МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ 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**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

Направление подготовки: «Прикладная математика и информатика»

Профиль подготовки: «Вычислительная математика и суперкомпьютерные технологии»

Отчет по лабораторной работе

**«Анализ производительности и оптимизация программ»**

**Выполнил:** студент группы 381903-3м

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Панов А.А.

Подпись

Нижний Новгород  
2019

# Введение

Рассматривается задача моделирования заряженных частиц в электромагнитном поле. Расчеты проводятся в двумерной области в виде прямоугольника. Область разбита на ячейки одинакового размера, в каждом узле ячейки определены электромагнитное поле и плотность тока. Внутри ячейки могут находиться частицы. Вычисления происходят по следующей схеме:

1. Вычисление электромагнитных полей.
2. Для каждой частицы:
   1. Вычисление силы Лоренца, действующей на частицу.
   2. Вычисление новой скорости и координаты частицы.
3. Вычисление токов, порождаемых движением частиц.

Этап 1

Интегрирование уравнений поля

Этап 2

Вычисление сил Лоренца

Интегрирование уравнений движения частиц

Этап 3

Вычисление токов

Рисунок 1 – Схема работы метода частиц в ячейках.

Численное моделирование происходит с заданным шагом по времени ∆𝑡, т.е. в дискретные моменты 0, ∆𝑡, 2∆𝑡, 3∆𝑡, … , последовательность завершается при превышении . При этом все данные о полях и частицах хранятся только на текущий момент времени и обновляются на соответствующих этапах метода.

Для решения задачи численного моделирования командой разработчиков (Мееров, Бастраков, Сурмин, ...) разработан программный комплекс PICADOR.

В данной работе рассмотрена частная задача с малым количеством частиц, в которой PICADOR показывает низкую производительностью.

# Сущность проблемы

Пусть size – количество ячеек в расчетной области, а N количество частиц. Алгоритмическая сложность первого этапа равна O(size). На втором (и на третьем) этапе некоторым образом обходятся все ячейки, внутри каждой ячейки обрабатываются все частицы. алгоритмическая сложность второго и третьего этапов равна O(size + N)

// Алгоритм 1. Базовый параллельный алгоритм этапа 2 (см. рис. Рисунок 1)

1. for walker in walkers:
2. #pragma omp parallel for
3. for cell in walker:
4. if cell not empty:
5. for particle in cell:
6. move(particle)

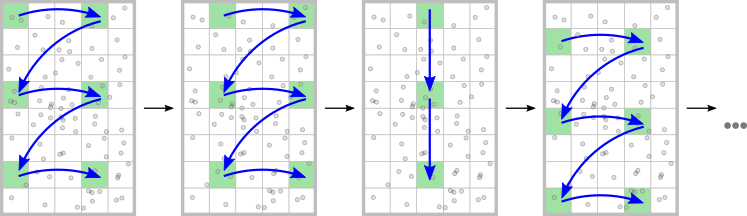


Рисунок 1 – Процесс обхода ячеек.

Существуют задачи, в которых нужно моделировать очень небольшое число частиц (большинство ячеек тогда пустые). Экспериментально обнаружено, что в данных задачах время работы второго и третьего этапа могут составлять до **80%** от общего времени работы. Это связано с многократным обходом ячеек, а также с довольно быстрым временем работы первого этапа. Также, даже уменьшение числа частиц до нуля, почти не сказалось на времени работы второго и третьего этапа. Это означает, что в данном случае нет смысла оптимизировать саму обработку частиц, нужно оптимизировать сам «обход».

# Анализ проблемы

## Конфигурация

* Компилятор:

Intel C++ Compiler 19.1

* Основные ключи компилятора:

/Qopenmp /O2 /arch:CORE-AVX2

* ОС:

Windows 10 Home

* Процессор:

Intel Core I5 9600K, частота зафиксирована на уровне 4.4 ГГц по всем ядрам

* Память:

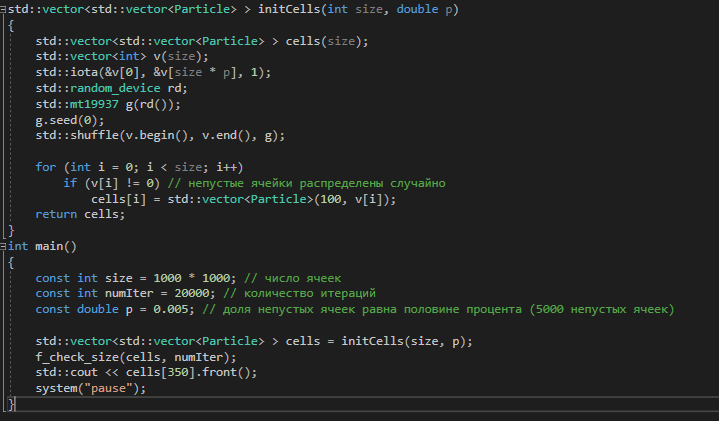
двухканальная ddr4, 2x8 Gb, 2800 МГц

Корректность работы программы проверялась с помощью вывода ответа на консоль.

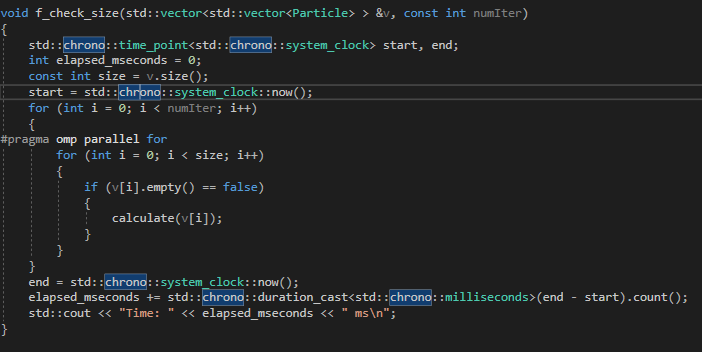
## Первая версия

Был написан бенчмарк в упрощенном виде моделирующий алгоритм 1 (этап 2).

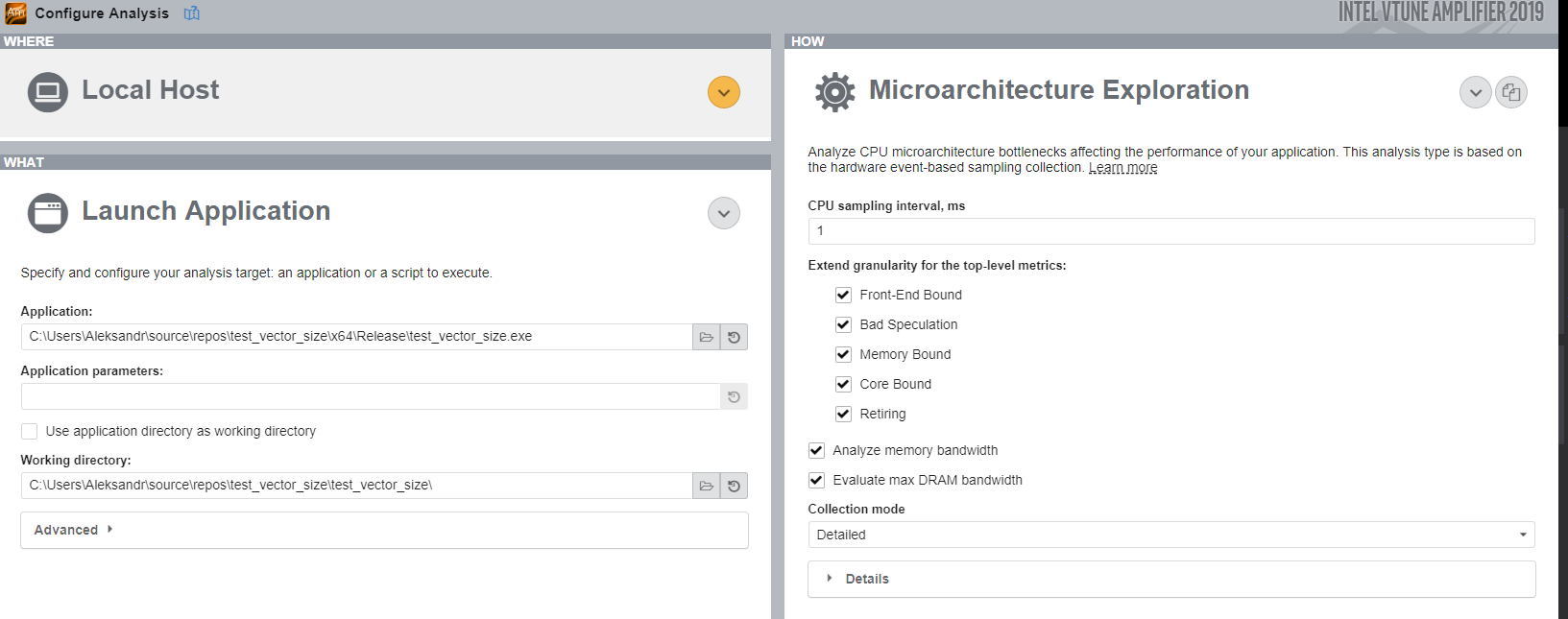
Создается 1000\*1000 ячеек (с 5000 случайно выбранными непустыми ячейками).



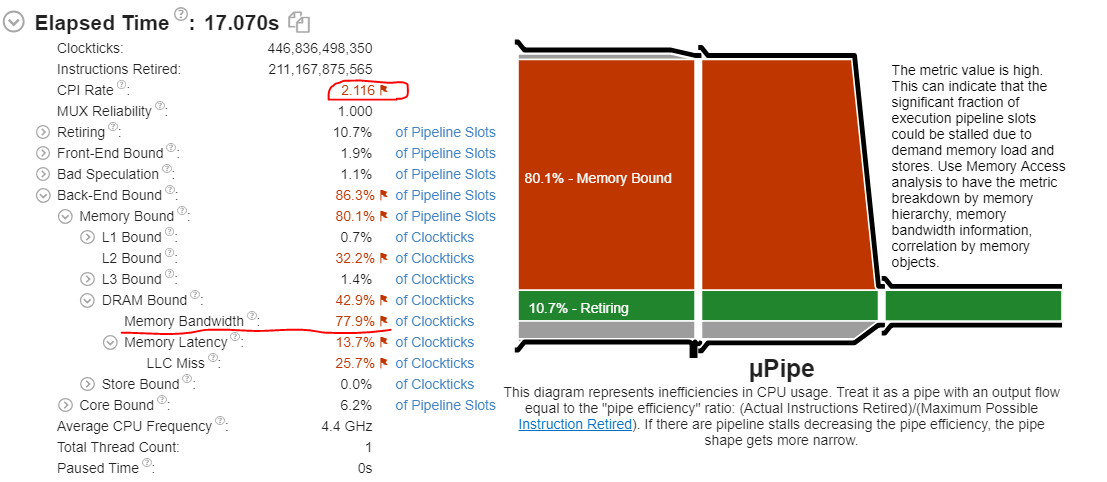
В f\_check\_size() выполняются вычисления с «частицами» в ячейке, при условии, что ячейка непустая.



Общее время работы программы составило около 17 секунд. Для оценки узких мест использовался анализ VTune Amplifier 2019.

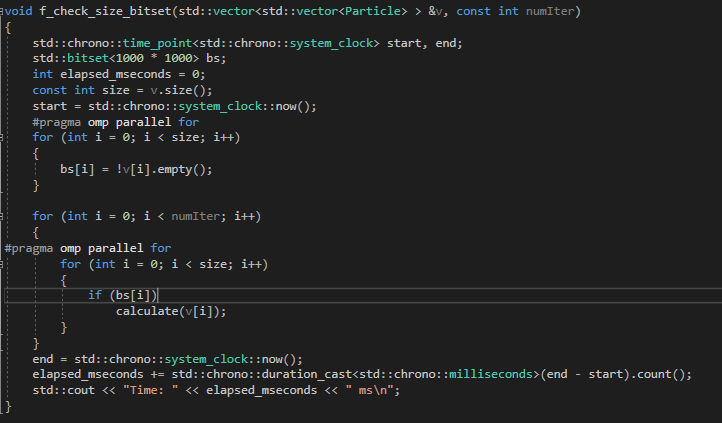


Анализ показал, что основная проблема это недостаточная пропускная способность памяти (Memory Bound). Также для выполнения одной инструкции требуется более двух тактов процессора (CPI Rate = 2.116).

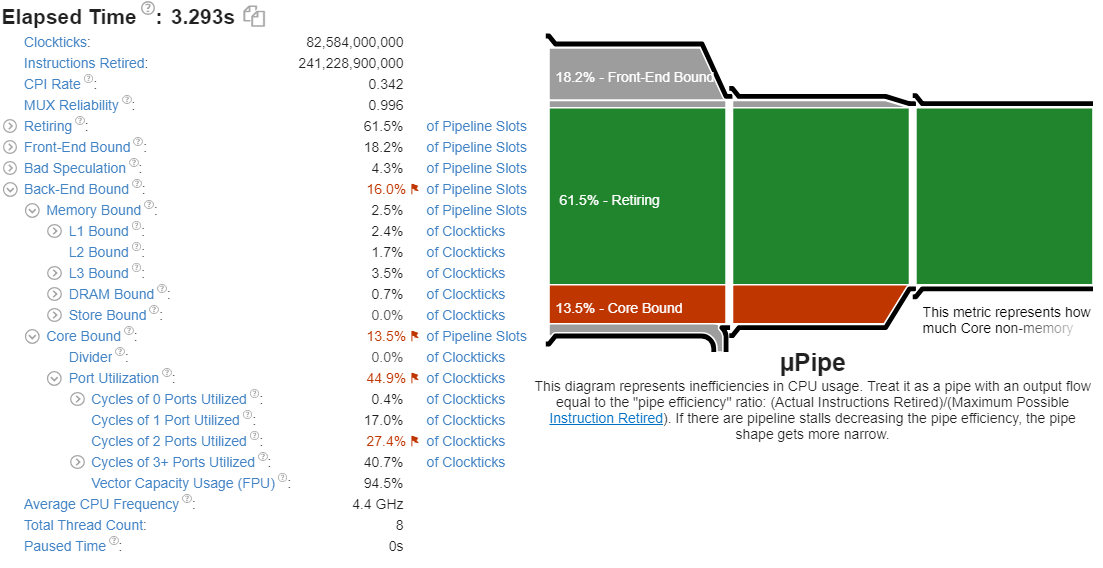


Для решения данной проблемы можно было бы хранить только индексы непустых ячеек и делать обход только по ним. Но в PICADOR обход ячеек выполняется специальным образом (см. рис. 1), а также частицы могут перелетать между ячейками, что потребовало бы обновление индексов. Поэтому было решено попробовать хранить информацию вида «есть ли частицы в ячейке» в bitset. Данная структура данных позволяет использовать гораздо меньше памяти.

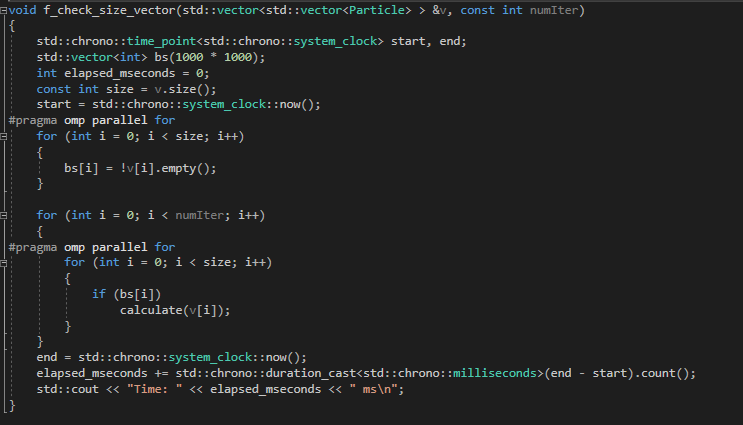
## Вторая версия программы



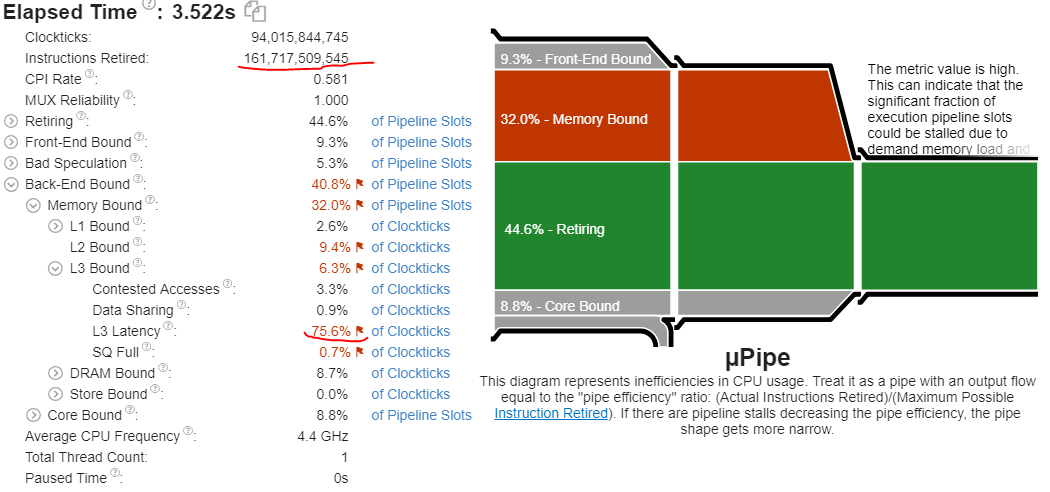
Использование bitset позволило сократить время работы с 17 секунд, до 3.3 и значительно уменьшило нагрузку на память.



CPI Rate также уменьшилось до 0.342, но при этом почти на четверть увеличилось общее количество выполненных инструкций! Появилось предположение, что компилятор сгенерировал слишком сложный код. Решено было заменить «сложный» bitset, на вектор содержащий размеры ячеек. Данная структура данных занимает больше памяти, но при этом гораздо проще устроена.

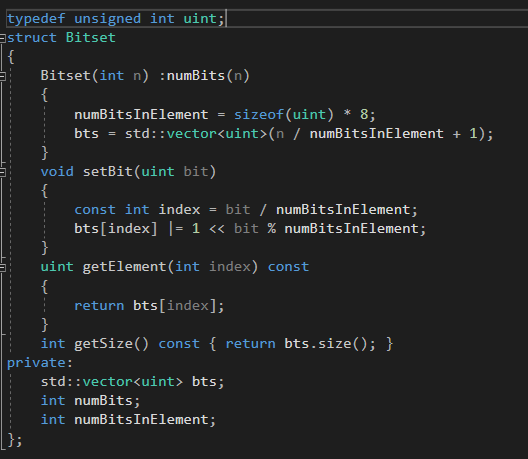


Время работы такого кода составило 3.4 секунды (почти столько же, сколько и с использованием bitset). Amplifier показал, что общее число выполненных инструкций с 240 млн уменьшилось до 160 млн! При этом видимо данные хорошо кэшировались и почти ушли проблемы с доступом в оперативную память.

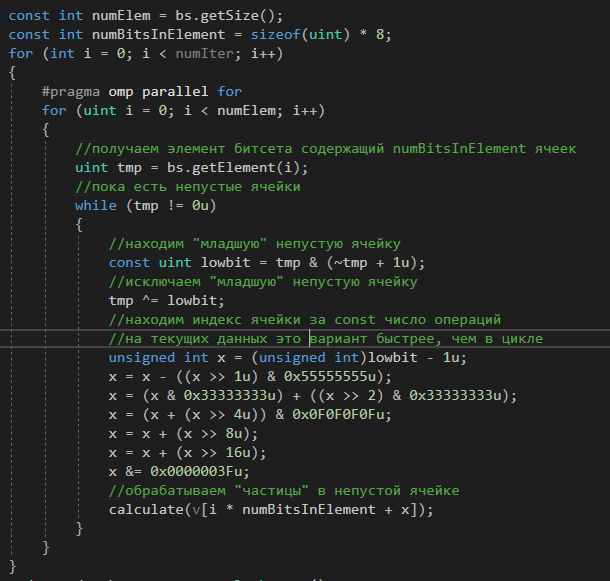


## Третья версия программы

Осталось совместить простое устройство структуры данных, с малым объемом используемой памяти. Для этого было решено написать свой простой bitset.



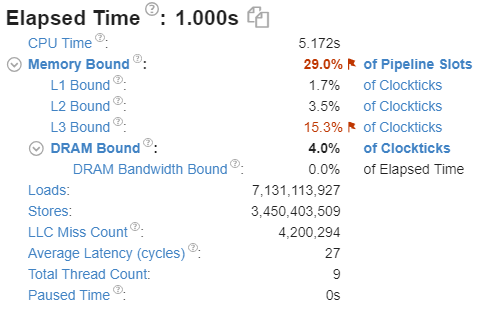
Главный вычислительный цикл следующий принял вид:

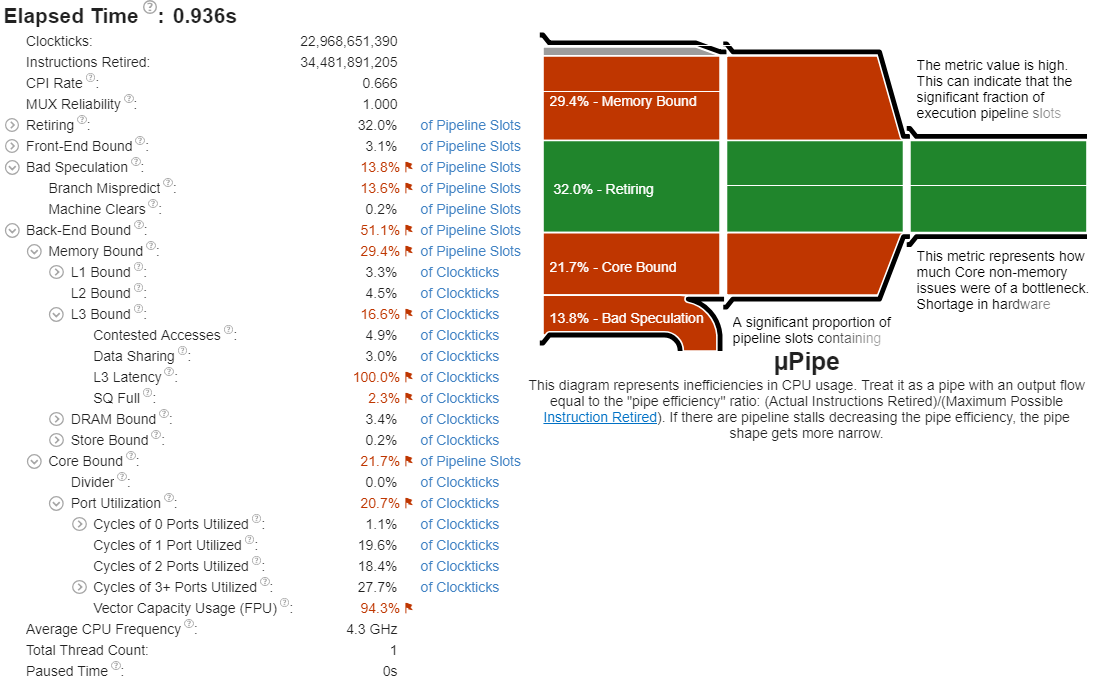


Время работы составило около 0.85 секунды (было 17 секунд, ускорение 20 кратное).

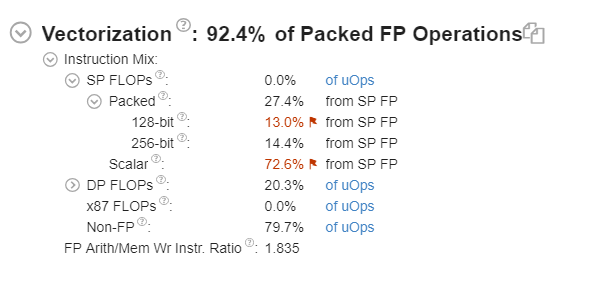
# Выводы

Анализ с помощью Amplifier показал, что общее число успешно выполненных инструкций уменьшилось со 160 млн. до 34 млн., проблемы с оперативной памятью почти полностью «перешли» на кэш L3.





В данном коде есть существенные проблемы с векторизацией. При этом векторизовать существующий код очень непросто, из-за if-ов.



Но можно это и не делать, ведь 2 и 3 этап (который моделирует данный бенчмарк) занимал всего 80% времени расчетов. Достигнутое 20 кратное ускорение, позволило уменьшить занимаемую долю времени расчетов до 17%, что является приемлемым результатом.

# Приложение

Код финальной версии:

#include <random>

#include <algorithm>

#include <iterator>

#include <iostream>

#include <numeric>

#include <chrono>

#include <bitset>

typedef double Particle;

inline void calculate(std::vector<Particle> &cell)

{

const int N = cell.size();

#pragma ivdep

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

{

cell[i]= cell[i] \* 0.9 + 1.1;

}

}

typedef unsigned int uint;

struct Bitset

{

Bitset(int n) :numBits(n)

{

numBitsInElement = sizeof(uint) \* 8;

bts = std::vector<uint>(n / numBitsInElement + 1);

}

void setBit(uint bit)

{

const int index = bit / numBitsInElement;

bts[index] |= 1 << bit % numBitsInElement;

}

uint getElement(int index) const

{

return bts[index];

}

int getSize() const { return bts.size(); }

private:

std::vector<uint> bts;

int numBits;

int numBitsInElement;

};

void f\_check\_size\_mybitset(std::vector<std::vector<Particle> > &v, const int numIter)

{

std::chrono::time\_point<std::chrono::system\_clock> start, end;

Bitset bs(1000 \* 1000);

int elapsed\_mseconds = 0;

const int size = v.size();

start = std::chrono::system\_clock::now();

#pragma omp parallel for

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

{

if (v[i].empty() == false)

bs.setBit(i);

}

const int numElem = bs.getSize();

const int numBitsInElement = sizeof(uint) \* 8;

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

{

#pragma omp parallel for

for (uint i = 0; i < numElem; i++)

{

//получаем элемент битсета содержащий numBitsInElement ячеек

uint tmp = bs.getElement(i);

//пока есть непустые ячейки

while (tmp != 0u)

{

//находим "младшую" непустую ячейку

const uint lowbit = tmp & (~tmp + 1u);

//исключаем "младшую" непустую ячейку

tmp ^= lowbit;

//находим индекс ячейки за const число операций

//на текущих данных это вариант быстрее, чем в цикле

unsigned int x = (unsigned int)lowbit - 1u;

x = x - ((x >> 1u) & 0x55555555u);

x = (x & 0x33333333u) + ((x >> 2) & 0x33333333u);

x = (x + (x >> 4u)) & 0x0F0F0F0Fu;

x = x + (x >> 8u);

x = x + (x >> 16u);

x &= 0x0000003Fu;

//обрабатываем "частицы" в непустой ячейке

calculate(v[i \* numBitsInElement + x]);

}

}

}

end = std::chrono::system\_clock::now();

elapsed\_mseconds += std::chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

std::cout << "Time: " << elapsed\_mseconds << " ms\n";

}

std::vector<std::vector<Particle> > initCells(int size, double p)

{

srand(0);

std::vector<std::vector<Particle> > cells(size);

std::vector<int> v(size);

std::iota(&v[0], &v[size \* p], 1);

std::random\_device rd;

std::mt19937 g(rd());

g.seed(0);

std::shuffle(v.begin(), v.end(), g);

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

if (v[i] != 0)

cells[i] = std::vector<Particle>(100, v[i]);

return cells;

}

int main()

{

const int size = 1000 \* 1000;

const int numIter = 20000;

const double p = 0.005;

std::vector<std::vector<Particle> > cells = initCells(size, p);

f\_check\_size\_mybitset(cells, numIter);

std::cout << cells[350].front();

}