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

Черемисин И.И

Работу проверил:

Ассистент кафедры ВС

Петухова Я.В.

Новосибирск 2021

# Задание

1. На языке С/С++/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. Оценить ускорение умножения для матриц фиксированного размера, например, 1000х1000, 2000х2000, 5000х5000, 10000х10000.

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

7. С помощью профилировщика для исходной программы и каждого способа оптимизации доступа к памяти оценить количество промахов при работе к КЭШ памятью (cache-misses).

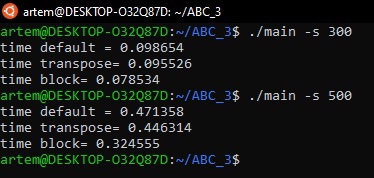
8. Подготовить отчет отражающий суть, этапы и результаты проделанной работы.

# Ход работы

1. После реализации функции DGEMM BLAS для последовательного умножения двух квадратных матриц с элементами типа double, провели серию испытаний и построили график зависимости времени выполнения программы от объёма входных данных:

График 1. Зависимость времени от размера матрицы

Предельный тип матрицы с точностью определить не смог, но явно предел выше 10000.



Ввод размерность матрицы в качестве аргумента командной строки при запуске программы.

1. Оценил ускорение умножения для матриц фиксированного размера:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **DGEMM** | **DGEMM\_OPT\_1** | **SPEEDUP OPT** |
| 400x400 | 0,202197 | 0,163072 | 1,23 |
| 800x800 | 2,181075 | 1,381345 | 1,58 |
| 1200x1200 | 4,722347 | 3,564118 | 1,32 |
| 1600x1600 | 17,713374 | 10,736066 | 1,65 |
| 2000x2000 | 32,28841 | 20,412221 | 1,58 |
| 5000x5000 | 49,49158 | 32,984185 | 1,50 |

Таблица 1. Ускорение умножения матриц.

Для блочного умножения матриц размер блока, при котором достигается максимальное ускорение равен 64.

1. С помощью профилировщика оценил количество промахов при работе с КЭШ-памятью (cache-misses):

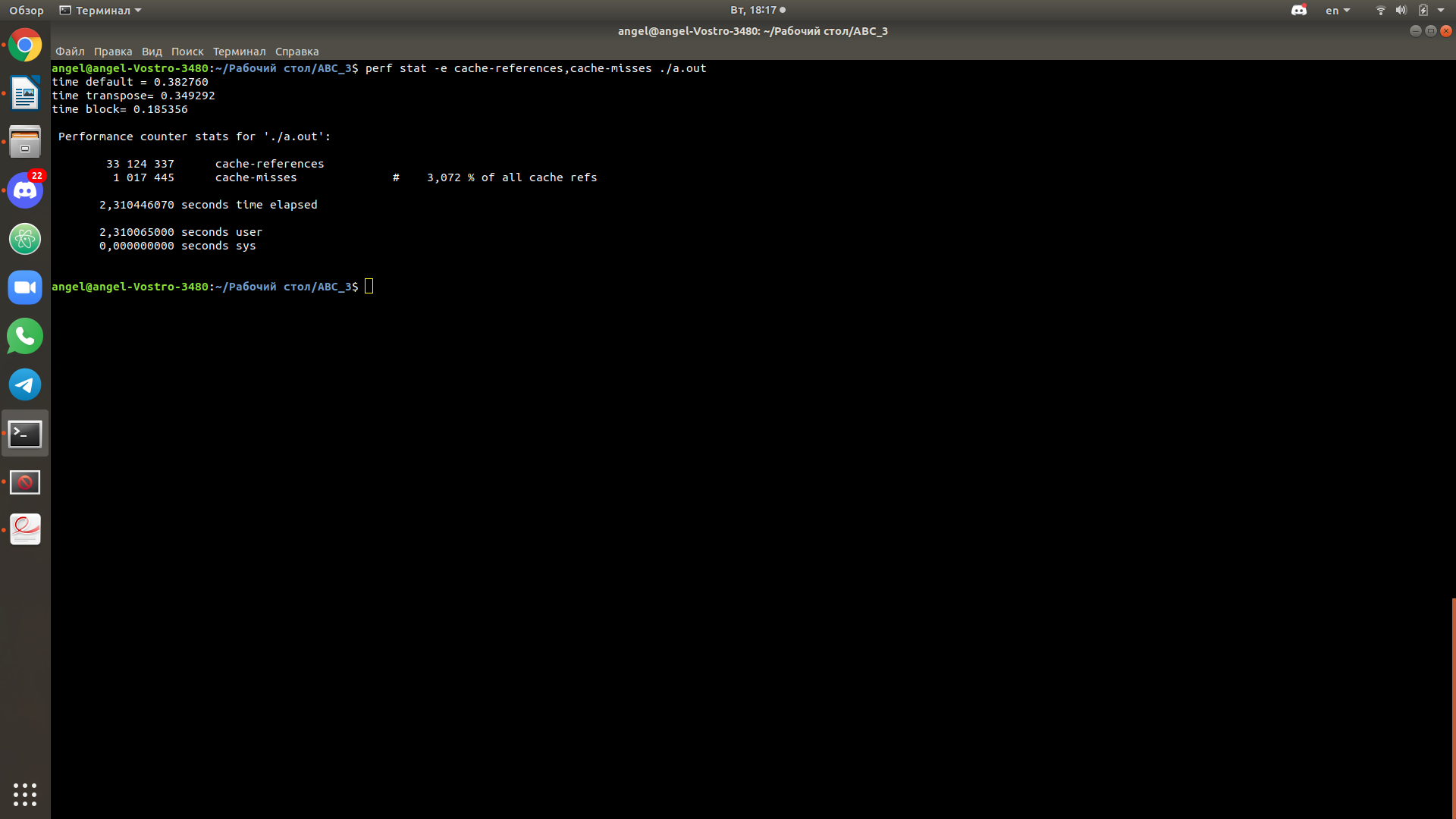


Рисунок 1. Количество промахов при обращении к КЕШ-памяти в неоптимизированной программе при размере матрицы 400x400.

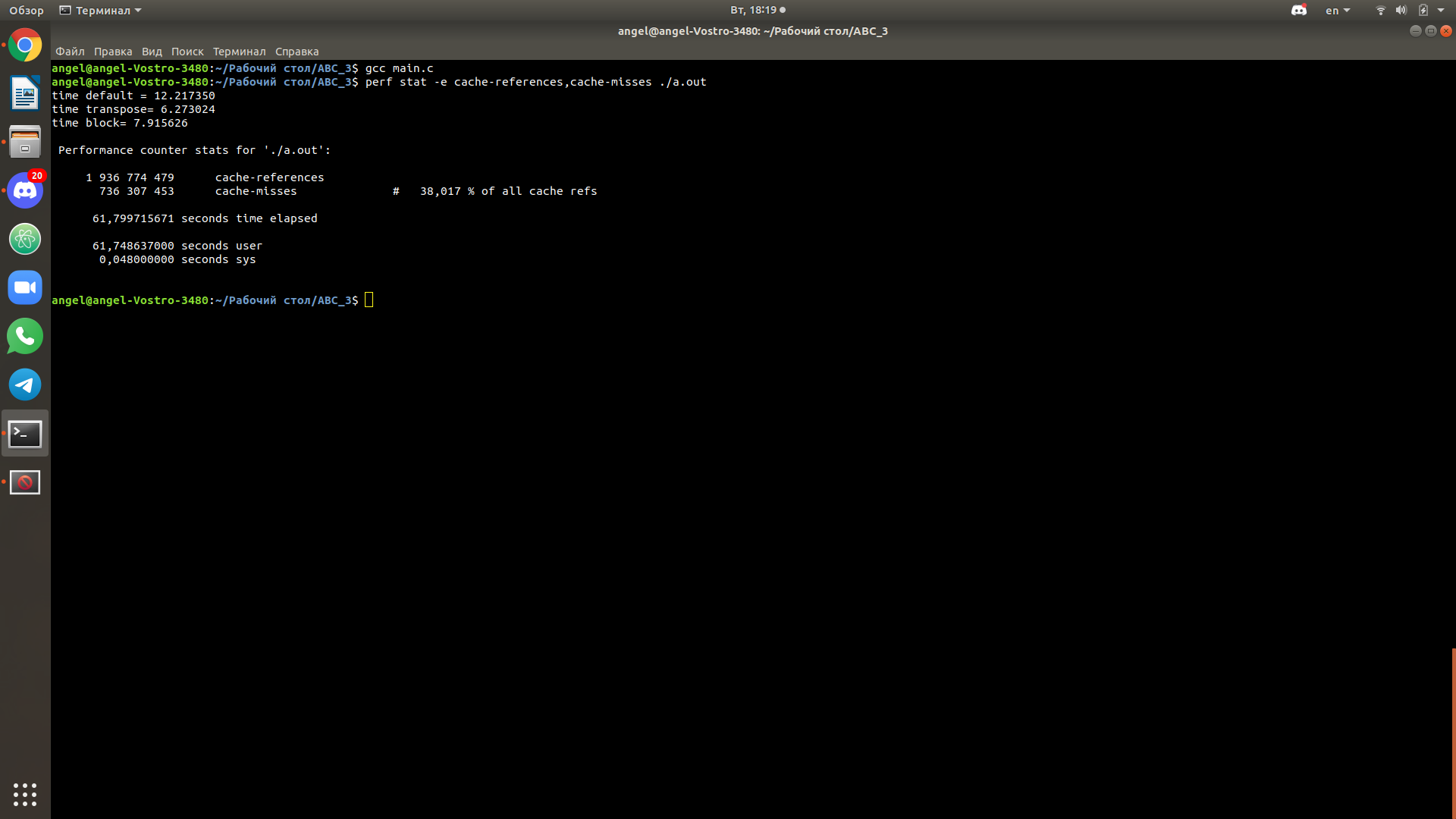


Рисунок 2. Количество промахов при обращении к КЕШ-памяти в неоптимизированной программе при размере матрицы 1200x1200.

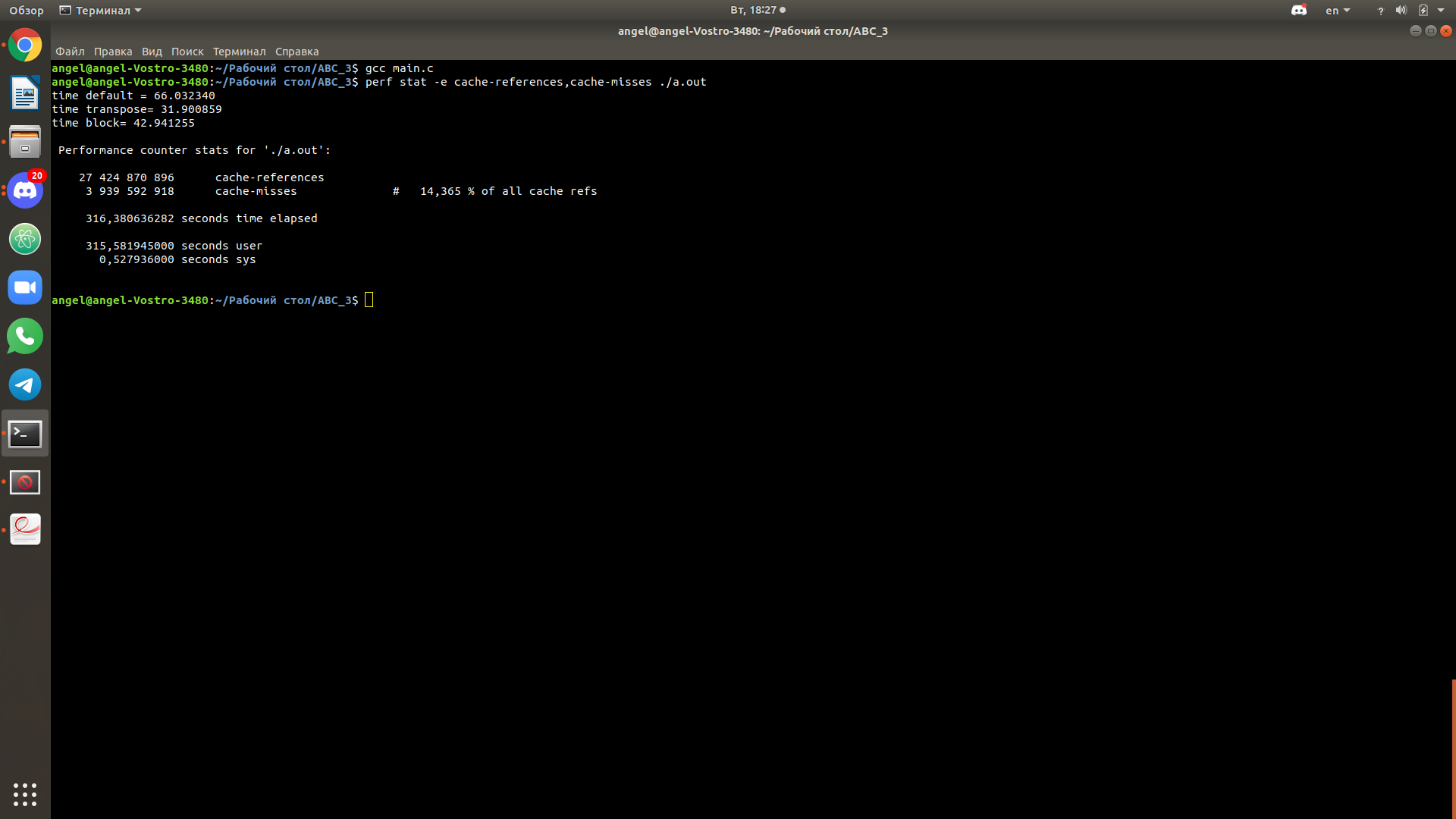


Рисунок 3. Количество промахов при обращении к КЕШ-памяти в неоптимизированной программе при размере матрицы 2000x2000.

# Листинг

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#define BS 64

**void** **print\_matrix**(**double** \*a, **int** SIZE);

**void** **dgemm\_default**(**double** \*a, **double** \*b, **double** \*c, **int** SIZE);

**void** **dgemm\_transpose**(**double** \*a, **double** \*b, **double** \*c, **int** SIZE);

**void** **dgemm\_block**(**double** \*a, **double** \*b, **double** \*c, **int** SIZE);

**void** **init\_matrix**(**double** \*a, **double** \*b, **double** \*c, **int** SIZE);

**void** **init\_matrix\_C**(**double** \*c, **int** SIZE);

**void** **print\_matrix**(**double** \*a, **int** SIZE);

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

{

**int** i;

**int** SIZE = **100**;

**for** (i = **0**; i < argc; i++)

{

**if** (strcmp("-s", argv[i]) == **0**)

{

i++;

SIZE = atoi(argv[i]);

}

}

**double** A[SIZE \* SIZE], B[SIZE \* SIZE], C[SIZE \* SIZE];

srand(time(**0**));

**struct** timespec mt1, mt2;

**long** **int** tt = **0**;

init\_matrix(A, B, C, SIZE);

clock\_gettime(CLOCK\_REALTIME, &mt1);

dgemm\_default(A, B, C, SIZE);

clock\_gettime(CLOCK\_REALTIME, &mt2);

tt = pow(**10**, **9**) \* (mt2.tv\_sec - mt1.tv\_sec) + (mt2.tv\_nsec - mt1.tv\_nsec);

printf("time default = %f**\n**", (**double**)tt / pow(**10**, **9**));

init\_matrix\_C(C, SIZE);

clock\_gettime(CLOCK\_REALTIME, &mt1);

dgemm\_transpose(A, B, C, SIZE);

clock\_gettime(CLOCK\_REALTIME, &mt2);

tt = pow(**10**, **9**) \* (mt2.tv\_sec - mt1.tv\_sec) + (mt2.tv\_nsec - mt1.tv\_nsec);

printf("time transpose= %f**\n**", (**double**)tt / pow(**10**, **9**));

init\_matrix\_C(C, SIZE);

clock\_gettime(CLOCK\_REALTIME, &mt1);

dgemm\_block(A, B, C, SIZE);

clock\_gettime(CLOCK\_REALTIME, &mt2);

tt = pow(**10**, **9**) \* (mt2.tv\_sec - mt1.tv\_sec) + (mt2.tv\_nsec - mt1.tv\_nsec);

printf("time block= %f**\n**", (**double**)tt / pow(**10**, **9**));

**return** **0**;

}

**void** **dgemm\_default**(**double** \*a, **double** \*b, **double** \*c, **int** SIZE)

{

**int** i, j, k;

**for** (i = **0**; i < SIZE; i++)

{

**for** (j = **0**; j < SIZE; j++)

{

**for** (k = **0**; k < SIZE; k++)

{

\*(c + i \* SIZE + j) += \*(a + i \* SIZE + k) \* \*(b + k \* SIZE + j);

}

}

}

}

**void** **dgemm\_transpose**(**double** \*a, **double** \*b, **double** \*c, **int** SIZE)

{

**int** i, j, k;

**for** (i = **0**; i < SIZE; i++)

{

**for** (k = **0**; k < SIZE; k++)

{

**for** (j = **0**; j < SIZE; j++)

{

\*(c + i \* SIZE + j) += \*(a + i \* SIZE + k) \* \*(b + k \* SIZE + j);

}

}

}

}

**void** **dgemm\_block**(**double** \*a, **double** \*b, **double** \*c, **int** SIZE)

{

**int** i0, j0, k0;

**int** c0[SIZE], a0[SIZE], b0[SIZE];

**for** (**int** i = **0**; i < SIZE; i += BS)

{

**for** (**int** j = **0**; j < SIZE; j += BS)

{

**for** (**int** k = **0**; k < SIZE; k += BS)

{

**for** (i0 = **0**, \*c0 = \*(c + i \* SIZE + j), \*a0 = \*(a + i \* SIZE + k); i0 < BS; ++i0, \*c0 += SIZE, \*a0 += SIZE)

{

**for** (k0 = **0**, \*b0 = \*(b + k \* SIZE + j); k0 < BS; ++k0, \*b0 += SIZE)

{

**for** (j0 = **0**; j0 < BS; ++j0)

{

c0[j0] += a0[k0] \* b0[j0];

}

}

}

}

}

}

}

**void** **init\_matrix**(**double** \*a, **double** \*b, **double** \*c, **int** SIZE)

{

**int** i, j, k;

**for** (i = **0**; i < SIZE; i++)

{

**for** (j = **0**; j < SIZE; j++)

{

**for** (k = **0**; k < SIZE; k++)

{

\*(a + i \* SIZE + j) = rand() % **100**;

\*(b + i \* SIZE + j) = rand() % **100**;

\*(c + i \* SIZE + j) = **0**;

}

}

}

}

**void** **init\_matrix\_C**(**double** \*c, **int** SIZE)

{

**int** i, j, k;

**for** (i = **0**; i < SIZE; i++)

{

**for** (j = **0**; j < SIZE; j++)

{

**for** (k = **0**; k < SIZE; k++)

{

\*(c + i \* SIZE + j) = **0**;

}

}

}

}

**void** **print\_matrix**(**double** \*a, **int** SIZE)

{

**int** i, j;

printf("Matrix:**\n**");

**for** (i = **0**; i < SIZE; i++)

{

**for** (j = **0**; j < SIZE; j++)

{

printf("%.2f ", \*(a + i \* SIZE + j));

}

printf("**\n**");

}

}