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

Кафедра Вычислительных систем

# Лабораторная работа №5
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. Для программы умножения двух квадратных матриц DGEMM BLAS разработанной в
задании 4 на языке C/C++ реализовать многопоточные вычисления. В потоках
необходимо реализовать инициализацию массивов случайными числами типа

double и равномерно распределить вычислительную нагрузку. Обеспечить возможность задавать размерность матриц и количество потоков при запуске программы. Многопоточность реализовать несколькими способами.
1) С использованием библиотеки стандарта POSIX Threads.
2) С использованием библиотеки стандарта OpenMP.
3) * С использованием библиотеки Intel TBB.
4) ** С использованием библиотеки стандарта MPI. Все матрицы помещаются в общей памяти одного вычислителя.
5) *** С использованием технологий многопоточности для графических сопроцессоров (GPU) - CUDA/OpenCL/OpenGL/OpenACC.
2. Для всех способов организации многопоточности построить график зависимости
коэффициента ускорения многопоточной программы от числа потоков для заданной
размерности матрицы, например, 5000, 10000 и 20000 элементов.
3. Определить оптимальное число потоков для вашего оборудования.
4. Подготовить отчет отражающий суть, этапы и результаты проделанной работы.

## Результаты работы

| Multiplication type | Launch count | Matrix size | Block size | Threads Cnt | Timer | Average time | Abs Error | Rel Error |
|---|---|---|---|---|---|---|---|---|
| POSIX_Threads | 5 | 5000 | 64 | 1 | clock_gettime() | 1758.38 | 211.76 | 2.550189e+03% |
| OpenMP | 5 | 5000 | 64 | 1 | clock_gettime() | 1438.14 | 365.95 | 9.311886e+03% |
| POSIX_Threads | 5 | 5000 | 64 | 2 | clock_gettime() | 763.98 | 161.53 | 3.415078e+03% |
| OpenMP | 5 | 5000 | 64 | 2 | clock_gettime() | 557.87 | 15.78 | 4.465234e+01% |
| POSIX_Threads | 5 | 5000 | 64 | 3 | clock_gettime() | 993.61 | 176.01 | 3.117999e+03% |
| OpenMP | 5 | 5000 | 64 | 3 | clock_gettime() | 443.03 | 105.11 | 2.493588e+03% |
| POSIX_Threads | 5 | 5000 | 64 | 4 | clock_gettime() | 549.79 | 102.68 | 1.917517e+03% |
| OpenMP | 5 | 5000 | 64 | 4 | clock_gettime() | 277.39 | 2.9 | 3.028769e+00% |
| POSIX_Threads | 5 | 5000 | 64 | 5 | clock_gettime() | 278.66 | 1.27 | 5.799782e-01% |
| OpenMP | 5 | 5000 | 64 | 5 | clock_gettime() | 227.89 | 2.29 | 2.299543e+00% |
| POSIX_Threads | 5 | 5000 | 64 | 6 | clock_gettime() | 429 | 124.09 | 3.589112e+03% |
| OpenMP | 5 | 5000 | 64 | 6 | clock_gettime() | 428.32 | 26.94 | 1.694355e+02% |
| POSIX_Threads | 5 | 5000 | 64 | 7 | clock_gettime() | 311.18 | 50.21 | 8.101752e+02% |
| OpenMP | 5 | 5000 | 64 | 7 | clock_gettime() | 238.45 | 2.61 | 2.858565e+00% |
| POSIX_Threads | 5 | 5000 | 64 | 8 | clock_gettime() | 360.54 | 135.28 | 5.075636e+03% |
| OpenMP | 5 | 5000 | 64 | 8 | clock_gettime() | 425.83 | 0.81 | 1.557091e-01% |

Таблица 1. Данные для 5000 элементов матрицы

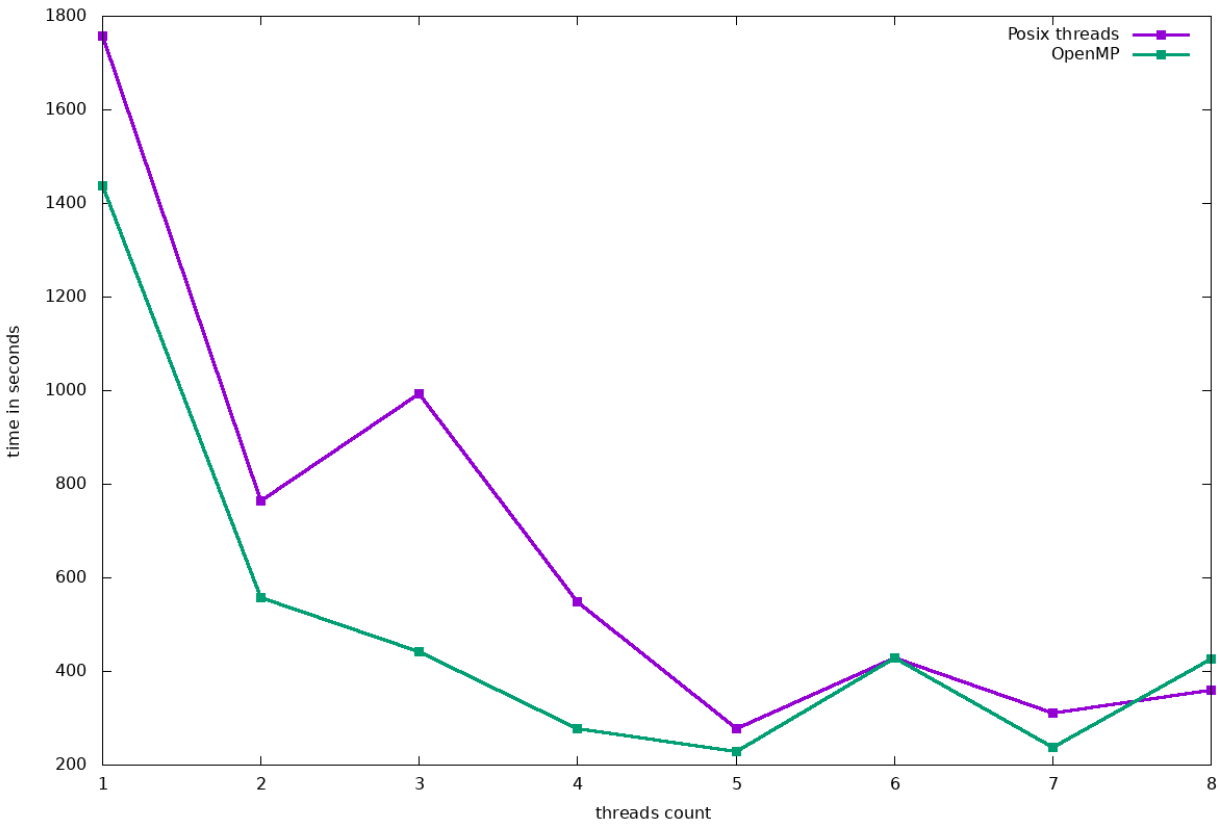


График 1. Зависимость времени выполнения от числа потоков на 5000 элементах

1. По данным из графика 1 видим, что увеличение числа потоков (threads count) обеспечивает ускорение, то есть уменьшение времени работы (time in seconds) программы при неизменном числе входных параметров.

   Видим, что метод Posix threads несколько менее эффективен в данном испытании, чем метод OpenMP.

   Помимо этого наблюдается скачкообразное поведение кривых, что связано с периодической загрузкой процессора иными задачами.

| Multiplication type | Launch count | Matrix size | Block size | Threads Cnt | Timer | Average time | Abs Error | Rel Error |
|---|---|---|---|---|---|---|---|---|
| POSIX_Threads | 5 | 7000 | 64 | 1 | clock_gettime() | 4073.57 | 12.11 | 3.601952e+00% |
| OpenMP | 5 | 7000 | 64 | 1 | clock_gettime() | 3186.56 | 11.13 | 3.888052e+00% |
| POSIX_Threads | 5 | 7000 | 64 | 2 | clock_gettime() | 2114.8 | 467.23 | 1.032274e+04% |
| OpenMP | 5 | 7000 | 64 | 2 | clock_gettime() | 1898.09 | 311.84 | 5.123406e+03% |
| POSIX_Threads | 5 | 7000 | 64 | 3 | clock_gettime() | 1570.04 | 347.74 | 7.701688e+03% |
| OpenMP | 5 | 7000 | 64 | 3 | clock_gettime() | 1047.18 | 44.05 | 1.852700e+02% |
| POSIX_Threads | 5 | 7000 | 64 | 4 | clock_gettime() | 991.06 | 13.1 | 1.732476e+01% |
| OpenMP | 5 | 7000 | 64 | 4 | clock_gettime() | 1049.57 | 326.01 | 1.012646e+04% |
| POSIX_Threads | 5 | 7000 | 64 | 5 | clock_gettime() | 993.91 | 309.9 | 9.662856e+03% |
| OpenMP | 5 | 7000 | 64 | 5 | clock_gettime() | 634.81 | 3.41 | 1.831741e+00% |
| POSIX_Threads | 5 | 7000 | 64 | 6 | clock_gettime() | 686.25 | 7.13 | 7.401548e+00% |
| OpenMP | 5 | 7000 | 64 | 6 | clock_gettime() | 554.41 | 4.27 | 3.282920e+00% |
| POSIX_Threads | 5 | 7000 | 64 | 7 | clock_gettime() | 804.86 | 4.1 | 2.090952e+00% |
| OpenMP | 5 | 7000 | 64 | 7 | clock_gettime() | 670.67 | 8.72 | 1.133082e+01% |
| POSIX_Threads | 5 | 7000 | 64 | 8 | clock_gettime() | 697.19 | 10.19 | 1.489610e+01% |
| OpenMP | 5 | 7000 | 64 | 8 | clock_gettime() | 582.18 | 8.62 | 1.276351e+01% |

Таблица 2. Данные для 7000 элементов матрицы

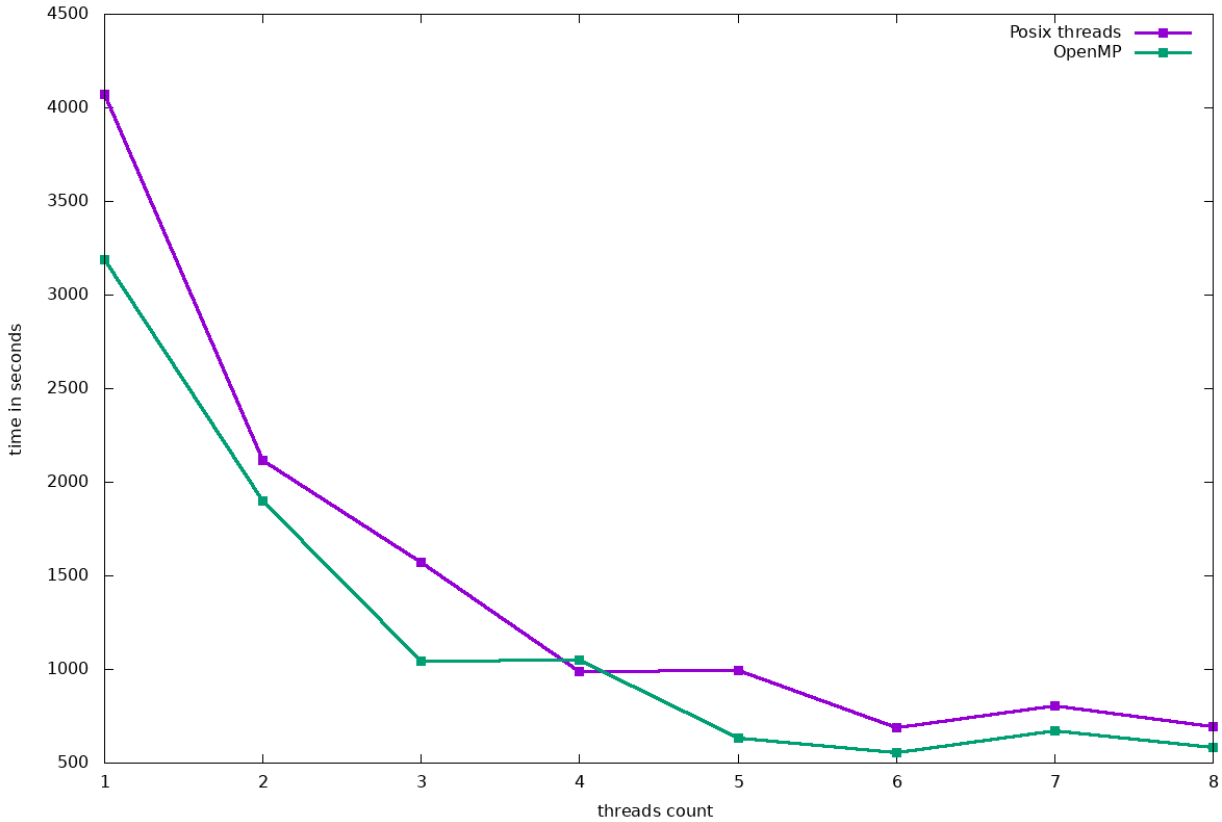


График 2. Зависимость времени выполнения от числа потоков на 7000 элементах

2. Из графика 2 видим, что при увеличении числа элементов в 1.4 раза, время выполнения последовательной программы методом Posix threads увеличивается примерно в 2.4 раза. В случае метода OpenMP время увеличится в 2.2 раза.

   В целом, видим, что распараллеливание методом OpenMP эффективнее, чем Posix threads.

| Multiplication type | Launch count | Matrix size | Block size | Threads count | Timer | Average time | Abs Error | Rel Error |
|---|---|---|---|---|---|---|---|---|
| usual | 3 | 1024 | 64 | 1 | clock_gettime() | 10.82 | 1.52 | 2.137567e+01% |
| row_by_row | 3 | 1024 | 64 | 1 | clock_gettime() | 5.32 | 0.54 | 5.413854e+00% |
| block | 3 | 1024 | 64 | 1 | clock_gettime() | 3.46 | 0.12 | 4.002582e-01% |
| POSIX_Threads | 3 | 1024 | 64 | 2 | clock_gettime() | 7.81 | 0.81 | 8.338165e+00% |
| OpenMP | 3 | 1024 | 64 | 2 | clock_gettime() | 5.69 | 0.23 | 9.625071e-01% |
| POSIX_Threads | 3 | 1024 | 64 | 8 | clock_gettime() | 3.74 | 0.08 | 1.758108e-01% |
| OpenMP | 3 | 1024 | 64 | 8 | clock_gettime() | 2.9 | 0.13 | 6.140194e-01% |

Таблица 3. Сравнение типов умножения матрицы при фиксированном числе элементов.

3. Сравнив все типы умножения матриц, видим, что параллельная программа в лучшем случае (8 потоков) уменьшает время выполнения программы для метода Posix threads относительно usual примерно в 2.9 раз,  для метода OpenMP — примерно в 3.7 раза.

## Листинг

Foo.cpp

```
#include "foo.h"

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

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

        else if (strcmp("-tc", argv[i]) == 0 ||
                 strcmp("--threads-count", argv[i]) == 0)
        {
            i++;
            threadsCnt = atoi(argv[i]);
            if (threadsCnt <= 0) {
                printf("Error in arguments of main(): incorrect value for --
threads-count \n");
                return 1;
            }
        }
    }

    return 0;
}

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';

        }
    }

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

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

int PrintMatrix(double *matrix, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%.2f ", matrix[i * n + j]);
        }
        printf("\n");
    }
    return 0;
}

void* MatrixMulForThread(void* voidpArgs)
{
    argsForThread* pArgs = (argsForThread*) voidpArgs;

    for (int i = pArgs->from; i < pArgs->to; i++) {
        for (int j = 0; j < pArgs->matrixSize; j++)
        for (int k = 0; k < pArgs->matrixSize; k++) {
            pArgs->matrixRes[i * pArgs->matrixSize + j]  +=
                pArgs->matrix1[i * pArgs->matrixSize + k] *
                pArgs->matrix2[k * pArgs->matrixSize + j];
        }
    }

    return 0;
}
```

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

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

    if (bCheck) for (i = 0; i < matrixSize*matrixSize; i++) {
        matrix1[i] = i;
        matrix2[i] = i;
        matrixRes[i] = 0;
    }
    else 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;
    }

    if (bCheck) {
        printf("\nmatrix1: \n");
        PrintMatrix(matrix1, matrixSize);
        printf("\nmatrix2: \n");
        PrintMatrix(matrix2, matrixSize);
        printf("\n");
    }

    struct timespec mt1, mt2;
    clock_gettime(CLOCK_REALTIME, &mt1);

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

            matrixRes[i * matrixSize + j]  +=
                matrix1[i * matrixSize + k] * matrix2[k * matrixSize + j];
        }
    }
    if (mulType_i == 2) {
        for (i = 0; i < matrixSize; i++)
        for (k = 0; k < matrixSize; k++)
        for (j = 0; j < matrixSize; j++) {
            matrixRes[i * matrixSize + j]  +=
                matrix1[i * matrixSize + k] * matrix2[k * matrixSize + j];
        }
    }
    if (mulType_i == 3) {
        double *m1, *m2, *mRes;
```

```
        int 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];
                }
            }
        }
    }
    if (mulType_i == 4) { /
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        argsForThread** pArgs = (argsForThread**) malloc(threadsCnt *
sizeof(argsForThread*));
        pthread_t thread_id[threadsCnt];
        int statuses[threadsCnt];
        int statuses_sum = 0;

        for (int i = 0; i < threadsCnt; i++) {
            pArgs[i] = (argsForThread*) malloc(sizeof(argsForThread));
            pArgs[i]->matrix1 = matrix1;
            pArgs[i]->matrix2 = matrix2;
            pArgs[i]->matrixRes = matrixRes;
            pArgs[i]->matrixSize = matrixSize;
            pArgs[i]->from = matrixSize / threadsCnt * i;
            pArgs[i]->to = matrixSize / threadsCnt * (i + 1);

            pthread_create(&thread_id[i], &attr, MatrixMulForThread,
pArgs[i]);
        }
        for (int i = 0; i < threadsCnt; i++) {

            statuses[i] = pthread_join(thread_id[i], NULL);
            statuses_sum += statuses[i];
        }

        if (statuses_sum != 0)
            printf("error, MatrixMulForThread() failed \n");

        for (int i = 0; i < threadsCnt; i++) {
            free(pArgs[i]);
        }
        free(pArgs);
```

```
    }
    if (mulType_i == 5) {
#pragma omp parallel for shared(matrix1, matrix2, matrixRes) \
private(j, k) num_threads(threadsCnt) schedule(static)
        for (i = 0; i < matrixSize; i++) {
            for (j = 0; j < matrixSize; j++) {

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

            }
        }
    }

    if (bCheck) {
        printf("\nmatrixRes: \n");
        PrintMatrix(matrixRes, matrixSize);
    }

    clock_gettime(CLOCK_REALTIME, &mt2);
    time_d = (double)(mt2.tv_sec - mt1.tv_sec) +
             (double)(mt2.tv_nsec - mt1.tv_nsec) / 1e9;

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

int WriteToCSV(char* mulType, int launchCnt, int matrixSize,
               int blockSize, int threadsCnt,
               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, "%d;", launchCnt);
    fprintf(fout, "%d;", matrixSize);
    fprintf(fout, "%d;", blockSize);
    fprintf(fout, "%d;", threadsCnt);
    fprintf(fout, "clock_gettime();"); ///Timer
    fprintf(fout, "%e;", avgTime);
    fprintf(fout, "%e;", absError);
    fprintf(fout, "%e%%;", relError);
```

```
    fprintf(fout, "\n");

    return 0;
}

int TestsHandler(char* mulType, int launchCnt,
                 int matrixSize, bool bCheck,
                 int blockSize, int &threadsCnt)
{
    double summand1 = 0, summand2 = 0;
    double time_d[launchCnt];
    for (int i = 0; i < launchCnt; i++) {
        if (strcmp("usual", mulType) == 0) {
            MatrixMul(1, blockSize, matrixSize, bCheck, time_d[i],
threadsCnt);
        }
        if (strcmp("row_by_row", mulType) == 0) {
            MatrixMul(2, blockSize, matrixSize, bCheck, time_d[i],
threadsCnt);
        }
        if (strcmp("block", mulType) == 0) {
            MatrixMul(3, blockSize, matrixSize, bCheck, time_d[i],
threadsCnt);
        }
        if (strcmp("POSIX_Threads", mulType) == 0) {
            MatrixMul(4, blockSize, matrixSize, bCheck, time_d[i],
threadsCnt);
        }
        if (strcmp("OpenMP", mulType) == 0) {
            MatrixMul(5, blockSize, matrixSize, bCheck, time_d[i],
threadsCnt);
        }
        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, threadsCnt,
avgTime, absError, relError);
    return 0;
}
```

# main.cpp

```
#include "foo.h"

int main(int argc, char *argv[]) {
    srand(time(0));
    int matrixSize = 100;
    char* mulType = (char*) malloc(15);
    strcpy(mulType, "usual\0");
    int launchCnt = 3;
    bool bCheck = false;
    int blockSize = GetCacheAlignment();
    int threadsCnt = 1;
    ProcessParameters(argc, argv, matrixSize, mulType,
                      launchCnt, bCheck, blockSize, threadsCnt);

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

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

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