*Кошманский Андрей*

**Результаты анализа производительности перемножения симметричной матрицы на верхне-треугольную с использованием различных оптимизаций**

Размерность матриц (n) – **1440**.

Количество экспериментов (для определения среднего времени операции вычисления перемножения матриц) – **8**.

Матрицы хранятся в векторном формате.

Используемый язык программирования – **C++**.

Используемый компилятор - **clang-omp++**:

* версия clang: 3.5.0;
* целевая процессорная архитектура: x86\_64-apple-darwin15.3.0;
* потоковая модель: posix.

Спецификация CPU, на котором выполнялись расчеты:

* наименование: Intel Core i5;
* количество процессоров: 1;
* общее количество ядер: 2 (вместе с виртуальными - 4);
* частота: 2.5 GHz;
* кэш L1: 32 Кб;
* кэш 2-го уровня: 256 Кб;
* кэш L3: 3 Мб.

Оперативная память – 4 Гб 1600 МГц DDR3.

Операционная система – OS X El Capitan 10.11.3.

**Результаты вычислений**

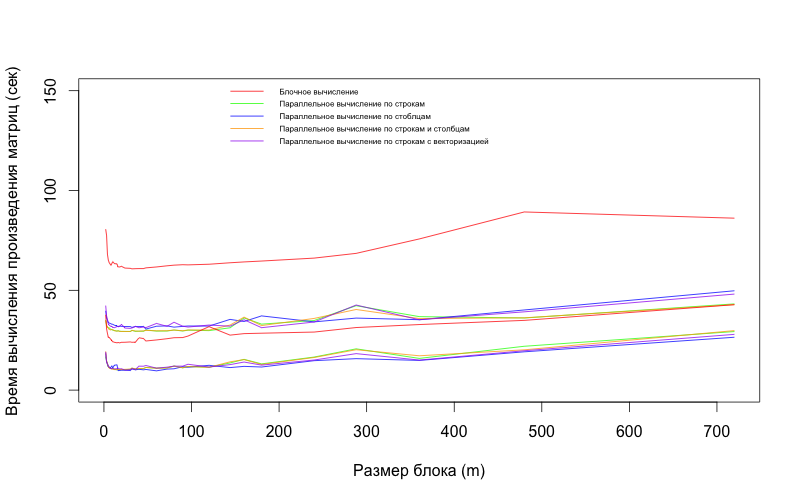
Без блочного вычисления и распараллеливания – 103.350 секунд; без блочного вычисления и распараллеливания, но с включённой директивой компилятора –O3 – 35.131 секунд.

Ниже в таблице представлены результаты блочных вычислений произведения матриц (с возможными распараллеливаниями и оптимизациями и без их использования). Скорости вычислений представлены в секундах (среднее значение всех экспериментов).

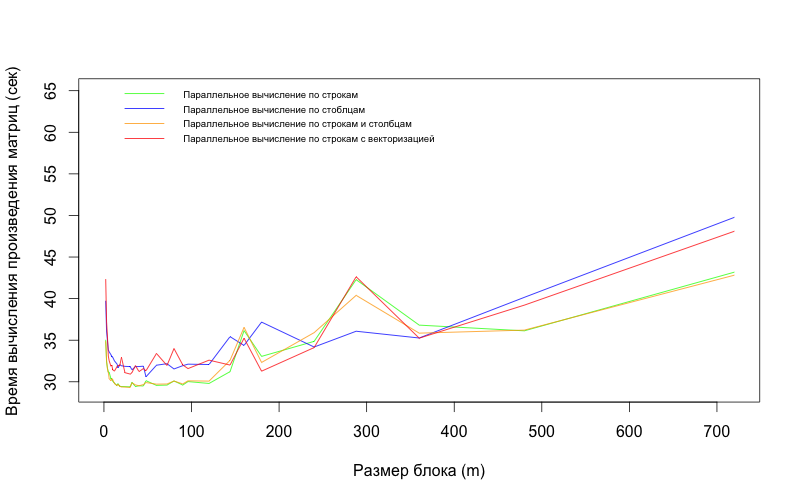
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Размер блока (m)** | **Блочное вычисление без оптимизаций** | **Параллельное вычисление по строкам матриц** | **Параллельное вычисление по столбцам матриц** | **Параллельное вычисление по строкам и столбцам матриц** | **Параллельное вычисление по строкам матриц с векторизацией** | **Блочное вычисление без распараллеливания (с директивой компилятора -O3)** | **Параллельное вычисление по строкам матриц с директивой -O3** | **Параллельное вычисление по столбцам матриц с директивой -O3** | **Параллельное вычисление по строкам и столбцам матриц с директивой -O3** | **Параллельное вычисление по строкам матриц с векторизацией и включенной директивой -O3** |
| 2 | 80.628 | 35.051 | 39.74 | 34.884 | 42.349 | 37.766 | 16.771 | 18.439 | 19.417 | 19.15 |
| 3 | 77.49 | 32.362 | 36.063 | 32.591 | 37.254 | 30.706 | 14.447 | 14.365 | 14.832 | 14.106 |
| 4 | 68.035 | 31.657 | 34.966 | 31.537 | 35.415 | 28.404 | 13.258 | 12.855 | 13.082 | 13.235 |
| 5 | 65.115 | 31.159 | 33.824 | 31.115 | 33.093 | 26.467 | 11.633 | 11.766 | 11.948 | 12.083 |
| 6 | 63.864 | 31.128 | 33.598 | 30.373 | 32.514 | 26.392 | 11.235 | 11.473 | 11.057 | 11.535 |
| 8 | 62.503 | 30.364 | 33.252 | 30.124 | 31.873 | 25.316 | 10.782 | 11.373 | 10.753 | 10.577 |
| 9 | 63.409 | 30.357 | 33.021 | 30.384 | 31.994 | 24.715 | 10.883 | 12.207 | 10.52 | 10.844 |
| 10 | 64.33 | 30.083 | 32.958 | 30.259 | 31.444 | 24.278 | 10.69 | 11.059 | 10.428 | 10.92 |
| 12 | 63.453 | 29.829 | 32.515 | 29.785 | 31.29 | 23.954 | 10.419 | 12.574 | 10.291 | 10.62 |
| 15 | 63.091 | 29.542 | 32.12 | 29.546 | 31.84 | 23.683 | 10.163 | 12.53 | 10.252 | 10.762 |
| 16 | 61.764 | 29.781 | 31.675 | 29.693 | 31.979 | 23.864 | 10.317 | 9.753 | 10.423 | 11.04 |
| 18 | 61.703 | 29.468 | 32.065 | 29.46 | 31.907 | 23.683 | 10.168 | 9.863 | 10.318 | 10.639 |
| 20 | 61.997 | 29.43 | 31.931 | 29.381 | 32.928 | 23.933 | 10.249 | 9.962 | 10.322 | 10.69 |
| 24 | 61.137 | 29.417 | 31.855 | 29.366 | 31.096 | 23.969 | 10.258 | 9.954 | 10.41 | 10.048 |
| 30 | 61.009 | 29.389 | 31.823 | 29.324 | 30.926 | 24.127 | 10.427 | 9.812 | 10.516 | 10.251 |
| 32 | 60.75 | 29.933 | 31.435 | 29.88 | 31.108 | 23.958 | 10.981 | 10.615 | 11.023 | 10.749 |
| 36 | 60.836 | 29.426 | 31.831 | 29.629 | 31.945 | 24.036 | 10.596 | 10.16 | 10.639 | 10.352 |
| 40 | 60.877 | 29.522 | 31.833 | 29.518 | 31.226 | 26.213 | 10.674 | 10.431 | 10.819 | 11.992 |
| 45 | 60.88 | 29.492 | 31.878 | 29.666 | 31.585 | 25.83 | 10.753 | 10.157 | 10.828 | 11.972 |
| 48 | 61.244 | 30.133 | 30.599 | 29.905 | 31.32 | 24.613 | 11.348 | 10.342 | 11.323 | 12.317 |
| 60 | 61.694 | 29.566 | 31.994 | 29.716 | 33.4 | 25.13 | 10.995 | 9.651 | 10.983 | 10.99 |
| 72 | 62.267 | 29.602 | 32.174 | 29.736 | 31.96 | 25.744 | 11.086 | 10.523 | 11.086 | 11.494 |
| 80 | 62.598 | 30.101 | 31.543 | 30.09 | 33.998 | 26.244 | 12.063 | 10.687 | 11.995 | 12.011 |
| 90 | 62.821 | 29.608 | 31.937 | 29.743 | 32.023 | 26.33 | 11.168 | 11.736 | 11.196 | 12.005 |
| 96 | 62.739 | 30.025 | 32.119 | 30.116 | 31.587 | 27.08 | 11.789 | 11.45 | 11.827 | 12.951 |
| 120 | 63.057 | 29.799 | 32.075 | 30.08 | 32.601 | 31.799 | 11.463 | 12.421 | 11.404 | 11.545 |
| 144 | 63.793 | 31.219 | 35.428 | 32.581 | 32.011 | 27.543 | 13.603 | 11.28 | 14.172 | 12.766 |
| 160 | 64.217 | 36.157 | 34.37 | 36.544 | 35.239 | 28.296 | 15.401 | 11.891 | 15.281 | 14.061 |
| 180 | 64.659 | 33.055 | 37.177 | 32.325 | 31.276 | 28.507 | 13.141 | 11.567 | 12.875 | 12.516 |
| 240 | 66.131 | 34.859 | 34.181 | 35.908 | 34.112 | 29.077 | 16.555 | 14.741 | 16.329 | 15.068 |
| 288 | 68.505 | 42.282 | 36.078 | 40.391 | 42.627 | 31.344 | 20.675 | 15.756 | 20.231 | 18.277 |
| 360 | 75.72 | 36.813 | 35.264 | 35.849 | 35.244 | 32.84 | 15.876 | 14.867 | 17.191 | 15.043 |
| 480 | 89.248 | 36.125 | 40.152 | 36.213 | 39.213 | 34.929 | 21.962 | 19.182 | 20.181 | 19.727 |
| 720 | 86.11 | 43.183 | 49.774 | 42.819 | 48.104 | 42.796 | 29.342 | 26.485 | 29.827 | 27.926 |

**Графическое представление результатов вычисления**

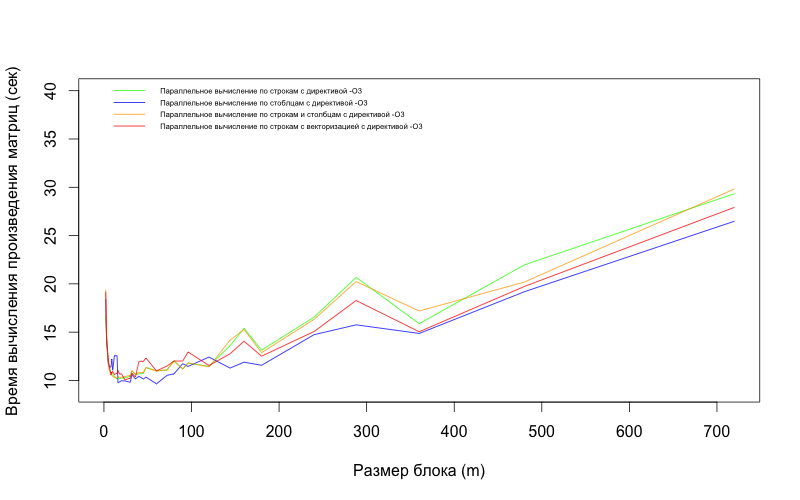
На рисунке ниже представлены результаты двух видов вычислений: без оптимизаций компилятора (выше) и с использованием директивы компилятора –O3 (ниже):



На данном ниже рисунке представлены результаты вычислений произведения матриц с использованием различных способов распараллеливания:



И, наконец, представляем на рисунке результаты вычислений произведения матриц с использованием различных способов распараллеливания с применеием оптимизаций компилятора –O3:



**Заключение:** наилучшие результаты в производительности (времени вычисления) нахождения произведения матриц показывает параллельное вычисление (по столбцам) с включенной директивой компилятора –O3 и размером блока в 60 матричных элементов – 9.651 секунд, а также размером блока в 16 элементов – 9.753 секунд.

Подобное наблюдение можно объяснить тем, что при размере блока в 16 элементов общее количество памяти, требуемое на перемножение строки на столбец, равно: 16 (число элементов) \* 2 (строка и столбец перемножаемых матриц) \* 1 Кб (для хранения переменной типа double в памяти) = 32 Кб. Таким образом, одна итерация (вычисление одного элемента в матрице, представляющей собой результат произведения) полностью умещается в самой быстрой кэш-памяти (первого уровня) процессора, что позволяет минимизировать количество наиболее дорогостоящих (в отношении времени выполнения вычислений) операций чтения данных.

**Приложение: листинг кода реализованной программы блочного вычисления произведения матриц**

#include <iostream>

#include <string>

#include <chrono>

#include <omp.h>

#include <fstream>

using namespace std;

//! Describes side's size of used matrix.

const int CONFIG\_MATRIX\_SIZE = 1440; // 2880

const double CONFIG\_MAX\_M\_VAL = 1000000; // range of matrix values

const double CONFIG\_MIN\_M\_VAL = -1000000;

const int CONFIG\_INITIAL\_BLOCK\_SIZE = 2; // inital block size and its change

const int CONFIG\_STEP\_TO\_CHANGE\_BLOCK = 1;

const int CONFIG\_MAX\_BLOCK\_SIZE = 1000;

const int CONFIG\_NUMBER\_OF\_EXPERIMENTS = 8; // number of evals to find mean

const string CONFIG\_LOG\_PATH = "logs/"; // where to store evaluations results

#define EXP\_START totalTime = 0; \

for (int i = 0; i < CONFIG\_NUMBER\_OF\_EXPERIMENTS; i++) \

{ \

fill\_n(C, squareItemsCount, 0); \

setTimerStart();

#define EXP\_END totalTime += getElapsedTimerTime(); \

}

#define BLOCK\_IT\_BEGIN blocksSize = CONFIG\_INITIAL\_BLOCK\_SIZE; \

while (blocksSize < CONFIG\_MAX\_BLOCK\_SIZE) \

{ \

if (CONFIG\_MATRIX\_SIZE % blocksSize == 0) \

{

#define BLOCK\_IT\_END } blocksSize += CONFIG\_STEP\_TO\_CHANGE\_BLOCK; }

//! Generates some random value in range.

double genRandom(double from, double to)

{

double f = (double) rand() / RAND\_MAX;

return (from + f \* (to - from));

}

//! Evaluates the number of elements in the triangular matrix with provided size.

int evalItemsCount(int size)

{

int k = 0;

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

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

k++;

return k;

}

//! Fills provided matrix as a vector.

void fillM(int elements, double \* matrix)

{

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

matrix[i] = genRandom(CONFIG\_MIN\_M\_VAL, CONFIG\_MAX\_M\_VAL);

}

//! Recursive func to convert square index to the vector's type.

int getVectorIndexOfLTM(int i, int j)

{

if (i == 0)

return 0;

return i + j + getVectorIndexOfLTM(i - 1, 0);

}

//! Converts matrix's indexes to vector's. Non-recursive.

int getFastVectorIndexOfLTM(int i, int j, int sizeArrFormat)

{

if (i == 0)

return 0;

return i \* (sizeArrFormat + 1) // square of the previous rect

// Skipped empties (rect + triangle).

- ((sizeArrFormat - i) \* i // rect

+ (int) (0.5 \* (i + 0.5) \* (i + 0.5))) // triangle

+ j; // current index shift

}

//! Reads a value from LTM vector with full index as if it was a symmetric matrix.

double getValueOfSymmetricLTM(double \* matrix, int i, int j, int sizeArrFormat)

{

if (j > i)

return matrix[getFastVectorIndexOfLTM(j, i, sizeArrFormat)];

else

return matrix[getFastVectorIndexOfLTM(i, j, sizeArrFormat)];

}

//! Recursive func to get vector index from matrix-type.

int getVectorIndexOfUTM(int i, int j, int sizeArrFormat)

{

if (i == 0)

return j;

return getVectorIndexOfUTM(i - 1, sizeArrFormat, sizeArrFormat) + j - (i - 1);

}

//! Fast implementation of matrix coords to vector convertion.

int getFastVectorIndexOfUTM(int i, int j, int sizeArrFormat)

{

return (i + 1) \* (sizeArrFormat + 1) // square of the total rect

- 1 // required shift

- (sizeArrFormat - j) // right shift

- (int) (0.5 \* (i + 0.5) \* (i + 0.5)); // full excluded points

}

//! Reads a value from UTM vector.

double getValueOfUTM(double \* matrix, int i, int j, int sizeArrFormat)

{

if (i > j)

return 0;

return matrix[getFastVectorIndexOfUTM(i, j, sizeArrFormat)];

}

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

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

void setTimerStart()

{

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

}

long getElapsedTimerTime()

{

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

return chrono::duration\_cast<chrono::nanoseconds>(finish - start).count();

}

//! Implements multiplication of two matrix. No any optimizations.

void simpleMultM(double \* A, double \* B, double \* C, int n)

{

int sizeArrFormat = n - 1;

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

{

int lineIndex = i \* n;

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

{

int totalLineIndex = lineIndex + j;

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

C[totalLineIndex] += (getValueOfSymmetricLTM(A, i, k, sizeArrFormat) \* getValueOfUTM(B, k, j, sizeArrFormat));

}

}

}

//! Implements multiplication of two matrix using nested blocks.

void blocksMultM(double \* A, double \* B, double \* C, int n, int m)

{

int sizeArrFormat = n - 1;

int blocksCount = n / m;

// Iterating throw blocks.

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

{

int im = i \* m;

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

{

int jm = j \* m;

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

{

int km = k \* m;

// Iterating throw blocks' elements.

for (int ii = 0; ii < m; ii++)

{

int imii = im + ii;

int lineIndex = (imii) \* n;

for (int jj = 0; jj < m; jj++)

{

int jmjj = jm + jj;

int totalLineIndex = lineIndex + jmjj;

for (int kk = 0; kk < m; kk++)

{

int kmkk = km + kk;

C[totalLineIndex] +=

getValueOfSymmetricLTM(A, imii, kmkk, sizeArrFormat) \*

getValueOfUTM(B, kmkk, jmjj, sizeArrFormat);

}

}

}

}

}

}

}

//! Implements multiplication of two matrix using nested blocks and also tries to make computations in parallel mode.

void parallelizedMainBlocksMultM(double \* A, double \* B, double \* C, int n, int m)

{

int sizeArrFormat = n - 1;

int blocksCount = n / m;

// Iterating throw blocks.

#pragma omp parallel for

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

{

int im = i \* m;

#pragma omp parallel for

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

{

int jm = j \* m;

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

{

int km = k \* m;

// Iterating throw blocks' elements.

for (int ii = 0; ii < m; ii++)

{

int imii = im + ii;

int lineIndex = (imii) \* n;

for (int jj = 0; jj < m; jj++)

{

int jmjj = jm + jj;

int totalLineIndex = lineIndex + jmjj;

// Very bad idea to make parallelizations here.

for (int kk = 0; kk < m; kk++)

{

int kmkk = km + kk;

C[totalLineIndex] +=

getValueOfSymmetricLTM(A, imii, kmkk, sizeArrFormat) \*

getValueOfUTM(B, kmkk, jmjj, sizeArrFormat);

}

}

}

}

}

}

}

//! Implements multiplication of two matrix using nested blocks and also tries to make computations in parallel mode.

void parallelizedMainLinesBlocksMultM(double \* A, double \* B, double \* C, int n, int m)

{

int sizeArrFormat = n - 1;

int blocksCount = n / m;

// Iterating throw blocks.

#pragma omp parallel for

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

{

int im = i \* m;

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

{

int jm = j \* m;

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

{

int km = k \* m;

// Iterating throw blocks' elements.

for (int ii = 0; ii < m; ii++)

{

int imii = im + ii;

int lineIndex = (imii) \* n;

for (int jj = 0; jj < m; jj++)

{

int jmjj = jm + jj;

int totalLineIndex = lineIndex + jmjj;

// Very bad idea to make parallelizations here.

for (int kk = 0; kk < m; kk++)

{

int kmkk = km + kk;

C[totalLineIndex] +=

getValueOfSymmetricLTM(A, imii, kmkk, sizeArrFormat) \*

getValueOfUTM(B, kmkk, jmjj, sizeArrFormat);

}

}

}

}

}

}

}

//! Implements multiplication of two matrix using nested blocks and also tries to make computations in parallel mode.

void parallelizedMainColsBlocksMultM(double \* A, double \* B, double \* C, int n, int m)

{

int sizeArrFormat = n - 1;

int blocksCount = n / m;

// Iterating throw blocks.

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

{

int im = i \* m;

#pragma omp parallel for

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

{

int jm = j \* m;

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

{

int km = k \* m;

// Iterating throw blocks' elements.

for (int ii = 0; ii < m; ii++)

{

int imii = im + ii;

int lineIndex = (imii) \* n;

for (int jj = 0; jj < m; jj++)

{

int jmjj = jm + jj;

int totalLineIndex = lineIndex + jmjj;

// Very bad idea to make parallelizations here.

for (int kk = 0; kk < m; kk++)

{

int kmkk = km + kk;

C[totalLineIndex] +=

getValueOfSymmetricLTM(A, imii, kmkk, sizeArrFormat) \*

getValueOfUTM(B, kmkk, jmjj, sizeArrFormat);

}

}

}

}

}

}

}

//! Implements multiplication of two matrix using nested blocks and also tries to make computations in parallel mode.

void parallelizedNestedLinesBlocksMultM(double \* A, double \* B, double \* C, int n, int m)

{

int sizeArrFormat = n - 1;

int blocksCount = n / m;

// Iterating throw blocks.

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

{

int im = i \* m;

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

{

int jm = j \* m;

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

{

int km = k \* m;

// Iterating throw blocks' elements.

#pragma omp parallel for

for (int ii = 0; ii < m; ii++)

{

int imii = im + ii;

int lineIndex = (imii) \* n;

for (int jj = 0; jj < m; jj++)

{

int jmjj = jm + jj;

int totalLineIndex = lineIndex + jmjj;

// Very bad idea to make parallelizations here.

for (int kk = 0; kk < m; kk++)

{

int kmkk = km + kk;

C[totalLineIndex] +=

getValueOfSymmetricLTM(A, imii, kmkk, sizeArrFormat) \*

getValueOfUTM(B, kmkk, jmjj, sizeArrFormat);

}

}

}

}

}

}

}

//! Implements multiplication of two matrix using nested blocks and also tries to make computations in parallel mode.

void parallelizedMainLinesVectorBlocksMultM(double \* A, double \* B, double \* C, int n, int m)

{

int sizeArrFormat = n - 1;

int blocksCount = n / m;

// Iterating throw blocks.

#pragma omp parallel for simd

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

{

int im = i \* m;

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

{

int jm = j \* m;

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

{

int km = k \* m;

// Iterating throw blocks' elements.

for (int ii = 0; ii < m; ii++)

{

int imii = im + ii;

int lineIndex = (imii) \* n;

for (int jj = 0; jj < m; jj++)

{

int jmjj = jm + jj;

int totalLineIndex = lineIndex + jmjj;

// Very bad idea to make parallelizations here.

for (int kk = 0; kk < m; kk++)

{

int kmkk = km + kk;

C[totalLineIndex] +=

getValueOfSymmetricLTM(A, imii, kmkk, sizeArrFormat) \*

getValueOfUTM(B, kmkk, jmjj, sizeArrFormat);

}

}

}

}

}

}

}

//! Adds some spaces before number, if needed.

const char \* outputFormatted(int number)

{

if (number < 10)

cout << " " << number;

else if (number < 100)

cout << " " << number;

else cout << number;

return "";

}

//! Outputs provided usual matrix into the console.

void printM(double \* matrix, int size)

{

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

{

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

{

cout << matrix[i \* size + j] << " ";

}

cout << endl;

}

}

//! Outputs provided symmetric LT matrix into the console.

void printSymmetricLTM(double \* matrix, int size)

{

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

{

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

{

cout << getValueOfSymmetricLTM(matrix, i, j, size - 1) << " ";

}

cout << endl;

}

}

//! Outputs provided UT matrix into the console.

void printUTM(double \* matrix, int size)

{

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

{

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

{

cout << getValueOfUTM(matrix, i, j, size - 1) << " ";

}

cout << endl;

}

}

//! Outputs all working matrix.

void printSourceAndResult(double \* A, double \* B, double \* C, int size)

{

cout << endl;

printSymmetricLTM(A, size);

cout << endl;

printUTM(B, size);

cout << endl;

printM(C, size);

cout << endl;

}

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

{

// To make different logs creating a dir.

string logFileNamePrefix = "default";

if (argc >= 2)

logFileNamePrefix = argv[1];

string pathToStoreLog = CONFIG\_LOG\_PATH + logFileNamePrefix;

system(("mkdir -p " + pathToStoreLog).c\_str());

pathToStoreLog += "/";

// Seeding random values.

srand(time(NULL));

// Saving number of elements in the triangular matrix with current size.

int itemsCount = evalItemsCount(CONFIG\_MATRIX\_SIZE);

// Size of the result matrix.

int squareItemsCount = CONFIG\_MATRIX\_SIZE \* CONFIG\_MATRIX\_SIZE;

// Generating source and result template matrix.

double \* A = new double[itemsCount]; fillM(itemsCount, A); // symmetric, stored as lower triangular

double \* B = new double[itemsCount]; fillM(itemsCount, B); // upper triangular

double \* C = new double[squareItemsCount]; // result matrix, rectangular

// Disclaimer.

cout << "Number of experiments for each method - " << CONFIG\_NUMBER\_OF\_EXPERIMENTS << ", matrix size - " << CONFIG\_MATRIX\_SIZE << endl;

// For further time estimations.

long totalTime;

// For further tiling.

int blocksSize;

// Simple multiplication.

EXP\_START

// Making simple multiplication of source matrix without any optimizations.

simpleMultM(A, B, C, CONFIG\_MATRIX\_SIZE);

EXP\_END

// Printing average eval time into the console.

cout << "Average time of the simple multiplication: " << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << endl;

// Saving result to the log file.

ofstream sf;

sf.open(pathToStoreLog + "simple.csv");

sf << "Experiments,Size(n),Time(ns)\n";

sf << CONFIG\_NUMBER\_OF\_EXPERIMENTS << "," << CONFIG\_MATRIX\_SIZE << "," << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << "\n";

sf.close();

// To check parallel computation results.

double \* checkC = new double[squareItemsCount];

copy(C, C + squareItemsCount, checkC);

// Blocks multiplication experiments.

cout << "No parallelization multiplications" << endl;

// Saving result to the log file.

ofstream bf;

bf.open(pathToStoreLog + "blocks.csv");

bf << "Experiments,Size(n),Blocksize(m),Time(ns)\n";

BLOCK\_IT\_BEGIN

EXP\_START

// Making nested blocks multiplication of source matrix without any optimizations.

blocksMultM(A, B, C, CONFIG\_MATRIX\_SIZE, blocksSize);

EXP\_END

// Printing average eval time.

cout << outputFormatted(blocksSize) << " blocks, time: " << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << endl;

// Writing result to file.

bf << CONFIG\_NUMBER\_OF\_EXPERIMENTS << "," << CONFIG\_MATRIX\_SIZE << "," << blocksSize << "," << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << "\n";

BLOCK\_IT\_END

bf.close();

// Main lines and columns parallelization experiments.

cout << "Using OpenMP pragma for main lines and columns" << endl;

// Saving result to the log file.

ofstream bmlcf;

bmlcf.open(pathToStoreLog + "blocks\_p\_main\_lines\_cols.csv");

bmlcf << "Experiments,Size(n),Blocksize(m),Time(ns)\n";

BLOCK\_IT\_BEGIN

EXP\_START

// Making nested blocks multiplication of source matrix with parallel computation optimization.

parallelizedMainBlocksMultM(A, B, C, CONFIG\_MATRIX\_SIZE, blocksSize);

EXP\_END

// Printing average eval time.

cout << outputFormatted(blocksSize) << " blocks, time: " << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << endl;

// Writing result to file.

bmlcf << CONFIG\_NUMBER\_OF\_EXPERIMENTS << "," << CONFIG\_MATRIX\_SIZE << "," << blocksSize << "," << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << "\n";

BLOCK\_IT\_END

bmlcf.close();

// Main lines parallelization experiments.

cout << "Using OpenMP pragma for main lines" << endl;

// Saving result to the log file.

ofstream bmlf;

bmlf.open(pathToStoreLog + "blocks\_p\_main\_lines.csv");

bmlf << "Experiments,Size(n),Blocksize(m),Time(ns)\n";

BLOCK\_IT\_BEGIN

EXP\_START

// Making nested blocks multiplication of source matrix with parallel computation optimization.

parallelizedMainLinesBlocksMultM(A, B, C, CONFIG\_MATRIX\_SIZE, blocksSize);

EXP\_END

// Printing average eval time.

cout << outputFormatted(blocksSize) << " blocks, time: " << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << endl;

// Writing result to file.

bmlf << CONFIG\_NUMBER\_OF\_EXPERIMENTS << "," << CONFIG\_MATRIX\_SIZE << "," << blocksSize << "," << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << "\n";

BLOCK\_IT\_END

bmlf.close();

// Main columns parallelization experiments.

cout << "Using OpenMP pragma for main columns" << endl;

// Saving result to the log file.

ofstream bmcf;

bmcf.open(pathToStoreLog + "blocks\_p\_main\_cols.csv");

bmcf << "Experiments,Size(n),Blocksize(m),Time(ns)\n";

BLOCK\_IT\_BEGIN

EXP\_START

// Making nested blocks multiplication of source matrix with parallel computation optimization.

parallelizedMainColsBlocksMultM(A, B, C, CONFIG\_MATRIX\_SIZE, blocksSize);

EXP\_END

// Printing average eval time.

cout << outputFormatted(blocksSize) << " blocks, time: " << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << endl;

// Writing result to file.

bmcf << CONFIG\_NUMBER\_OF\_EXPERIMENTS << "," << CONFIG\_MATRIX\_SIZE << "," << blocksSize << "," << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << "\n";

BLOCK\_IT\_END

bmcf.close();

// Nested lines parallelization experiments.

/\*

cout << "Using OpenMP pragma for nested lines" << endl;

// Saving result to the log file.

ofstream bnlf;

bnlf.open(pathToStoreLog + "blocks\_p\_nested\_lines.csv");

bnlf << "Experiments,Size(n),Blocksize(m),Time(ns)\n";

BLOCK\_IT\_BEGIN

EXP\_START

// Making nested blocks multiplication of source matrix with parallel computation optimization.

parallelizedNestedLinesBlocksMultM(A, B, C, CONFIG\_MATRIX\_SIZE, blocksSize);

EXP\_END

// Printing average eval time.

cout << outputFormatted(blocksSize) << " blocks, time: " << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << endl;

// Writing result to file.

bnlf << CONFIG\_NUMBER\_OF\_EXPERIMENTS << "," << CONFIG\_MATRIX\_SIZE << "," << blocksSize << "," << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << "\n";

BLOCK\_IT\_END

bnlf.close();

\*/

// Main lines parallelization experiments with vectorization.

cout << "Using OpenMP pragma for main lines with vectorization" << endl;

// Saving result to the log file.

ofstream bmlvf;

bmlvf.open(pathToStoreLog + "blocks\_p\_main\_lines\_vector.csv");

bmlvf << "Experiments,Size(n),Blocksize(m),Time(ns)\n";

BLOCK\_IT\_BEGIN

EXP\_START

// Making nested blocks multiplication of source matrix with parallel computation optimization.

parallelizedMainLinesVectorBlocksMultM(A, B, C, CONFIG\_MATRIX\_SIZE, blocksSize);

EXP\_END

// Printing average eval time.

cout << outputFormatted(blocksSize) << " blocks, time: " << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << endl;

// Writing result to file.

bmlvf << CONFIG\_NUMBER\_OF\_EXPERIMENTS << "," << CONFIG\_MATRIX\_SIZE << "," << blocksSize << "," << (totalTime / CONFIG\_NUMBER\_OF\_EXPERIMENTS) << "\n";

BLOCK\_IT\_END

bmlvf.close();

// Checking if parallel computation is correct.

bool failure = false;

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

if (C[i] != checkC[i])

{

failure = true;

break;

}

if (failure)

cout << "Parallel computations are invalid!" << endl;

// Releasing used memory.

delete[] A;

delete[] B;

delete[] C;

delete[] checkC;

return 0;

}