**Лабораторная C0916**

Прудников Евгений

[**Параметры кэш памяти на ноутбуке** 2](#_Toc464044608)

[**Параметры ноутбука для экспериментов** 2](#_Toc464044609)

[**Анализатор кэш промахов** 3](#_Toc464044610)

[**Результаты работы анализатора. Сравнение с valgrind** 4](#_Toc464044611)

[**Умножение матриц n x n** 4](#_Toc464044612)

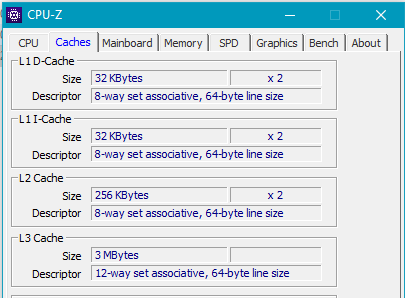
[**Транспонирование матрицы n x n** 4](#_Toc464044613)

[**Выводы и наблюдения** 5](#_Toc464044614)

[**Литература** 5](#_Toc464044615)

[**Приложение** 6](#_Toc464044616)

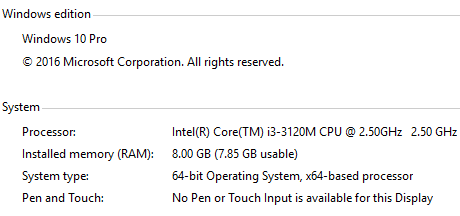
# **Параметры кэш памяти на ноутбуке**

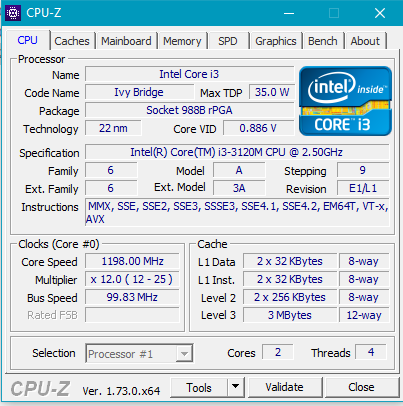
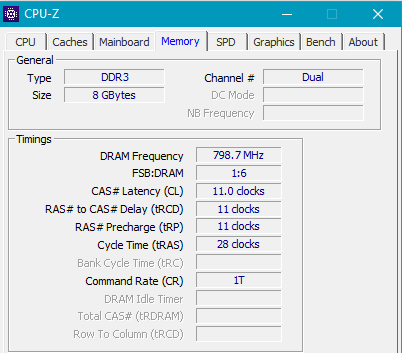


При предсказании кэш промахов использовался кэш L3 размером 3Mb и 12 каналами, так как анализатор valgrind использует его (значение LLd в выводе значит last level data cache) при моделировании.

# **Параметры ноутбука для экспериментов**

Операционная система:





# **Анализатор кэш промахов**

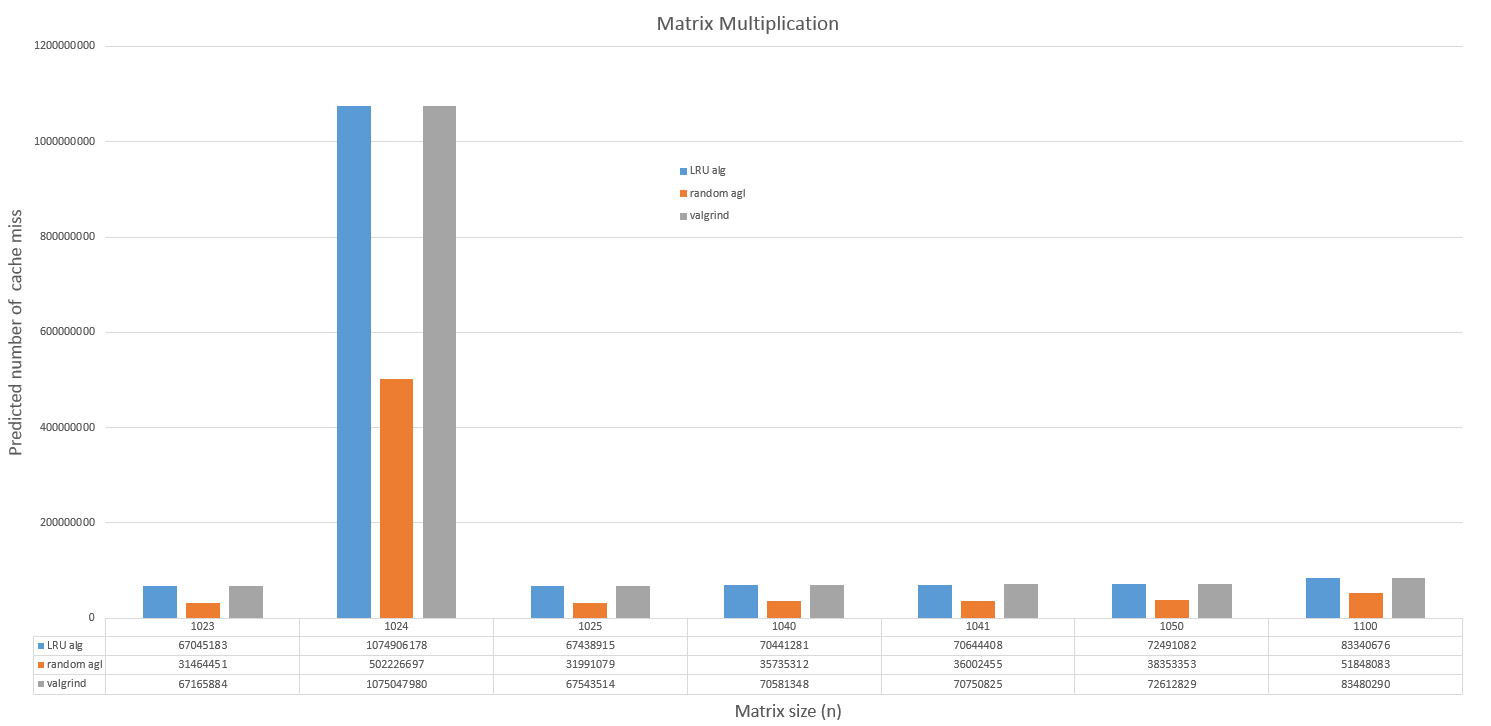
Исходный код в [**приложении**](#_Приложение).

Код программы main.cpp по умножению матриц был дополнен анализатором кэш промахов для алгоритма Simple. А также была добавлена функция транспонирования матрицы и дополнена анализатором. Анализатор может использовать 2 алгоритма: **LRU** (вытеснение самой давно использованной кэш-линии) и **random** (вытеснения случайной кэш-линии).

Аргументы программы: argv[1] – размер матриц (одно число n), argv[2] – алгоритм (0 – LRU, 1 - random), argv[3] – задача (0 – умножение матриц по алгоритму Simple, 1 – транспонирование матрицы a и запись результата в матрицу b)

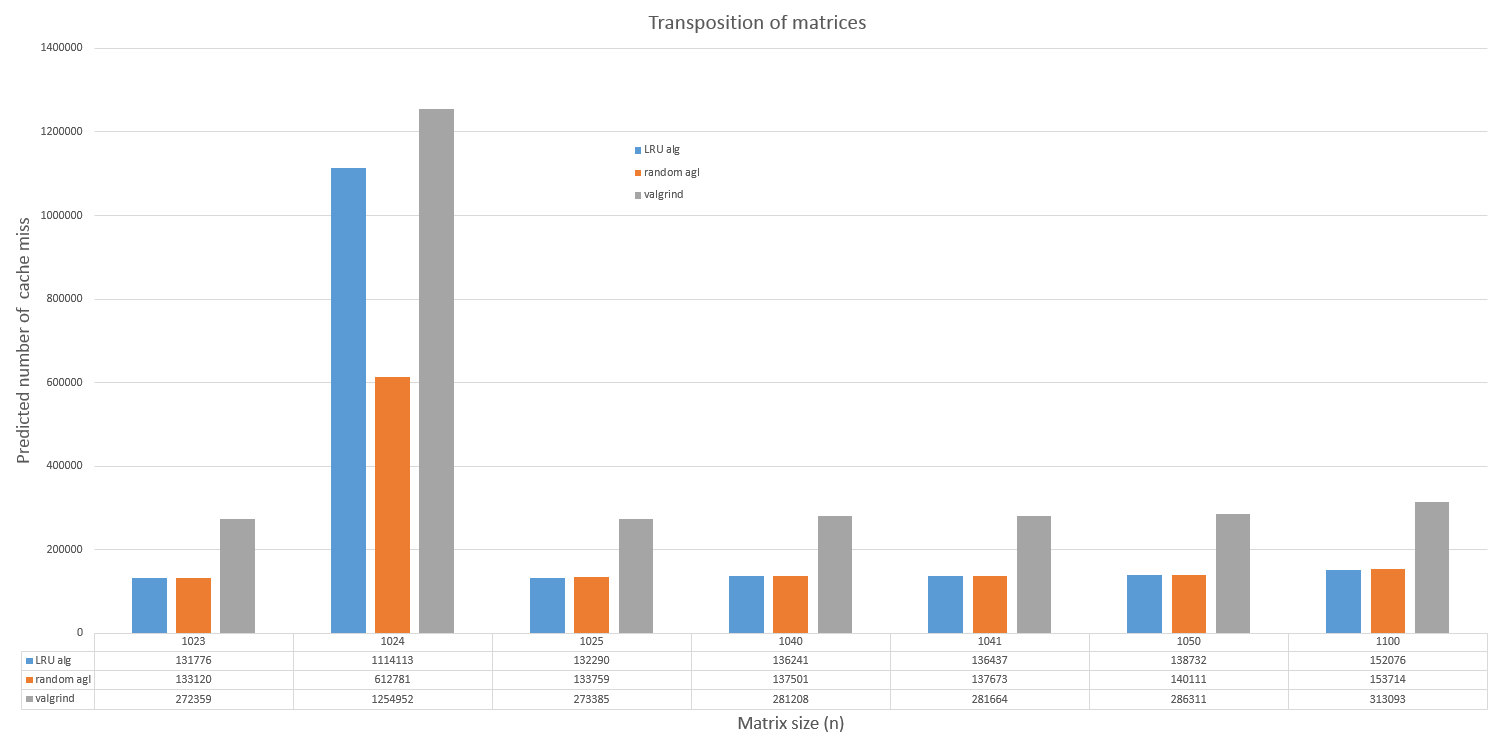
# **Результаты работы анализатора. Сравнение с valgrind**

## **Умножение матриц n x n**



На графике виден общий тренд: на 1024 все 3 анализатора предсказывают очень большое количество промахов, для остальных значений n это количество линейно увеличивается с увеличением n. Однако алгоритм вытеснения случайной кэш-линии (random) предсказывает ~2 раза меньшее число промахов чем остальные, предположительно это происходит из-за размера использованного кэша в 3Mb.

## **Транспонирование матрицы n x n**



При транспонировании матрицы общий тренд сохраняется (для 1024 большие значения, для остальных линейно растут), однако теперь LRU и random показывают ~одинаковые значения (кроме 1024) а анализатор valgrind в ~2 раза большие. Это связано с тем, что при транспонировании матрицы в самой функции не так много операций (меньше чем при умножении) а valgrind учитывает также операции заполнения матриц случайными значениями (функция FillRandom), а в предыдущем графике эти операции составляют малую долю и практически не влияют на результат, здесь же они заметны. Анализаторы LRU и Random в свою очередь анализируют только операции в функции транспонирования без учета FillRandom.

# **Выводы и наблюдения**

При умножении матриц количество кэш промахов, показываемых LRU анализатором, почти не отличается от значений анализатора valgrind, однако алгоритм random показывает ~ 2 раза меньшие значения. Возможно это связано с размером используемого кэша 3 Mb.

При транспонировании матриц напротив значения LRU и random практически совпадают, однако valgrind показывает ~2 раза большие значения. Из-за меньшего количества операций при транспонировании матриц, операции функции FillRandom становятся заметны при анализе valgrind, а анализаторы LRU и random их не учитывают, они анализируют только операции в самой функции Transpose

# **Литература**

* **Optimizing software in C++. An optimization guide for Windows, Linux and Mac platforms.** Agner Fog. Technical University of Denmark.
* **An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms.** Markus Kowarschik and Christian Weib
* **Что такое кэш процессора, и как он работает.**

Сергей Пахомов. КомпьютерПресс Январь 2013

# **Приложение**

Описание в пункте[**Анализатор кэш промахов**](#_Анализатор_кэш_промахов)

**Вывод программы:**

n = 1023

algorithm: LRU

task: Multiply

timeSimple: 743.414

# of checks: 3211797501 # of hits: 3144752318 # of misses: 67045183 Miss Rate: 0.0208747

**Исходный код:**

#include <iostream>

#include <vector>

#include <random>

#include <ctime>

using namespace std;

// cache parameters

const long cacheSize = 3145728, ways = 12, cacheLineSize = 64;

long numberOfSets = cacheSize / (ways \* cacheLineSize);

// initialize cache, lru

vector<vector<long>> cache(numberOfSets, vector<long>(ways, -10));

vector<vector<long>> LRU(numberOfSets, vector<long>(ways, -10));

long check = 0, hit = 0;

//Function to change the tag order in the LRU algorithm

void InitializeLRU(vector<vector<long>> & lru)

{

for (int i = 0; i < lru.size(); i++)

for (int j = 0; j < lru[i].size(); j++)

lru[i][j] = j;

}

void BringToTop(vector<vector<long>> & lru, long set, long ways, long x)

{

long i, pos;

for (i = 0; i < ways; i++)

if (lru[set][i] == x)

pos = i;

for (i = pos; i < ways - 1; i++)

lru[set][i] = lru[set][i + 1];

lru[set][ways - 1] = x;

}

template <typename T>

void AnalyzeAddress(T &variable, vector<vector<long>> &cache, vector<vector<long>> &LRU, int cacheLineSize, long &check, long &hit, int algorithm)

{

long address = (long)&variable;

long randomLine = 0;

long numberOfSets = cache.size();

long ways = cache[0].size();

long set, tag, found, pos;

set = (address / cacheLineSize) % numberOfSets;

tag = address / (cacheLineSize \* numberOfSets);

long i;

check++;

found = 0;

for (i = 0; i < ways; i++)

if (cache[set][i] == tag)

{

found = 1;

pos = i;

break;

}

if (found)

{

hit++;

BringToTop(LRU, set, ways, pos);

}

else

{

if (algorithm == 0)

{

i = LRU[set][0];

cache[set][i] = tag;

BringToTop(LRU, set, ways, i);

}

else

{

randomLine = rand() % ways;

cache[set][randomLine] = tag;

}

}

}

void MultSimple(const float\* \_\_restrict a, const float\* \_\_restrict b, float\* \_\_restrict c, int n, int algorithm)

{

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

{

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

{

c[i \* n + j] = 0.f;

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

{

// analyze variables address

AnalyzeAddress(a[i \* n + k], cache, LRU, cacheLineSize, check, hit, algorithm);

AnalyzeAddress(b[k \* n + j], cache, LRU, cacheLineSize, check, hit, algorithm);

AnalyzeAddress(c[i \* n + j], cache, LRU, cacheLineSize, check, hit, algorithm);

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

}

}

}

}

void Transpose(const float\* \_\_restrict a, float\* \_\_restrict b, int n, int algorithm)

{

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

{

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

{

AnalyzeAddress(a[j \* n + i], cache, LRU, cacheLineSize, check, hit, algorithm);

AnalyzeAddress(b[i \* n + j], cache, LRU, cacheLineSize, check, hit, algorithm);

b[i \* n + j] = a[j \* n + i];

}

}

}

void FillRandom(float\* a, int n)

{

std::default\_random\_engine eng;

std::uniform\_real\_distribution<float> dist;

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

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

a[i \* n + j] = dist(eng);

}

}

}

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

{

InitializeLRU(LRU);

const int n = atoi(argv[1]);

cout << "n = " << n << endl;

const int algorithm = atoi(argv[2]); // 0 - LRU , 1 - random

cout << "algorithm: " << (algorithm == 0 ? "LRU" : "random") << endl;

const int task = atoi(argv[3]); // 0 - Multiply , 1 - Transpose

cout << "task: " << (task == 0 ? "Multiply" : "Transpose") << endl;

float \*a = new float[n \* n];

float \*b = new float[n \* n];

float \*c = new float[n \* n];

FillRandom(a, n);

FillRandom(b, n);

if (task == 0)

{

const auto startTime = clock();

MultSimple(a, b, c, n, algorithm);

const auto endTime = clock();

cout << "timeSimple: " << double(endTime - startTime) / CLOCKS\_PER\_SEC << '\n';

}

else

{

const auto startTime = clock();

Transpose(a, b, n, algorithm);

const auto endTime = clock();

cout << "timeTranspose: " << double(endTime - startTime) / CLOCKS\_PER\_SEC << '\n';

}

cout << "# of checks: " << check;

cout << " # of hits: " << hit;

cout << " # of misses: " << check - hit;

cout << " Miss Rate: " << float(check - hit) / float(check);

//getchar(); // for windows

return 0;

}