МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

**Федеральное государственное автономное   
образовательное учреждение высшего образования   
Национальный исследовательский Нижегородский государственный университет   
им. Н.И. Лобачевского**

|  |
| --- |
| **Институт информационных технологий, математики и механики** |

Направление подготовки

|  |
| --- |
| **Фундаментальная информатика и информационные технологии** |

(указывается код и наименование направления подготовки / специальности)

Направленность образовательной программы

|  |
| --- |
| **магистерская программа «Компьютерная графика и моделирование живых и технических систем»** |

**Образовательный курс: «Анализ производительности и оптимизации ПО»**

**Отчет по лабораторной работе**

**Промахи кэша**

(указывается профиль / магистерская программа / специализация)

**Выполнил(а):** студент(ка) группы 381706-3м

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.С.Зубарева

Подпись

Н. Новгород

2019

# Задача: L3 cache miss -> L3 cache hit

Характеристики компьютера:

Операционная система: Windows 7

Процессор: Intel Core i5 – 62004 2.30 GHz

L1 – 128 Кб, L2 – 512 Кб, L3- 3072 Кб

Оперативная память – 4 Гб

Описание проблемы

Кэш ЦП - это аппаратный кэш, используемый центральным процессором (ЦП) компьютера для снижения средней стоимости (времени или энергии) доступа к данным из основной памяти. Кэш-это меньшая, более быстрая память, ближе к ядру процессора, который хранит копии данных из часто используемых основных расположений памяти . По сути кэш-память выполняет роль быстродействующего буфера памяти хранящего информацию, которая может потребоваться процессору. Таким образом процессор получает необходимые данные в десятки раз быстрее, чем при считывании их из оперативной памяти.

Кэш память выполнена в виде микросхем статической оперативной памяти (SRAM), которые устанавливаются на системной плате либо встроены в процессор. В сравнении с другими видами памяти, статическая память способна работать на очень больших скоростях. Скорость кэша зависит от объема конкретной микросхемы. Самой распространенной на сегодняшний день считается трехуровневая система кэша L1,L2, L3:

L1 — самая маленькая по объему (всего несколько десятков килобайт), но самая быстрая по скорости и наиболее важная. Она содержит данные наиболее часто используемые процессором и работает без задержек. Обычно количество микросхем памяти уровня L1 равно количеству ядер процессора, при этом каждое ядро получает доступ только к своей микросхеме L1.

L2 по скорости уступает памяти L1, но выигрывает в объеме, который измеряется уже в нескольких сотнях килобайт. Она предназначена для временного хранения важной информации, вероятность обращения к которой ниже, чем у информации хранящейся в L1.

L3 — имеет самый большой объем из трех уровней (может достигать десятков мегабайт), но и обладает самой медленной скоростью. Кэш память L3 служит общей для всех ядер процессора. L3 предназначен для временного хранения тех важных данных, вероятность обращения к которым чуть ниже, чем у информации которая хранится в первых двух уровнях. Она также обеспечивает взаимодействие ядер процессора между собой.

Cache miss (кэш-промах) и Cache hit (кэш-попадание)

В процессе работы отдельные блоки информации копируются из основной памяти в кэш-память. При обращении процессора за командой или данными сначала проверяется их наличие в кэш-памяти. Если необходимая информация находится в кэше, она быстро извлекается. Это кэш-попадание. Если необходимая информация в кэш-памяти отсутствует (кэш-промах), то она выбирается из основной памяти, передается в микропроцессор и одновременно заносится в кэш-память. Повышение быстродействия вычислительной системы достигается в том случае, когда кэш-попадания реализуются намного чаще, чем кэш-промахи.

Описание решаемой задачи

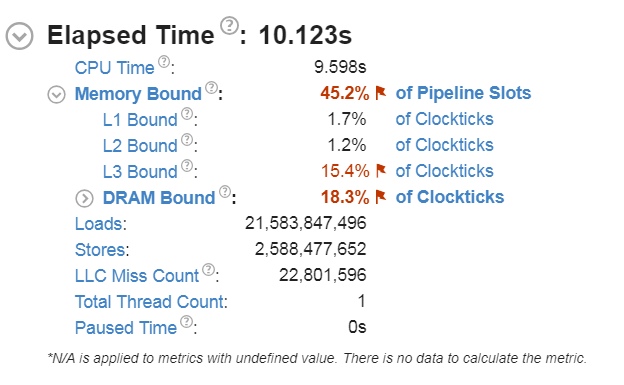
Проведем анализ простого классического алгоритма умножения матриц большой размерности (1000\*1000), заполненных рандомными положительными числами от 1до 100.

Обнаружение и решение проблемы

Для анализа исполььзуем Intel® VTune™ Amplifier XE 2019.

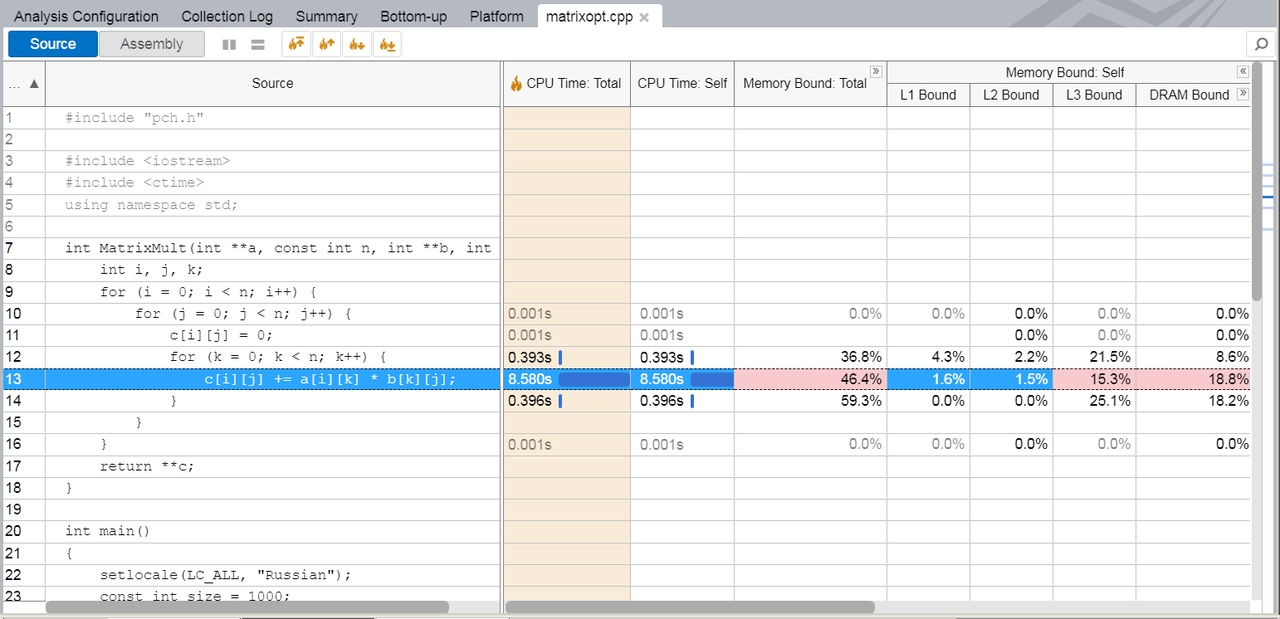
Intel® VTune™ Amplifier XE – профилировщик для оптимизации производительности и масштабируемости. Помогает за короткое время обнаружить проблемы производительности и причины плохой масштабируемости приложений на многоядерных системах.

Запустим Memory Access Tools:

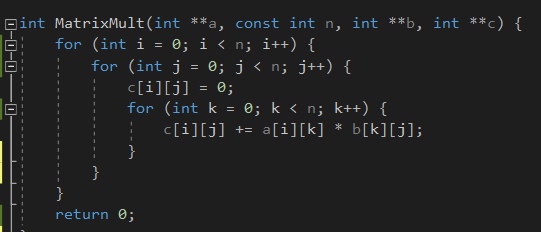


Время работы алгоритма: 10.123s.

Видим проблемы в значениях Memory Bound, а именно в кэш-памяти L3 и основной памятью (DRAM Bound). Посмотрим в каком месте кода обнаружена проблема:



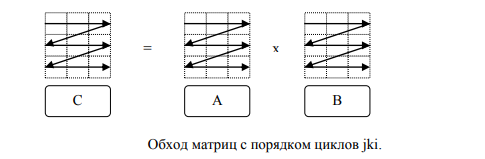
Видим, что это как цикл функции MatrixMult:



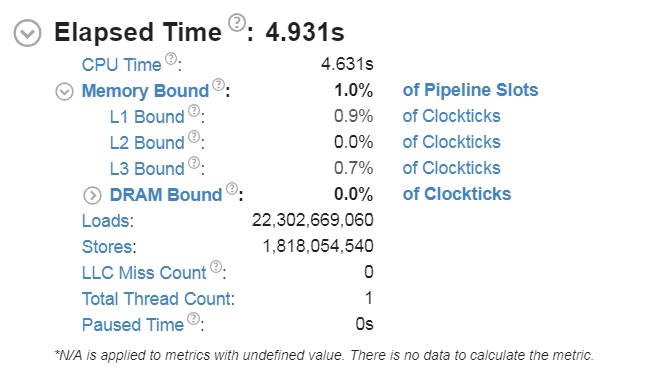
В реализации алгоритма умножения матриц порядок циклов (соответствующий определению):



Учитывая, что матрицы хранятся непрерывным вектором, обход матриц B и С не оптимален (хоть и имеет регулярный характер). Значительную роль в снижение производительности играет обход матрицы B. Элементы матрицы B используются n3 раз, а матрицы С – n2 раз. Что бы оптимизировать код, нужно сделать порядок обхода матриц последовательным. Транспонируем матрицу B:



Как следствие, повысилась производительность. Запустим Vtune Memory Access и проведем анализ:



Результаты

Был проведен анализ алгоритма умножения матриц в профилировщике в Intel® VTune™ Amplifier XE. Проведены анализ и оптимизация работы с памятью.

Современные процессоры чувствительны к тому, в каком порядке происходит чтение и запись в память. Если чтение и запись происходит последовательно, то процессор может это предсказать и заранее загрузить данные в кэш-память. Доступ к кэш-памяти гораздо быстрее доступа к оперативной памяти. Кроме того, данные в кэш-память загружаются не поэлементно, а сразу группами.

Низкая производительность алгоритма была из-за неудачно организованной работы с памятью. Проведена оптимизация, изменен порядок обхода матриц на последовательный. Таким образом, была увеличена производительность и время работы больше чем в 2 раза.

Приложение

#include <iostream>

#include <ctime>

using namespace std;

int MatrixMult(int \*\*a, const int n, int \*\*b, int \*\*c) {

int \*\*bT = new int \*[n];

for (int i = 0; i < n; i++) bT[i] = new int[n];

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

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

bT[i][j] = b[j][i];

}

}

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

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

c[i][j] = 0;

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

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

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

}//cout << "\n";

}

return 0;

}

int main()

{

setlocale(LC\_ALL, "Russian");

const int size = 1000;

int \*\*A=new int \*[size];

for (int i = 0; i < size; i++) A[i] = new int [size];

int \*\*B = new int \*[size];

for (int i = 0; i < size; i++) B[i] = new int[size];

int \*\*C = new int \*[size];

for (int i = 0; i < size; i++) C[i] = new int[size];

srand(time(NULL));

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

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

A[i][j] = rand() % 100 + 1;

//cout << A[i][j] << " ";

}//cout << "\n";

}

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

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

B[i][j] = rand() % 100 + 1;

//cout << B[i][j] << " ";

}//cout << "\n";

}

MatrixMult(A, size, B, C);

delete []A;

delete[]B;

delete[]C;

return 0;

}