrixSize = atoll(argv[i]);
            if (matrixSize == 0) {
                printf("Error in arguments of main(): incorrect value for --
launch-count \n");
                return 1;
            }
        }

        else if (strcmp("-t", argv[i]) == 0 ||
                 strcmp("--multiplication-type", argv[i]) == 0)
        {
            i++;
            if (strcmp("usual", argv[i]) == 0 ||
                strcmp("row_by_row", argv[i]) == 0 ||
                strcmp("block", argv[i]) == 0)
            {
                strcpy(mulType, argv[i]);
            }
            else {
                printf("Error in arguments of main(): incorrect value for --
multiplication-type \n");
                return 1;
            }
        }

        else if (strcmp("-l", argv[i]) == 0 ||
                 strcmp("--launch-count", argv[i]) == 0)
        {
            i++;
```

```
            launchCnt = atoll(argv[i]);
            if (launchCnt == 0) {
                printf("Error in arguments of main(): incorrect value for --
launch-count \n");
                return 1;
            }
        }

        else if (strcmp("-c", argv[i]) == 0 ||
                 strcmp("--check", argv[i]) == 0)
        {
            bCheck = true;
        }
        else if (strcmp("-b", argv[i]) == 0 ||
                 strcmp("--block-size", argv[i]) == 0)
        {
            i++;
            blockSize = atoll(argv[i]);
            if (blockSize == 0) {
                printf("Error in arguments of main(): incorrect value for --
block-size \n");
                return 1;
            }
        }
    }

    return 0;
}

long long min(long long a, long long b) {
    if (a < b)
        return a;
    else
        return b;
}

long long GetCacheAlignment() {
    FILE *fcpu;
    if ((fcpu = fopen("/proc/cpuinfo", "r")) == NULL) {
        printf("Error: can't open /proc/cpuinfo \n");
        return -1;
    }
    size_t m = 0;
    char *line = NULL, *temp = (char*) malloc(50);
    while (getline(&line, &m, fcpu) > 0) {
        if (strstr(line, "cache_alignment")) {
            strcpy(temp, &line[18]);
            break;
        }
    }
```

```
    for (int i = 0; i < 50; i++) {
        if (temp[i] == ' ' || temp[i] == '\n') {
            temp[i] = '\0';

        }
    }
    long long val = atoll(temp);
    if (val == 0) {
        printf("Error in GetCacheAlignment(): can't atoll \n");
        return -1;
    }

    fclose(fcpu);
    free(temp);
    return val;
}

int PrintMatrix(double **matrix, long long n) {
    long long i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%.6f ", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}

int MatrixMul(int mulType_i, long long matrixSize,
              bool bCheck, double &time_d) {
    long long i, j, k;

    double **matrix1 = new double*[matrixSize];
    double **matrix2 = new double*[matrixSize];
    double **matrixRes = new double*[matrixSize];
    for (i = 0; i < matrixSize; i++) {
        matrix1[i] = new double[matrixSize];
        matrix2[i] = new double[matrixSize];
        matrixRes[i] = new double[matrixSize];
    }

    for (i = 0; i < matrixSize; i++) {
        for (j = 0; j < matrixSize; j++) {
            matrix1[i][j] = rand() / 123456 + (double)rand() / RAND_MAX;
            matrix2[i][j] = rand() / 123456 + (double)rand() / RAND_MAX;
            matrixRes[i][j] = 0;
        }
    }

    if (bCheck) {
```

```
    printf("before multiplication: \n");
    PrintMatrix(matrix1, min((long long)3, matrixSize));
    printf("\n");
    PrintMatrix(matrix2, min((long long)3, matrixSize));
    printf("\n");
    PrintMatrix(matrixRes, min((long long)3, matrixSize));
    printf("\n");
}

clock_t start, stop;
long long time_i = 0;
start = clock();

if (mulType_i == 1) {
    for (i = 0; i < matrixSize; i++)
    for (j = 0; j < matrixSize; j++)
    for (k = 0; k < matrixSize; k++) {

        matrixRes[i][j]  += matrix1[i][k] * matrix2[k][j];
    }
}
else if (mulType_i == 2) {
    for (i = 0; i < matrixSize; i++)
    for (k = 0; k < matrixSize; k++)
    for (j = 0; j < matrixSize; j++) {
        matrixRes[i][j]  += (double)matrix1[i][k] * matrix2[k][j];
    }
}
else {
    printf("Error in MatrixMul(), wrong mulType_i");
}

stop = clock();
time_i += stop - start;
time_d = (double)time_i / CLOCKS_PER_SEC;

if (bCheck) {
    printf("\nafter multiplication: \n");
    printf("time_d=%f \n", time_d);
    PrintMatrix(matrix1, min((long long)3, matrixSize));
    printf("\n");
    PrintMatrix(matrix2, min((long long)3, matrixSize));
    printf("\n");
    PrintMatrix(matrixRes, min((long long)3, matrixSize));
    printf("\n");
}

for (i = 0; i < matrixSize; i++) {
    delete(matrix1[i]);
    delete(matrix2[i]);
    delete(matrixRes[i]);
```

```
    }
    delete(matrix1);
    delete(matrix2);
    delete(matrixRes);
    return 0;
}

int MatrixBlockMul(long long blockSize, long long matrixSize,
                   bool bCheck, double &time_d) {
    long long i, j, k;

    double *matrix1 = new double[matrixSize*matrixSize];
    double *matrix2 = new double[matrixSize*matrixSize];
    double *matrixRes = new double[matrixSize*matrixSize];

    for (i = 0; i < matrixSize*matrixSize; i++) {
        matrix1[i] = rand() / 123456 + (double)rand() / RAND_MAX;
        matrix2[i] = rand() / 123456 + (double)rand() / RAND_MAX;
        matrixRes[i] = 0;
    }

    clock_t start, stop;
    long long time_i = 0;
    start = clock();

    double *m1, *m2, *mRes;
    long long i0, j0, k0;

    for (i = 0; i < matrixSize; i += blockSize)
    for (j = 0; j < matrixSize; j += blockSize)
    for (k = 0; k < matrixSize; k += blockSize) {
        for (i0 = 0, mRes = (matrixRes + i * matrixSize + j),
             m1 = (matrix1 + i * matrixSize + k); i0 < blockSize;
             ++i0, mRes += matrixSize, m1 += matrixSize)
        {
            for (k0 = 0, m2 = (matrix2 + k * matrixSize + j);
                 k0 < blockSize; ++k0, m2 += matrixSize)
            {
                for (j0 = 0; j0 < blockSize; ++j0)
                    mRes[j0] += m1[k0] * m2[j0];
            }
        }
    }

    stop = clock();
    time_i += stop - start;
    time_d = (double)time_i / CLOCKS_PER_SEC;

    delete(matrix1);
    delete(matrix2);
    delete(matrixRes);
```

```
    return 0;
}


int WriteToCSV(char* mulType, long long launchCnt, long long matrixSize,
               long long blockSize,
               double avgTime, double absError, double relError)
{

    FILE *fout;
    if ((fout = fopen("../data/output.csv", "a")) == NULL) {
        printf("Error in Write_to_csv(): can't open output.csv \n");
        return 1;
    }

    fprintf(fout, "%s;", mulType);
    fprintf(fout, "%lld;", launchCnt);
    fprintf(fout, "%lld;", matrixSize);
    fprintf(fout, "%lld;", blockSize);
    fprintf(fout, "clock();"); ///Timer
    fprintf(fout, "%e;", avgTime);
    fprintf(fout, "%e;", absError);
    fprintf(fout, "%e%%;", relError);
    fprintf(fout, "\n");

    return 0;
}

int TestsHandler(char* mulType, long long launchCnt,
                 long long matrixSize, bool bCheck,
                 long long blockSize)
{

    double summand1 = 0, summand2 = 0;
    double time_d[launchCnt];

    for (long long i = 0; i < launchCnt; i++) {
        if (strcmp("usual", mulType) == 0) {
            MatrixMul(1, matrixSize, bCheck, time_d[i]);
        }
        else if (strcmp("row_by_row", mulType) == 0) {
            MatrixMul(2, matrixSize, bCheck, time_d[i]);
        }
        else if (strcmp("block", mulType) == 0) {
            MatrixBlockMul(blockSize, matrixSize, bCheck, time_d[i]);
        }
        summand1 += time_d[i] * time_d[i];
        summand2 += time_d[i];
    }

    summand1 /= launchCnt;
```

```
    summand2 /= launchCnt;
    double avgTime = summand2;
    summand2 *= summand2;
    double dispersion = summand1 - summand2;
    double absError = sqrt(dispersion);
    double relError = dispersion / avgTime * 100;
    WriteToCSV(mulType, launchCnt, matrixSize, blockSize, avgTime, absError,
relError);
    return 0;
}
```

## main.cpp

```
#include "foo.h"

int main(int argc, char *argv[]) {
    srand(time(0));
    long long matrixSize = 100;
    char* mulType = (char*) malloc(15);
    strcpy(mulType, "usual\0");
    long long launchCnt = 3;
    bool bCheck = false;
    long long blockSize = GetCacheAlignment();
    ProcessParameters(argc, argv, matrixSize, mulType, launchCnt, bCheck,
blockSize);

    printf("--- arguments of main(): --- \n");
    printf("matrixSize = %lld \n", matrixSize);
    printf("mulType = %s \n", mulType);
    printf("launchCnt = %lld \n", launchCnt);
    printf("bCheck = %d \n", bCheck ? 1 : 0);
    if (strcmp("block", mulType) == 0)
        printf("blockSize = %lld \n", blockSize);

    TestsHandler(mulType, launchCnt, matrixSize, bCheck, blockSize);

    free(mulType);
    return 0;
}
```