МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Направление подготовки: «Фундаментальная информатика и информационные технологии»
Магистерская программа: «Компьютерная графика»

Образовательный курс «Анализ производительности и оптимизации ПО»

ОТЧЕТ

по лабораторной работе

Оптимизация умножения матриц

Выполнила:
студентка группы 381806-2м
Михайлова Светлана

Нижний Новгород
2019

Содержание

Цели

Цель данной работы оптимизировать простейший кубический алгоритм перемножения двух матриц.

Профилирование тестовой программы

Для начала была написана обычная реализация перемножения двух матриц с использованием трех циклов, которая приведена ниже:

```
void MultiplyWithOutOptimization(int** aMatrix, unsigned int aRowNum, unsigned int aColumnNum,
                                 int** bMatrix, unsigned int bColumnNum)
{
    int** product = new int* [aRowNum];
    for (int i = 0; i < aRowNum; i++)
        product[i] = new int[bColumnNum];

    for (int i = 0; i < aRowNum; i++)
        for (int j = 0; j < bColumnNum; j++)
            product[i][j] = 0;

    for (int row = 0; row < aRowNum; row++) {
        for (int col = 0; col < bColumnNum; col++) {
            for (int inner = 0; inner < aColumnNum; inner++) {
                product[row][col] += aMatrix[row][inner] * bMatrix[inner][col];
            }
        }
    }
}
```

Рисунок 1. Простой кубический алгоритм перемножения двух матриц.

В качестве тестовых данных будем брать две квадратные случайно сгенерированные матрицы 1000 на 1000, каждый int элемент которой находится в диапазоне от 1 до 10.
В таком случае время работы простого кубического алгоритма: **12.4537** секунд. Будем считать этот результат отправной точкой для последующих оптимизаций.
Профилирование производилось на ПК со следующими характеристиками:

- Операционная система: Windows 10 Home
- CPU: Intel Core i5-6200U (2.3 GHz)
- GPU: NVidia 940M (2 GB)
- Оперативная память: 8 GB

Начальная реализация будет работать всегда медленно. Итерация по «inner», обращается к элементам матрицы b по столбцам. Такое чтение очень дорогое, потому что процессору приходится каждый раз подкачивать данные из памяти, вместо того чтобы брать их готовыми из кеша.

Проблема алгоритма в том, что он практически не использует этой возможности. При этом, случайный доступ к памяти (по столбцам) приводит к сбросу кеш линии и инициализации процедуры обновления кеш линии при каждом обращении.

Запустим Intel Vtune Amplifier Access Tool и проверим эту теорию:

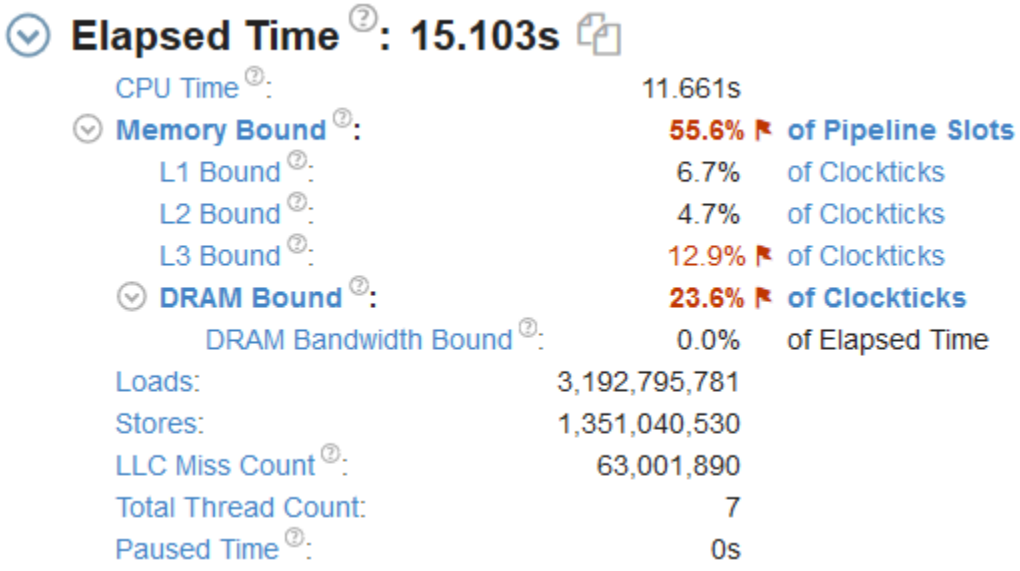

Рисунок 2. Статистика по работе с памятью изначального алгоритма

Попробуем исправить это и реализовать последовательный доступ к элементам матриц, чтобы получить максимальную выгоду от кеша. Для этого просто транспонируем матрицу b и будем обращаться к ее элементам по строкам.

```
void Multiply(int** aMatrix, unsigned int aRowNum, unsigned int aColumnNum,
              int** bMatrix, unsigned int bColumnNum)
{
    int* product = new int[aRowNum * bColumnNum];
    int* column = new int[aColumnNum];

    for (int j = 0; j < bColumnNum; j++)
        {
            for (int k = 0; k < aColumnNum; k++)
                column[k] = bMatrix[k][j];
            for (int i = 0; i < aRowNum; i++)
            {
                int* row = aMatrix[i];
                int summand = 0;
                for (int k = 0; k < aColumnNum; k++)
                    summand += row[k] * column[k];
                product[i * aRowNum + j] = summand;
            }
        }

}
```

Рисунок 3. Перемножение матриц с оптимизацией по кэшу (умножение транспонированной матрицы).

Время работы этой функции: 3.1832 секунды.

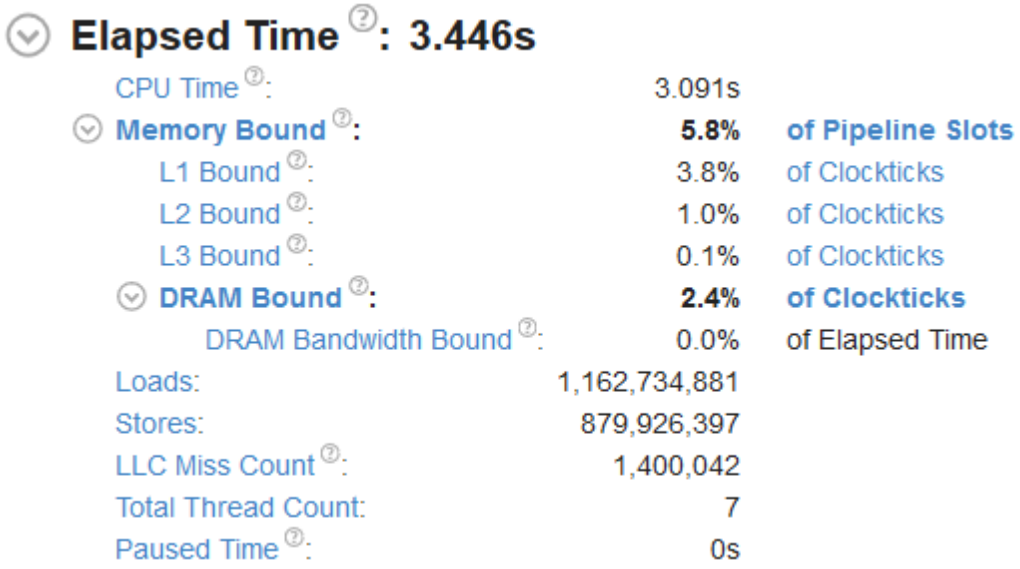

Рисунок 4. Статистика работы с памятью алгоритма с оптимизацией по кэшу.

Но это еще не все. Сейчас наша функция выполняется только в одном потоке вместо того, чтобы использовать все ресурсы CPU и выполняться в четырех. Попробуем исправить и это.

```
void Multiply(int** aMatrix, unsigned int aRowNum, unsigned int aColumnNum,
              int** bMatrix, unsigned int bColumnNum)
{
    int* product = new int[aRowNum * bColumnNum];
    int* column = new int[aColumnNum];
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        #pragma omp for
        for (int j = 0; j < bColumnNum; j++)
        {
            for (int k = 0; k < aColumnNum; k++)
                column[k] = bMatrix[k][j];
            for (int i = 0; i < aRowNum; i++)
            {
                int* row = aMatrix[i];
                int summand = 0;
                for (int k = 0; k < aColumnNum; k++)
                    summand += row[k] * column[k];
                product[i * aRowNum + j] = summand;
            }
        }
    }
}
```

Рисунок 5. Перемножение матриц с оптимизацией по кэшу (умножение транспонированной матрицы) и распараллеливанием.

Время работы после использования многопоточности: **1.33592** секунды, то есть скорость выполнения увеличилась примерно в двенадцать раз относительно первого измерения.

Заключение

В данной работе было показано, что можно значительно увеличить скорость выполнения задачи. Правильное чтение и запись, а также использование многопоточности позволило в 12 раз ускорить изначальный алгоритм.

Приложение

```
#include "pch.h"
#include <iostream>
#include <string>
#include <omp.h>

using namespace std;

int** GenerateMatrix(unsigned int rowNum, unsigned int columnNum)
{
   int** matrix = new int*[rowNum];
   for (int i = 0; i < rowNum; i++)
      matrix[i] = new int[columnNum];

   omp_set_num_threads(4);
   #pragma omp parallel for
   for (int i = 0; i < rowNum; i++) {
      for (int j = 0; j < columnNum; j++) {
         matrix[i][j] = rand() % 10 + 1;
      }
   }

   return matrix;
}

void MultiplyWithOutOptimization(int** aMatrix, unsigned int aRowNum, unsign
ed int aColumnNum, int** bMatrix, unsigned int bColumnNum)
{
   int** product = new int* [aRowNum];
   for (int i = 0; i < aRowNum; i++)
      product[i] = new int[bColumnNum];

   for (int i = 0; i < aRowNum; i++)
      for (int j = 0; j < bColumnNum; j++)
         product[i][j] = 0;

   for (int row = 0; row < aRowNum; row++) {
      for (int col = 0; col < bColumnNum; col++) {
         for (int inner = 0; inner < aColumnNum; inner++) {
            product[row][col] += aMatrix[row][inner] * bMatrix[inner][col];
         }
      }
   }
```

```
}

void Multiply(int** aMatrix, unsigned int aRowNum, unsigned int aColumnNum,
            int** bMatrix, unsigned int bColumnNum)
{
   int* product = new int[aRowNum * bColumnNum];
   int* column = new int[aColumnNum];
   omp_set_num_threads(4);
   #pragma omp parallel
   {
      #pragma omp for
      for (int j = 0; j < bColumnNum; j++)
      {
         for (int k = 0; k < aColumnNum; k++)
            column[k] = bMatrix[k][j];
         for (int i = 0; i < aRowNum; i++)
         {
            int* row = aMatrix[i];
            int summand = 0;
            for (int k = 0; k < aColumnNum; k++)
               summand += row[k] * column[k];
            product[i * aRowNum + j] = summand;
         }
      }
   }
}

int main()
{
   const int aRowNum = 1000;
   const int aColumnNum = 1000;
   const int bColumnNum = 1000;
   int** aMatrix = GenerateMatrix(aRowNum, aColumnNum);
   int** bMatrix = GenerateMatrix(aColumnNum, bColumnNum);
   double start, end;

   start = omp_get_wtime();
   MultiplyWithOutOptimization(aMatrix, aRowNum, aColumnNum, bMatrix, bC
olumnNum);
   end = omp_get_wtime();
   cout << "Work took " << end - start << "sec. time.\n";

   start = omp_get_wtime();
   Multiply(aMatrix, aRowNum, aColumnNum, bMatrix, bColumnNum);
```

```
    end = omp_get_wtime();
    cout << "Work with optimizations took " << end - start << "sec. time.\n";

    for (int i = 0; i < aRowNum; i++)
        delete[] aMatrix[i];
    delete aMatrix;

    for (int i = 0; i < aColumnNum; i++)
        delete[] bMatrix[i];
    delete bMatrix;

    getchar();
}
```