Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Архитектура процессоров и технология CUDA

ОТЧЕТ

по лабораторной работе №2

на тему

ВЕКТОРИЗАЦИЯ. ТЕХНОЛОГИЯ OPENMP

Студенты: Е.А. Петрович

М.А. Ходосевич

Преподаватель: Т.С. Жук

МИНСК 2024

**1 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ**

**1.1 Цель работы**

Изучить особенности векторизации и распараллеливания кода применительно к задаче перемножения двух матриц.

**1.2 Исходные данные к работе**

К лабораторной работе №1 «Изучение особенностей работы кэш-памяти» добавить следующие реализации:

Реализация 4: векторизация. Информацию по векторизации можно послушать на лекции, а также в презентации «презентация прошлых лет.pdf». То, как это делается для Intel, представлено во вспомогательном материале.

Векторизируется \*Реализация 3\* для кэша первого уровня (L1)!

Векторизация делается обязательно вручную (ассемблер либо интринсики)!

Реализация 5: технология OpenMP. Добавить к реализации 4 поддержку многопоточности с помощью технологии OpenMP.

На защите лабораторной работы №2 необходимо продемонстрировать список инструкций (расширений) доступных Вашему процессору (`lscpu` Вам в помощь).

Дополнительные условия:

– можно векторизировать абсолютно все реализации: оформить в виде таблицы в отчёте;

– можно добавить поддержку многопоточности для абсолютно всех реализаций: оформить в виде таблицы в отчёте.

**2 ВЫПОЛНЕНИЕ РАБОТЫ**

В ходе экспериментов изучалась производительность алгоритмов умножения матриц с использованием AVX-интринсиков и параллелизма через OpenMP. Основной целью было определить оптимальные размеры блоков для эффективного использования кэшей разных уровней процессора (L1, L2, L3). Проводились тесты с разными размерами блоков, чтобы минимизировать промахи кэша и ускорить расчет.

В ходе экспериментов изменялись следующие параметры:

– Размер блоков для разбиения матриц;

– Измерялось время выполнения умножения матрицы для каждого размера блоков;

– Количество потоков, между которыми распараллеливались вычисления.

В конечном итоге был выбран блок матрицы, который практически полностью занимал бы кэш, этот блок используется в дальнейшем. Количество потоков было выбрано равным количеству логических процессоров на ноутбуке – 8.

В таблицу 2.1 сведены время умножения матриц в обычном режиме, с векторизацией на всех уровнях кэша и с векторизацией с многопоточностью.

Таблица 2.1 – Результаты работы

|  |  |  |  |
| --- | --- | --- | --- |
|  | Обычный режим | Векторизация | OpenMP |
| L1 | 29.7 | 3.7 | 3.6 |
| L2 | 26.2 | 3.8 | 3.7 |
| L3 | 25.6 | 4.5 | 4.3 |

**3 ВЫВОДЫ**

В ходе лабораторной работы были изучены особенности векторизации и распараллеливания кода для задачи перемножения матриц. Были добавлены функции для векторизации и распараллеливания векторизации. Результаты были сведены в таблицу 2.1 Можно сделать вывод, что векторизация работает значительно быстрее обычного умножения, а распараллеливание потоков уменьшает время самой векторизации.

**ПРИЛОЖЕНИЕ А**

(обязательное)

Исходный текст программы

Содержимое файла laba2.cpp:

#include <iostream>

#include <chrono>

#include <intrin.h>

#include <immintrin.h> // Для AVX

#include <omp.h>

#include <string>

#include <cmath>

class Timer {

public:

void start()

{

m\_StartTime = \_\_rdtsc();

m\_bRunning = true;

}

void stop()

{

m\_EndTime = \_\_rdtsc();

m\_bRunning = false;

}

unsigned long long elapsedCycles() const {

if (m\_bRunning) {

return \_\_rdtsc() - m\_StartTime;

}

return m\_EndTime - m\_StartTime;

}

private:

unsigned long long m\_StartTime = 0;

unsigned long long m\_EndTime = 0;

bool m\_bRunning = false;

};

bool isMultiplying(int C1, int R2) {

return C1 == R2;

}

float\*\* fillMatrix(float\*\* matrix, int R, int C) {

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

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

matrix[i][j] = (float)(rand());

}

}

return matrix;

}

template <typename T>

T\*\* createMatrix(int R, int C) {

T\*\* matrix = new T \* [R];

bool isEmpty = false;

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

matrix[i] = new T[C];

if (!isEmpty) {

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

matrix[i][j] = static\_cast<T>(rand() / 10e3);

}

}

}

return matrix;

}

float\*\* variant1(float\*\* A, float\*\* B, int R1, int R2, int C1, int C2, bool writeConsole = false) {

auto timer = new Timer();

float\*\* result = createMatrix<float>(R1, C2);

timer->start();

for (int row = 0; row < R1; row++) {

for (int col = 0; col < C2; col++) {

for (int inner = 0; inner < C1; inner++) {

result[row][col] += A[row][inner] \* B[inner][col];

}

}

}

timer->stop();

if (writeConsole)

for (int row = 0; row < R1; ++row) {

for (int col = 0; col < C2; ++col) {

std::cout << result[row][col] << " ";

}

std::cout << "\n";

}

double elapsedTimeInSeconds = static\_cast<double>(timer->elapsedCycles()) / (1.19 \* 1e9);

std::cout << "\nMatrix multiplication time: " << timer->elapsedCycles() << " cycles.\n";

std::cout << "\nMatrix multiplication time: " << elapsedTimeInSeconds << " seconds.\n\n";

return result;

}

float\*\* variant2(float\*\* A, float\*\* B, int R1, int R2, int C1, int C2, int block\_size, bool writeConsole = false) {

auto timer2 = new Timer();

float\*\* result = createMatrix<float>(R1, C2);

timer2->start();

// Итерируемся по блокам

for (int block\_row = 0; block\_row < R1; block\_row += block\_size) {

for (int block\_col = 0; block\_col < C2; block\_col += block\_size) {

for (int inner\_block = 0; inner\_block < C1; inner\_block += block\_size) {

// Высчитываем размер текущего блока

int block\_row\_end = std::min(block\_row + block\_size, R1);

int block\_col\_end = std::min(block\_col + block\_size, C2);

int inner\_block\_end = std::min(inner\_block + block\_size, C1);

// Перемножаем блоки

for (int i = block\_row; i < block\_row\_end; i++) {

for (int j = block\_col; j < block\_col\_end; j++) {

for (int k = inner\_block; k < inner\_block\_end; k++) {

result[i][j] += A[i][k] \* B[k][j];

}

}

}

}

}

}

timer2->stop();

if (writeConsole)

for (int row = 0; row < R1; ++row) {

for (int col = 0; col < C2; ++col) {

std::cout << result[row][col] << " ";

}

std::cout << "\n";

}

double elapsedTimeInSeconds = static\_cast<double>(timer2->elapsedCycles()) / (1.19 \* 1e9);

std::cout << "\nMatrix multiplication time: " << timer2->elapsedCycles() << " cycles.\n";

std::cout << "\nMatrix multiplication time: " << elapsedTimeInSeconds << " seconds.\n\n";

return result;

}

double\*\* avx\_intrinsics(double\*\* A, double\*\* B, int R1, int R2, int C1, int C2, int block\_size, bool writeConsole = false) {

auto timer3 = new Timer();

timer3->start();

double\*\* result = createMatrix<double>(R1, C2);

timer3->start();

for (int block\_row = 0; block\_row < R1; block\_row += block\_size) {

for (int inner\_block = 0; inner\_block < C1; inner\_block += block\_size) {

for (int block\_col = 0; block\_col < C2; block\_col += block\_size) {

int block\_row\_end = std::min(block\_row + block\_size, R1);

int inner\_block\_end = std::min(inner\_block + block\_size, C1);

int block\_col\_end = std::min(block\_col + block\_size, C2);

for (int i = block\_row; i < block\_row\_end; i++) {

for (int k = inner\_block; k < inner\_block\_end; k++) {

\_\_m256d a\_vec = \_mm256\_set1\_pd(A[i][k]); // Загружаем элемент из A в каждый элемент регистра

// Цикл по столбцам текущего блока матрицы B

int j;

for (j = block\_col; j <= block\_col\_end - 4; j += 4) {

// Загружаем четыре элемента из матрицы B

\_\_m256d b\_vec = \_mm256\_loadu\_pd(&B[k][j]);

// Загружаем четыре элемента результата

\_\_m256d c\_vec = \_mm256\_loadu\_pd(&result[i][j]);

// Умножаем и суммируем четыре элемента

\_\_m256d res\_vec = \_mm256\_add\_pd(c\_vec, \_mm256\_mul\_pd(a\_vec, b\_vec));

// Записываем результат обратно в результат

\_mm256\_storeu\_pd(&result[i][j], res\_vec);

}

// Обрабатываем оставшиеся элементы

for (; j < block\_col\_end; ++j) {

result[i][j] += A[i][k] \* B[k][j];

}

}

}

}

}

}

timer3->stop();

// Вывод матрицы в консоль

if (writeConsole)

for (int row = 0; row < R1; ++row) {

for (int col = 0; col < C2; ++col) {

std::cout << result[row][col] << " ";

}

std::cout << "\n";

}

double elapsedTimeInSeconds = static\_cast<double>(timer3->elapsedCycles()) / (1.19 \* 1e9);

std::cout << "\nMatrix multiplication time: " << timer3->elapsedCycles() << " cycles.\n";

std::cout << "\nMatrix multiplication time: " << elapsedTimeInSeconds << " seconds.\n\n";

return result;

}

double\*\* avx\_intrinsics\_openMP(double\*\* A, double\*\* B, int R1, int R2, int C1, int C2, int block\_size, bool writeConsole = false) {

auto timer4 = new Timer();

double\*\* result = createMatrix<double>(R1, C2); // Создаем результирующую матрицу размером R1 x C2

timer4->start();

// Параллельный блок для обработки блоков строк

#pragma omp parallel num\_threads(12)

{

#pragma omp for // Приводит к тому, что работа в цикле for внутри параллельного региона будет разделена между потоками.

for (int block\_row = 0; block\_row < R1; block\_row += block\_size) {

for (int inner\_block = 0; inner\_block < C1; inner\_block += block\_size) {

for (int block\_col = 0; block\_col < C2; block\_col += block\_size) {

int block\_row\_end = std::min(block\_row + block\_size, R1);

int inner\_block\_end = std::min(inner\_block + block\_size, C1);

int block\_col\_end = std::min(block\_col + block\_size, C2);

for (int i = block\_row; i < block\_row\_end; i++) {

for (int k = inner\_block; k < inner\_block\_end; k++) {

\_\_m256d a\_vec = \_mm256\_set1\_pd(A[i][k]); // Загружаем элемент из A в каждый элемент регистра

// Цикл по столбцам текущего блока матрицы B

int j;

for (j = block\_col; j <= block\_col\_end - 4; j += 4) {

// Загружаем четыре элемента из матрицы B

\_\_m256d b\_vec = \_mm256\_loadu\_pd(&B[k][j]);

// Загружаем четыре элемента результата

\_\_m256d c\_vec = \_mm256\_loadu\_pd(&result[i][j]);

// Умножаем и суммируем четыре элемента

\_\_m256d res\_vec = \_mm256\_add\_pd(c\_vec, \_mm256\_mul\_pd(a\_vec, b\_vec));

// Записываем результат обратно в результат

\_mm256\_storeu\_pd(&result[i][j], res\_vec);

}

// Обрабатываем оставшиеся элементы

for (; j < block\_col\_end; ++j) {

result[i][j] += A[i][k] \* B[k][j];

}

}

}

}

}

}

}

// Вывод матрицы в консоль

if (writeConsole)

for (int row = 0; row < R1; ++row) {

for (int col = 0; col < C2; ++col) {

std::cout << result[row][col] << " ";

}

std::cout << "\n";

}

double elapsedTimeInSeconds = static\_cast<double>(timer4->elapsedCycles()) / (1.19 \* 1e9);

std::cout << "\nMatrix multiplication time: " << timer4->elapsedCycles() << " cycles.\n";

std::cout << "\nMatrix multiplication time: " << elapsedTimeInSeconds << " seconds.\n\n";

return result;

}

bool isMatrixCompare(float\*\* A, float\*\* B, int R, int C) {

for (int row = 0; row < R; row++) {

for (int col = 0; col < C; col++) {

float res = fabs(A[row][col] - B[row][col]);

if (res > 0.00001) {

std::cout << "\nMatrix error: " << A[row][col] << " " << B[row][col] << "\n\n\n";

return false;

}

}

}

return true;

}

double\*\* convertFloatMatrixToDouble(float\*\* floatMatrix, int R, int C) {

double\*\* doubleMatrix = new double\* [R];

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

doubleMatrix[i] = new double[C];

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

doubleMatrix[i][j] = static\_cast<float>(floatMatrix[i][j]);

}

}

return doubleMatrix;

}

float\*\* convertDoubleMatrixToFloat(double\*\* doubleMatrix, int R, int C) {

float\*\* floatMatrix = new float\* [R];

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

floatMatrix[i] = new float[C];

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

floatMatrix[i][j] = static\_cast<float>(doubleMatrix[i][j]);

}

}

return floatMatrix;

}

int main() {

srand(time(0));

//1254\*1254 максимальная матрица влезающая в 3кэш

const int R1 = 1180;

const int C1 = 1200;

const int R2 = 1200;

const int C2 = 1300;

int L1 = 128; // 156 ячеек - тоесть 156\*156\*4 байта - 95 КБ. L1 кеш - 320 КБ (320/3\*0,9=96)

int L2 = 384; // 396 ячеек - тоесть 396\*396\*4 байта - 612 КБ. L2 кеш - 2 МБ (2/3\*0,9=0,6)

int L3 = 640; // 686 ячеек - тоесть 686\*686\*4 байта - 1.79 МБ. L3 кеш - 6 МБ (6\*3\*0,9=1,8)

float\*\* A = createMatrix<float>(R1, C1);

A = fillMatrix(A, R1, C1);

float\*\* B = createMatrix<float>(R2, C2);

B = fillMatrix(B, R2, C2);

if (isMultiplying(C1, R2)) {

float\*\* one = variant1(A, B, R1, R2, C1, C2);

//float\*\* two = variant2(A, B, R1, R2, C1, C2, L1);

//variant2(A, B, R1, R2, C1, C2, L2);

//variant2(A, B, R1, R2, C1, C2, L3);

double\*\* Acopy = convertFloatMatrixToDouble(A, R1, C1);

double\*\* Bcopy = convertFloatMatrixToDouble(B, R2, C2);

double\*\* three = avx\_intrinsics(Acopy, Bcopy, R1, R2, C1, C2, L3);

double\*\* four = avx\_intrinsics\_openMP(Acopy, Bcopy, R1, R2, C1, C2, L3);

float\*\* fourCopy = convertDoubleMatrixToFloat(four, R1, C2);

bool equal = isMatrixCompare(one, fourCopy, R1, C2);

std::cout << "\nIs matrix equal: " << equal << std::endl;

}

else {

std::cout << "\nMatrix CAN'T be multiplying" << std::endl;

}

}