**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе № 1**

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

**Тема: Оптимизация производительности алгоритма**

**Вариант 1**

|  |  |
| --- | --- |
| Студент гр. 9306 | Андреева А.В. |
| Преподаватель | Пазников А.А. |

Санкт-Петербург

2023

#### Цель работы

Оптимизировать код алгоритма способами обращения к памяти и процессору.

#### Задание

Вариант 1 – умножение матриц.

#### Выполнение

## **1. Первоначальный алгоритм**

Для начала напишем простой алгоритм и проверим его на небольших матрицах, чтобы удостовериться, что всё работает корректно:

#include <iostream>

#include <omp.h>

using namespace std;

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

//две матрицы 2х2

int row1 = 2, row2 = 2, col1 = 2, col2 = 2;

int\*\* a, \*\* b, \*\* c;

//заполнение первой матрицы

a = new int\* [row1];

for (int i = 0; i < row1; i++)

{

a[i] = new int[col1];

for (int j = 0; j < col1; j++)

{

a[i][j] = rand() % 10;

}

}

//заполнение второй матрицы

b = new int\* [row2];

for (int i = 0; i < row2; i++)

{

b[i] = new int[col2];

for (int j = 0; j < col2; j++)

{

b[i][j] = rand() % 10;

}

}

clock\_t start = clock();

// Умножение матриц

c = new int\* [row1];

for (int i = 0; i < row1; i++)

{

c[i] = new int[col2];

for (int j = 0; j < col2; j++)

{

c[i][j] = 0;

for (int k = 0; k < col1; k++)

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

}

}

clock\_t end = clock();

cout << "time = " << end - start << endl;

// Вывод элементов первой матрицы

cout << endl << "mantix a:" << endl;

for (int i = 0; i < row1; i++)

{

for (int j = 0; j < col1; j++)

cout << a[i][j] << " ";

cout << endl;

}

// Вывод элементов второй матрицы

cout << endl << "mantix b:" << endl;

for (int i = 0; i < row2; i++)

{

for (int j = 0; j < col2; j++)

{

cout << b[i][j] << " ";

}

cout << endl;

}

// Вывод матрицы произведения

cout << endl << "result mantix:" << endl;

for (int i = 0; i < row1; i++)

{

for (int j = 0; j < col2; j++)

cout << c[i][j] << " ";

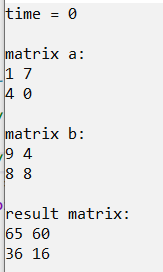
cout << endl;

}

return 0;

}

Вектор вычисляется правильно, однако из-за маленького размера данных все происходит слишком быстро:



Увеличиваем размерность данных:

#include <iostream>

#include <omp.h>

using namespace std;

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

//две матрицы 2000х2000

int row1 = 2000, row2 = 2000, col1 = 2000, col2 = 2000;

int\*\* a, \*\* b, \*\* c;

//заполнение первой матрицы

a = new int\* [row1];

for (int i = 0; i < row1; i++)

{

a[i] = new int[col1];

for (int j = 0; j < col1; j++)

{

a[i][j] = j % 10;

}

}

//заполнение второй матрицы

b = new int\* [row2];

for (int i = 0; i < row2; i++)

{

b[i] = new int[col2];

for (int j = 0; j < col2; j++)

{

b[i][j] = j % 10;

}

}

clock\_t start = clock();

// Умножение матриц

c = new int\* [row1];

for (int i = 0; i < row1; i++)

{

c[i] = new int[col2];

for (int j = 0; j < col2; j++)

{

c[i][j] = 0;

for (int k = 0; k < col1; k++)

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

}

}

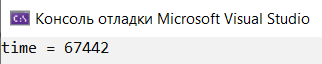
clock\_t end = clock();

cout << "time = " << end - start << endl;

return 0;

}

Время выполнения:



Будем считать это значение отправной точкой.

## **2. Построчное обращение ко второй матрице – пространственная локальность**

Пространственная локальность - когда мы обращаемся к данным, а потом обращаемся к другим данным, которые расположены в памяти рядом с первоначальными. За счет этого достигается быстродействие.

Чтобы добиться пространственной локальности, которая в нашем случае будет заключаться в том, что при обращении к элементам матриц мы будем на каждом следующем шаге обращаться к соседним элементам, нам следует обходить как первую, так и вторую матрицу построчно. Первая и так обходится построчно, а для второй придется изменить алгоритм: мы не будем вычислять каждый элемент окончательной матрицы за раз, а будем вычислять элементы частично на каждой итерации при проходах по матрицам построчно.

#include <iostream>

#include <omp.h>

using namespace std;

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

//две матрицы 2000х2000

int row1 = 2000, row2 = 2000, col1 = 2000, col2 = 2000;

int\*\* a, \*\* b, \*\* c;

//заполнение первой матрицы

a = new int\* [row1];

for (int i = 0; i < row1; i++)

{

a[i] = new int[col1];

for (int j = 0; j < col1; j++)

{

a[i][j] = j % 10;

}

}

//заполнение второй матрицы

b = new int\* [row2];

for (int i = 0; i < row2; i++)

{

b[i] = new int[col2];

for (int j = 0; j < col2; j++)

{

b[i][j] = j % 10;

}

}

clock\_t start = clock();

// Умножение матриц

c = new int\* [row1];

**for (int i = 0; i < row1; i++) {**

**c[i] = new int[col2] {};**

**for (int k = 0; k < col1; k++) {**

**int r1 = a[i][k];**

**// построчное обращение ко 2 матрице**

**for (int j = 0; j < col2; j++) {**

**c[i][j] += r1 \* b[k][j];**

**}**

**}**

**}**

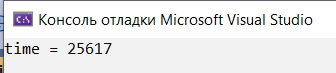
clock\_t end = clock();

cout << "time = " << end - start << endl;

return 0;

}

Время выполнения:



## **3. Конвейер**

Конвейерная реализация – стратегия, при которой за 1 итерацию цикла вычисляется не 1 элемент, а несколько. Так, в нашем случае, при умножении матрицы на вектор мы рассматриваем сразу 4 строки матрицы вместо 1. Также особенностью является то, что после работы «конвейерной» части алгоритма нам необходимо в обычном режиме завершить вычисления, относящиеся к строкам, которые не попали в конвейер. Это происходит из-за того, что число строк матрицы может не делиться нацело на количество, обрабатываемое за 1 проходку конвейера, тем самым порождая остаток из нескольких строк, которые все же нужно домножить.

#include <iostream>

#include <omp.h>

using namespace std;

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

//две матрицы 2000х2000

int row1 = 2000, row2 = 2000, col1 = 2000, col2 = 2000;

int\*\* a, \*\* b, \*\* c;

//заполнение первой матрицы

a = new int\* [row1];

for (int i = 0; i < row1; i++)

{

a[i] = new int[col1];

for (int j = 0; j < col1; j++)

{

a[i][j] = j % 10;

}

}

//заполнение второй матрицы

b = new int\* [row2];

for (int i = 0; i < row2; i++)

{

b[i] = new int[col2];

for (int j = 0; j < col2; j++)

{

b[i][j] = j % 10;

}

}

clock\_t start = clock();

// Умножение матриц

c = new int\* [row1];

**int maxCol2 = (col2 / 4) \* 4;**

**for (int i = 0; i < row1; i++) {**

**c[i] = new int[col2] {};**

**for (int k = 0; k < col1; k++) {**

**int r1 = a[i][k];**

**//построчное обращение ко 2 матрице + конвейер**

**for (int j = 0; j < maxCol2; j += 4) {**

**c[i][j] += r1 \* b[k][j];**

**c[i][j + 1] += r1 \* b[k][j + 1];**

**c[i][j + 2] += r1 \* b[k][j + 2];**

**c[i][j + 3] += r1 \* b[k][j + 3];**

**}**

**//доработка после конвейера**

**for (int j = maxCol2; j < col2; j++) {**

**c[i][j] += r1 \* b[k][j];**

**}**

**}**

**}**

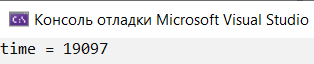
clock\_t end = clock();

cout << "time = " << end - start << endl;

return 0;

}

Время выполнения:



## **4. Многопоточность с помощью open MP**

#include <iostream>

#include <omp.h>

using namespace std;

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

//две матрицы 2000х2000

int row1 = 2000, row2 = 2000, col1 = 2000, col2 = 2000;

int\*\* a, \*\* b, \*\* c;

//заполнение первой матрицы

a = new int\* [row1];

for (int i = 0; i < row1; i++)

{

a[i] = new int[col1];

for (int j = 0; j < col1; j++)

{

a[i][j] = j % 10;

}

}

//заполнение второй матрицы

b = new int\* [row2];

for (int i = 0; i < row2; i++)

{

b[i] = new int[col2];

for (int j = 0; j < col2; j++)

{

b[i][j] = j % 10;

}

}

clock\_t start = clock();

// Умножение матриц

c = new int\* [row1];

int maxCol2 = col2 / 4;

//многопоточность

**#pragma omp parallel for**

for (int i = 0; i < row1; i++) {

c[i] = new int[col2] {};

for (int k = 0; k < col1; k++) {

int r1 = a[i][k];

//построчное обращение ко 2 матрице + конвейер

for (int j = 0; j < maxCol2; j += 4) {

c[i][j] += r1 \* b[k][j];

c[i][j + 1] += r1 \* b[k][j + 1];

c[i][j + 2] += r1 \* b[k][j + 2];

c[i][j + 3] += r1 \* b[k][j + 3];

}

//доработка после конвейера

for (int j = maxCol2; j < col2; j++) {

c[i][j] += r1 \* b[k][j];

}

}

}

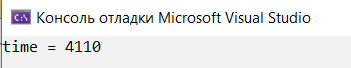
clock\_t end = clock();

cout << "time = " << end - start << endl;

return 0;

}

Время выполнения:



## **Тестирование**

Далее составим сводную таблицу с результатами тестирования каждой из версий. Метрики получены из VTune:

|  |  |
| --- | --- |
| Алгоритм | Метрики |
| 1. Оригинальный |  |
| 2. Пространственная локальность |  |
| 3. Конвейер |  |
| 4. Многопоточность |  |

Метрики, которые мы исследуем:

* CPU Time - время, в течение которого ЦП активно выполняет наше приложение.
* Memory Bound - Эта метрика показывает, в какой степени как проблемы с подсистемой памяти влияют на производительность. В частности L1 Bound, L2 Bound, L3 Bound показывают, насколько часто выполнение останавливалось, сталкиваясь с проблемой ненахождения данных в соответствующих кэшах.
* DRAM Bound - Эта метрика показывает, как часто ЦП зависал из-за основной памяти. Эта проблема решается кэшированием.
* Loads – количество загрузок/обращений к данным.
* Stores – количество сохранений данных.
* LLC Miss Count - показывает общее количество загрузок по требованию, которые не попали в LLC (last-level cache - кэш последнего уровня).
* Average Latency (cycles) - Эта метрика показывает среднюю задержку нагрузки в циклах.
* Total Thread Count – количество потоков.
* Clockticks – количество тактов, прошедших во время выполнения задачи. Такты - сигналы, которые генерируются процессором для координации всех операций.
* Instructions – количество инструкций, то есть команд, необходимых для выполнения.
* CPI Rate – количество тактов на инструкцию. Эта метрика зависит от, собственно, тактовой частоты процессора, которая определяет, насколько быстро выполняются инструкции, и от количества инструкций, которые нужно выполнить.

## **Выводы**

Из представленных скриншотов видно, что:

* Оригинальный алгоритм имел проблемы с памятью: показатели Memory Bound и DRAM Bound превышают установленные в программе в качестве приемлемых 20%.
* Поэтому решением этой проблемой послужило изменение алгоритма на более «дружелюбный к кэшу» путем добавления пространственной локальности в первом варианте оптимизации. Это дало улучшение по времени в 2.7 раза, а также уменьшение количества загрузок и сохранений, уменьшение промахов кэша, уменьшение IPC более чем в 2 раза.
* Вторая оптимизация с конвейером уменьшила время по сравнению с предыдущей в 1.3 раза, вместе с тем уменьшив почти все перечисленные выше метрики.
* При многопоточности улучшилось время по сравнению с предыдущей оптимизацией в 4,5 раза. Количество обращений к основной памяти немного увеличилось, т.к. повысился процент Memory Bound, также увеличилось значение CPI.