**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №2**

**по дисциплине «Параллельные алгоритмы и системы»**

**Тема: Методы оптимизации кода на примере перемножения матриц**

|  |  |
| --- | --- |
| Студент гр. 9307 | Немцов М.Л. |
| Преподаватель | Пазников А.А. |

Санкт-Петербург

2023

#### Цель работы

Изучить основные методы оптимизации кода.

**Задача**

Реализовать перемножение матриц и оптимизировать код.

**Подготовка**

Все замеры производились на следующей системе:

* Процессор: Intel Core i7-7700, 4 ядра, 8 потоков;
* ОЗУ: DDR4 16Gb 3200, 2 канальная;
* ОС: Windows 11

**Изначальный вариант программы без оптимизации**

Изначальная реализация перемножения матриц представлена ниже:

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

{

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

{

C[i\*N + j] = 0;

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

C[i\*N + j] += A[i\*K + k] \* B[k\*N + j];

}

}

При решении задачи «в лоб» фантастических результатов ожидать не стоило:

* Время выполнения: 2884,148 мсек;
* Производительность: 0,987 GFLOPS.

Как видно из результатов задействовано лишь малая часть вычислительных мощностей процессора. Посмотрим, что даст распараллеливание вычислений.

**Распараллеливание вычислений с помощью OpenMP**

Для распараллеливания вычислений воспользуемся библиотекой OpenMP, которая значительно упрощает организацию и логику работы потоков, что позволяет уменьшить время написания кода и упростить логику программы.

Посмотрим на результаты измерений (рис. 1):

Рис. 1 Результаты измерений

Из результатов измерений видно, что использование многопоточности позволило нам сократить время вычисления нашей задачи. Однако увеличение числа потоков не всегда даёт прирост в скорости выполнения программы, т.к. при увлечении потоков (> 8) время увеличивается, происходить большее число переключений контекста, а на переключение между контекстами затрачивается время. В нашем случае, оптимальное число потоков оказалось равным 8.

Перейдем теперь к оптимизации кода.

**Шаг 1: Устранение очевидных недостатков**

Устраним следующие очевидные недостатки:

* Вычисление адресов элементов массивов можно упростить — вынести постоянную часть из внутреннего цикла.
* В изначальной версии доступ к элементам массива B производится не последовательно. Его можно упорядочить, если поменять порядок вычисления таким образом, чтобы внутренним циклом был последовательный обход по строчкам для всех трех матриц;

В результате изменений получаем следующий код:

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

{

float \* c = C + i \* N;

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

c[j] = 0;

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

{

const float \* b = B + k \* N;

float a = A[i\*K + k];

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

c[j] += a \* b[j];

}

}

После первого шага оптимизации получаем следующие результаты:

* Время выполнения: 277,577 мсек;
* Производительность: 10,259 GFLOPS.

Неплохой результат, такими небольшими правками, получили ускорение в 10 раз! Посмотрим, что ещё можно оптимизировать.

**Шаг 2: Векторизация внутреннего цикла**

Если внимательно посмотреть на внутренний цикл (по переменной j), то видно, что вычисления можно проводить блоками (векторами). Практически все современные процессоры позволяют проводить вычисления над такими векторами. В частности, набор инструкций AVX оперирует с векторами размерностью 256 бит. Что позволяет выполнить 8 операций для вещественных чисел с одинарной точностью за такт. AVX2/FMA делает еще один шаг вперед — он позволяет выполнить слитную операцию умножения и сложения (d = a\*b + c) над вектором. Настольные процессоры Интел начиная с 4-го поколения имеют 2 256-bit FMA модуля, что позволяет им теоретически выполнять 2\*2\*8 = 32 операции (float-32) за такт. К счастью, инструкции AVX2/FMA достаточно легко задействовать напрямую из С/С++ при помощи встроенных функций (intrinsics).

После изменения код имеет следующий вид:

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

{

float \* c = C + i \* N;

for (int j = 0; j < N; j += 8)

\_mm256\_storeu\_ps(c + j + 0, \_mm256\_setzero\_ps());

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

{

const float \* b = B + k \* N;

\_\_m256 a = \_mm256\_set1\_ps(A[i\*K + k]);

for (int j = 0; j < N; j += 16)

{

\_mm256\_storeu\_ps(c + j + 0, \_mm256\_fmadd\_ps(a,

\_mm256\_loadu\_ps(b + j + 0), \_mm256\_loadu\_ps(c + j + 0)));

\_mm256\_storeu\_ps(c + j + 8, \_mm256\_fmadd\_ps(a,

\_mm256\_loadu\_ps(b + j + 8), \_mm256\_loadu\_ps(c + j + 8)));

}

}

}

После второго шага оптимизации получаем следующие результаты:

* Время выполнения: 185,138 мсек;
* Производительность: 15,381 GFLOPS.

Получили прирост около 40%, в принципе неплохо, но можно лучше. Но что же ограничивает?

Необходимо взять во внимание следующие факторы:

* Все современные компиляторы достаточные умные, и вполне справляются с задачей автовекторизации простых циклов. Уже в 1-м варианте компилятор фактически задействовал инструкции AVX2/FMA, потому ручная оптимизация дала не столь высокий результат;
* Скорость расчетов в данном случае упирается не в вычислителные возможности процессора, а в скорость загрузки и выгрузки данных. В данном случае процессору для задействования 2 256-bit FMA блоков требуется загрузить 4 и выгрузить 2 256-bit вектора за такт. Это в два раза превышает даже пропускную способность L1 кеша процессора (512/256 bit), не говоря уже о пропускной способности памяти, которая еще на порядок меньше (64-bit на канал).

Основная проблема состоит в ограниченной пропускной способности памяти в современных процессорах. Процессор фактически простаивает 90% времени, ожидая, когда данные загрузятся и сохранятся в памяти. Значит необходимо при дальнейшей оптимизации минимизировать доступ к памяти.

**Шаг 3: Создание микроядра**

В предыдущей версии на 1 FMA операцию приходится 2 загрузки и 1 выгрузка.

Больше всего загрузок и выгрузок происходит с результирующей матрицей С: данные из нее нужно загрузить, прибавить к ним произведение C[i][j] += A[i][k]\*B[k][j], а потом сохранить. И так много раз. Наиболее быстрая память, с которой может работать процессор — это его собственные регистры. Если мы будем хранить результирующее значение матрицы С в регистре процессора, то в процессе расчета нужно будет подгружать только значение матриц A и B. Теперь у нас на 1 FMA операцию приходится только 2 загрузки.

Если мы будем хранить в регистрах значения двух соседних столбцов матрицы C[i][j] и C[i][j+1], то сможем повторно использовать загруженное значение матрицы A[i][k]. И на 1 FMA операцию потребуется только 1.5 загрузки. Кроме того, сохраняя результат в 2 независимых регистра, мы позволим процессору выполнять 2 FMA операции за такт. Аналогично можно хранить в регистрах значения двух соседних строк — тогда будет осуществляться экономия на загрузке значений матрицы B.

Всего настольные процессоры Интел начиная с 2-го поколения имеют 16 256-bit векторных регистров (справедливо для 64-bit режима процессора). 12 из них можно использовать для хранения кусочка результирующей матрицы С размером 6x16.

Для вычисления маленьких кусочков матрицы С была написана функция micro\_6x16() (код функции представлен в приложении, файл step3.cpp). Также была написана вспомогательная функция для инициализации матрицы С init\_c (код функции представлен в приложении, файл step3.cpp).

Функция имеет следующий вид:

for (int i = 0; i < M; i += 6)

{

for (int j = 0; j < N; j += 16)

{

init\_c(6, 16, C + i\*N + j, N);

micro\_6x16(K, A + i\*K, K, 1, B + j, N, C + i\*N + j, N);

}

}

После 3го шага оптимизации получаем следующие результаты:

* Время выполнения: 88,777 мсек;
* Производительность: 32,282 GFLOPS.

Получили прирост почти в 2,5 раза, уже гораздо лучше, но можно еще оптимизировать.

**Шаг 4: Переупорядочивание матрицы B**

Микроядро за каждую итерацию загружает два 256-bit вектора из матрицы B. Причем каждый раз из новой строчки. Это делает невозможным для процессора эффективное кеширование этих данных. Для исправления введем следующие изменения:

* Скопируем данные матрицы B во временный буфер таким образом, чтобы данные, необходимые одному микроядру лежали рядом.
* Изменим порядок обхода матрицы С: сначала будем ходить по столбцам и только потом по строкам. Это позволит эффективнее использовать переупорядоченные значения матрицы B.

Для хранения буфера заведем небольшую структуру:

struct buf\_t

{

float \* p;

int n;

buf\_t(int size) : n(size), p((float\*)\_mm\_malloc(size \* 4, 64)) {}

~buf\_t() { \_mm\_free(p); }

};

Для переупорядочивания матрицы В использовалась следующая функция:

void reorder\_b\_16(int K, const float \* B, int ldb, float \* bufB)

{

for (int k = 0; k < K; ++k, B += ldb, bufB += 16)

{

\_mm256\_storeu\_ps(bufB + 0, \_mm256\_loadu\_ps(B + 0));

\_mm256\_storeu\_ps(bufB + 8, \_mm256\_loadu\_ps(B + 8));

}

}

Функция имеет следующий вид:

for (int j = 0; j < N; j += 16)

{

buf\_t bufB(16\*K);

reorder\_b\_16(K, B + j, N, bufB.p);

for (int i = 0; i < M; i += 6)

{

init\_c(6, 16, C + i\*N + j, N);

micro\_6x16(K, A + i\*K, K, 1, bufB.p, 16, C + i\*N + j, N);

}

}

После 4го шага оптимизации получаем следующие результаты:

* Время выполнения: 38,859 мсек;
* Производительность: 73,282 GFLOPS.

Получили прирост снова почти в 2,5 раза, значит мы на верном пути.

**Шаг 5: Локализация данных в кэше L1**

Чтобы буфер поместился в кэш L1, введем макроядро — вспомогательную функцию, которая производит расчеты над областью данных, которые влазят в кэш:

void macro\_v5(int M, int N, int K, const float \* A, int lda,

const float \* B, int ldb, float \* bufB, float \* C, int ldc)

{

for (int j = 0; j < N; j += 16)

{

reorder\_b\_16(K, B + j, ldb, bufB);

for (int i = 0; i < M; i += 6)

micro\_6x16(K, A + i\*lda, lda, 1, bufB, 16, C + i\*ldc + j, ldc);

}

}

Функция имеет следующий вид:

const int L1 = 32 \* 1024;

int mK = std::min(L1 / 4 / 16, K);

buf\_t bufB(16 \* mK);

for(int k = 0; k < K; k += mK)

{

int dK = std::min(K, k + mK) - k;

if(k == 0)

init\_c(M, N, C, N);

macro\_v5(M, N, dK, A + k, K, B + k\*N, N, bufB.p, C, N);

}

После 5го шага оптимизации получаем следующие результаты:

* Время выполнения: 30,762 мсек;
* Производительность: 82,384 GFLOPS.

Получаем прирост примерно на 15% - это значит мы приближаемся к финалу. В дальнейшем можно задействовать кэши L2 и L3, но это даст еще прирост в 10-15%.

**Выводы**

В ходе выполнения лабораторной работы мы познакомились с основными методами оптимизации кода, которые позволяют задействовать все возможности процессора и ускорить процесс больше чем в 80 раз. Далее представлены графики изменения времени и производительности (рис. 2 – 3). Из графиков видно, что время сократилось в 93 раза, а производительность выросла в 80 раз.

Рис. 2 График сокращения времени

Рис. 3 График роста производительности

**ПРИЛОЖЕНИЕ**

Optim\_matrix.cpp

#include "defs.h"

#ifdef \_MSC\_VER

#define NOMINMAX

#include <windows.h>

\_\_forceinline double time()

{

LARGE\_INTEGER counter, frequency;

QueryPerformanceCounter(&counter);

QueryPerformanceFrequency(&frequency);

return double(counter.QuadPart) / double(frequency.QuadPart);

}

#else

#include <sys/time.h>

inline \_\_attribute\_\_((always\_inline)) double time()

{

timeval t1;

gettimeofday(&t1, NULL);

return t1.tv\_sec + t1.tv\_usec \* 0.000001;

}

#endif

void init(buf\_t & buf)

{

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

buf.p[i] = float(rand()) / float(RAND\_MAX) - 0.5f;

}

inline float square(float x)

{

return x\*x;

}

inline float invalid(float a, float b, float e2)

{

float d2 = square(a - b);

return d2 > e2 && d2 > e2\*(a\*a + b\*b);

}

bool check(const buf\_t & control, const buf\_t & current, const std::string & desc, float eps = 0.001f)

{

assert(control.n == current.n);

float e2 = square(eps);

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

{

if (invalid(control.p[i], current.p[i], e2))

{

std::cout << desc << " : check error at " << i << ": ";

std::cout << std::setprecision(4) << std::fixed << control.p[i] << " != " << current.p[i] << std::endl;

return false;

}

}

return true;

}

int S = 1152;

int M = S, N = S, K = S, L = 0;

double TIME = 1.0;

bool test(gemm\_t gemm, const std::string & desc, const buf\_t & a, const buf\_t & b, const buf\_t & control)

{

buf\_t current(M\*N);

double t = 0;

int n = 0;

while(t < TIME)

{

double start = time();

gemm(M, N, K, a.p, b.p, current.p);

t += time() - start;

n++;

}

double gflops = 2\*double(M\*N)\*K\*n / t / (1024\* 1024\* 1024);

std::cout << desc << " : " << std::setprecision(3) << std::fixed << gflops << " GFLOPS; t = " << t/n\*1000.0f << " msec." << std::endl;

return check(control, current, desc);

}

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

{

if (argc > 1) M = atoi(argv[1]);

if (argc > 2) N = atoi(argv[2]);

if (argc > 3) K = atoi(argv[3]);

if (argc > 4) L = atoi(argv[4]);

buf\_t a(M\*K), b(K\*N), c(M\*N);

init(a);

init(b);

if (L <= 0 && !test(gemm\_v0, "gemm\_v0", a, b, c)) return 1;

if (L <= 1 && !test(gemm\_v1, "gemm\_v1", a, b, c)) return 1;

if (L <= 2 && !test(gemm\_v2, "gemm\_v2", a, b, c)) return 1;

if (L <= 3 && !test(gemm\_v3, "gemm\_v3", a, b, c)) return 1;

if (L <= 4 && !test(gemm\_v4, "gemm\_v4", a, b, c)) return 1;

if (L <= 5 && !test(gemm\_v5, "gemm\_v5", a, b, c)) return 1;

return 0;

}

defs.h

#pragma once

#include <immintrin.h>

#include <assert.h>

#include <stdlib.h>

#include <string>

#include <iostream>

#include <iomanip>

#include <algorithm>

typedef void (\*gemm\_t)(int M, int N, int K, const float\* A, const float\* B, float\* C);

void gemm\_v0(int M, int N, int K, const float\* A, const float\* B, float\* C);

void gemm\_v1(int M, int N, int K, const float\* A, const float\* B, float\* C);

void gemm\_v2(int M, int N, int K, const float\* A, const float\* B, float\* C);

void gemm\_v3(int M, int N, int K, const float\* A, const float\* B, float\* C);

void gemm\_v4(int M, int N, int K, const float\* A, const float\* B, float\* C);

void gemm\_v5(int M, int N, int K, const float\* A, const float\* B, float\* C);

void micro\_6x16(int K, const float\* A, int lda, int step, const float\* B, int ldb, float\* C, int ldc);

void init\_c(int M, int N, float\* C, int ldc);

void reorder\_b\_16(int K, const float\* B, int ldb, float\* \_B);

void reorder\_a\_6(const float\* A, int lda, int M, int K, float\* bufA);

struct buf\_t

{

float\* p;

int n;

buf\_t(int size) : n(size), p((float\*)\_mm\_malloc(size \* 4, 64)) {}

~buf\_t() { \_mm\_free(p); }

};

step0.cpp

#include "defs.h"

void gemm\_v0(int M, int N, int K, const float\* A, const float\* B, float\* C)

{

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

{

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

{

C[i \* N + j] = 0;

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

C[i \* N + j] += A[i \* K + k] \* B[k \* N + j];

}

}

}

step1.cpp

#include "defs.h"

void gemm\_v1(int M, int N, int K, const float\* A, const float\* B, float\* C)

{

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

{

float\* c = C + i \* N;

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

c[j] = 0;

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

{

const float\* b = B + k \* N;

float a = A[i \* K + k];

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

c[j] += a \* b[j];

}

}

}

step2.cpp

#include "defs.h"

void gemm\_v2(int M, int N, int K, const float\* A, const float\* B, float\* C)

{

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

{

float\* c = C + i \* N;

for (int j = 0; j < N; j += 8)

\_mm256\_storeu\_ps(c + j + 0, \_mm256\_setzero\_ps());

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

{

const float\* b = B + k \* N;

\_\_m256 a = \_mm256\_set1\_ps(A[i \* K + k]);

for (int j = 0; j < N; j += 16)

{

\_mm256\_storeu\_ps(c + j + 0, \_mm256\_fmadd\_ps(a,

\_mm256\_loadu\_ps(b + j + 0), \_mm256\_loadu\_ps(c + j + 0)));

\_mm256\_storeu\_ps(c + j + 8, \_mm256\_fmadd\_ps(a,

\_mm256\_loadu\_ps(b + j + 8), \_mm256\_loadu\_ps(c + j + 8)));

}

}

}

}

step3.cpp

#include "defs.h"

void micro\_6x16(int K, const float\* A, int lda, int step,

const float\* B, int ldb, float\* C, int ldc)

{

\_\_m256 c00 = \_mm256\_setzero\_ps();

\_\_m256 c10 = \_mm256\_setzero\_ps();

\_\_m256 c20 = \_mm256\_setzero\_ps();

\_\_m256 c30 = \_mm256\_setzero\_ps();

\_\_m256 c40 = \_mm256\_setzero\_ps();

\_\_m256 c50 = \_mm256\_setzero\_ps();

\_\_m256 c01 = \_mm256\_setzero\_ps();

\_\_m256 c11 = \_mm256\_setzero\_ps();

\_\_m256 c21 = \_mm256\_setzero\_ps();

\_\_m256 c31 = \_mm256\_setzero\_ps();

\_\_m256 c41 = \_mm256\_setzero\_ps();

\_\_m256 c51 = \_mm256\_setzero\_ps();

const int offset0 = lda \* 0;

const int offset1 = lda \* 1;

const int offset2 = lda \* 2;

const int offset3 = lda \* 3;

const int offset4 = lda \* 4;

const int offset5 = lda \* 5;

\_\_m256 b0, b1, a0, a1;

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

{

b0 = \_mm256\_loadu\_ps(B + 0);

b1 = \_mm256\_loadu\_ps(B + 8);

a0 = \_mm256\_set1\_ps(A[offset0]);

a1 = \_mm256\_set1\_ps(A[offset1]);

c00 = \_mm256\_fmadd\_ps(a0, b0, c00);

c01 = \_mm256\_fmadd\_ps(a0, b1, c01);

c10 = \_mm256\_fmadd\_ps(a1, b0, c10);

c11 = \_mm256\_fmadd\_ps(a1, b1, c11);

a0 = \_mm256\_set1\_ps(A[offset2]);

a1 = \_mm256\_set1\_ps(A[offset3]);

c20 = \_mm256\_fmadd\_ps(a0, b0, c20);

c21 = \_mm256\_fmadd\_ps(a0, b1, c21);

c30 = \_mm256\_fmadd\_ps(a1, b0, c30);

c31 = \_mm256\_fmadd\_ps(a1, b1, c31);

a0 = \_mm256\_set1\_ps(A[offset4]);

a1 = \_mm256\_set1\_ps(A[offset5]);

c40 = \_mm256\_fmadd\_ps(a0, b0, c40);

c41 = \_mm256\_fmadd\_ps(a0, b1, c41);

c50 = \_mm256\_fmadd\_ps(a1, b0, c50);

c51 = \_mm256\_fmadd\_ps(a1, b1, c51);

B += ldb; A += step;

}

\_mm256\_storeu\_ps(C + 0, \_mm256\_add\_ps(c00, \_mm256\_loadu\_ps(C + 0)));

\_mm256\_storeu\_ps(C + 8, \_mm256\_add\_ps(c01, \_mm256\_loadu\_ps(C + 8)));

C += ldc;

\_mm256\_storeu\_ps(C + 0, \_mm256\_add\_ps(c10, \_mm256\_loadu\_ps(C + 0)));

\_mm256\_storeu\_ps(C + 8, \_mm256\_add\_ps(c11, \_mm256\_loadu\_ps(C + 8)));

C += ldc;

\_mm256\_storeu\_ps(C + 0, \_mm256\_add\_ps(c20, \_mm256\_loadu\_ps(C + 0)));

\_mm256\_storeu\_ps(C + 8, \_mm256\_add\_ps(c21, \_mm256\_loadu\_ps(C + 8)));

C += ldc;

\_mm256\_storeu\_ps(C + 0, \_mm256\_add\_ps(c30, \_mm256\_loadu\_ps(C + 0)));

\_mm256\_storeu\_ps(C + 8, \_mm256\_add\_ps(c31, \_mm256\_loadu\_ps(C + 8)));

C += ldc;

\_mm256\_storeu\_ps(C + 0, \_mm256\_add\_ps(c40, \_mm256\_loadu\_ps(C + 0)));

\_mm256\_storeu\_ps(C + 8, \_mm256\_add\_ps(c41, \_mm256\_loadu\_ps(C + 8)));

C += ldc;

\_mm256\_storeu\_ps(C + 0, \_mm256\_add\_ps(c50, \_mm256\_loadu\_ps(C + 0)));

\_mm256\_storeu\_ps(C + 8, \_mm256\_add\_ps(c51, \_mm256\_loadu\_ps(C + 8)));

}

void init\_c(int M, int N, float\* C, int ldc)

{

for (int i = 0; i < M; ++i, C += ldc)

for (int j = 0; j < N; j += 8)

\_mm256\_storeu\_ps(C + j, \_mm256\_setzero\_ps());

}

void gemm\_v3(int M, int N, int K, const float\* A, const float\* B, float\* C)

{

for (int i = 0; i < M; i += 6)

{

for (int j = 0; j < N; j += 16)

{

init\_c(6, 16, C + i \* N + j, N);

micro\_6x16(K, A + i \* K, K, 1, B + j, N, C + i \* N + j, N);

}

}

}

step4.cpp

#include "defs.h"

void reorder\_b\_16(int K, const float\* B, int ldb, float\* bufB)

{

for (int k = 0; k < K; ++k, B += ldb, bufB += 16)

{

\_mm256\_storeu\_ps(bufB + 0, \_mm256\_loadu\_ps(B + 0));

\_mm256\_storeu\_ps(bufB + 8, \_mm256\_loadu\_ps(B + 8));

}

}

void gemm\_v4(int M, int N, int K, const float\* A, const float\* B, float\* C)

{

for (int j = 0; j < N; j += 16)

{

buf\_t bufB(16 \* K);

reorder\_b\_16(K, B + j, N, bufB.p);

for (int i = 0; i < M; i += 6)

{

init\_c(6, 16, C + i \* N + j, N);

micro\_6x16(K, A + i \* K, K, 1, bufB.p, 16, C + i \* N + j, N);

}

}

}

step5.cpp

#include "defs.h"

void macro\_v5(int M, int N, int K, const float \* A, int lda,

const float \* B, int ldb, float \* bufB, float \* C, int ldc)

{

for (int j = 0; j < N; j += 16)

{

reorder\_b\_16(K, B + j, ldb, bufB);

for (int i = 0; i < M; i += 6)

micro\_6x16(K, A + i\*lda, lda, 1, bufB, 16, C + i\*ldc + j, ldc);

}

}

void gemm\_v5(int M, int N, int K, const float \* A, const float \* B, float \* C)

{

const int L1 = 32 \* 1024;

int mK = std::min(L1 / 4 / 16, K);

buf\_t bufB(16 \* mK);

for(int k = 0; k < K; k += mK)

{

int dK = std::min(K, k + mK) - k;

if(k == 0)

init\_c(M, N, C, N);

macro\_v5(M, N, dK, A + k, K, B + k\*N, N, bufB.p, C, N);

}

}

matrix.cpp

#include <iostream>

#include <cstdlib>

#include <chrono>

#include <omp.h>

using namespace std;

void randomiseMatrix(int\*\* matrix, int n, int m) {

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

for (int j = 0; j < m; j++) {

matrix[i][j] = rand() % 11;

}

}

}

int main(int argc, char\*\* argv) {

srand(time(NULL));

int n1 = 1000;

int m1 = 500;

int n2 = 500;

int m2 = 1200;

//Матрица n1 x m1

int\*\* matrix1;

//Матрица n2 x m2

int\*\* matrix2;

matrix1 = (int\*\*)malloc(sizeof(int\*) \* n1);

for (int i = 0; i < n1; i++) {

matrix1[i] = (int\*)malloc(sizeof(int) \* m1);

}

matrix2 = (int\*\*)malloc(sizeof(int\*) \* n2);

for (int i = 0; i < n2; i++) {

matrix2[i] = (int\*)malloc(sizeof(int) \* m2);

}

//Генерируем случайные матрицы для умножения

randomiseMatrix(matrix1, n1, m1);

randomiseMatrix(matrix2, n2, m2);

int\*\* result = (int\*\*)malloc(sizeof(int\*) \* n1);;

for (int i = 0; i < n1; i++) {

result[i] = (int\*)malloc(sizeof(int) \* m2);

}

//Устанавливаем число потоков

int threadsNum = 16;

omp\_set\_num\_threads(threadsNum);

int i, j, k;

auto start = chrono::steady\_clock::now();

#pragma omp parallel for shared(matrix1, matrix2, result) private(i, j, k)

for (i = 0; i < n1; i++) {

for (j = 0; j < m2; j++) {

result[i][j] = 0;

for (k = 0; k < m1; k++) {

result[i][j] += (matrix1[i][k] \* matrix2[k][j]);

}

}

}

auto end = chrono::steady\_clock::now();

cout << "Elapsed time in milliseconds: "

<< chrono::duration\_cast<chrono::milliseconds>(end - start).count()

<< " ms" << endl;

return 0;

}