

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:
 - 2.1. Вычисление силы Лоренца, действующей на частицу.
 - 2.2. Вычисление новой скорости и координаты частицы.
3. Вычисление токов, порождаемых движением частиц.

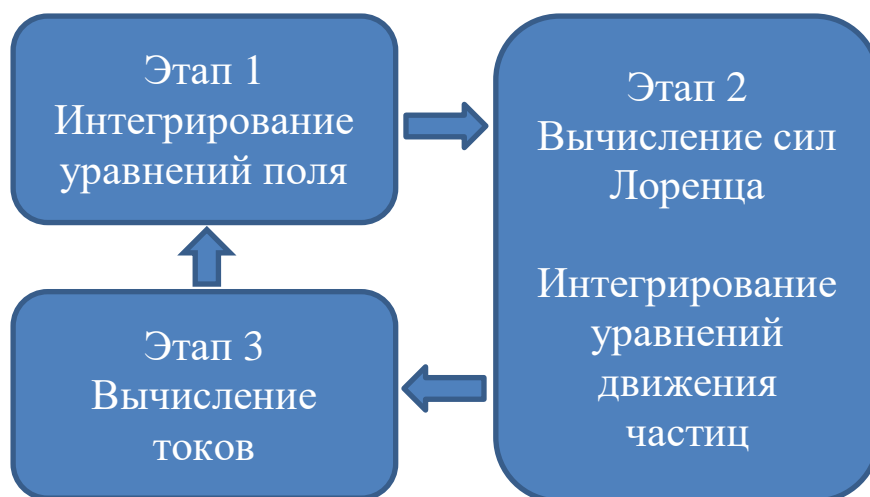


Рисунок 1 – Схема работы метода частиц в ячейках.

Численное моделирование происходит с заданным шагом по времени Δt , т.е. в дискретные моменты $0, \Delta t, 2\Delta t, 3\Delta t, \dots$, последовательность завершается при превышении t_{\max} . 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.

1. 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)
```

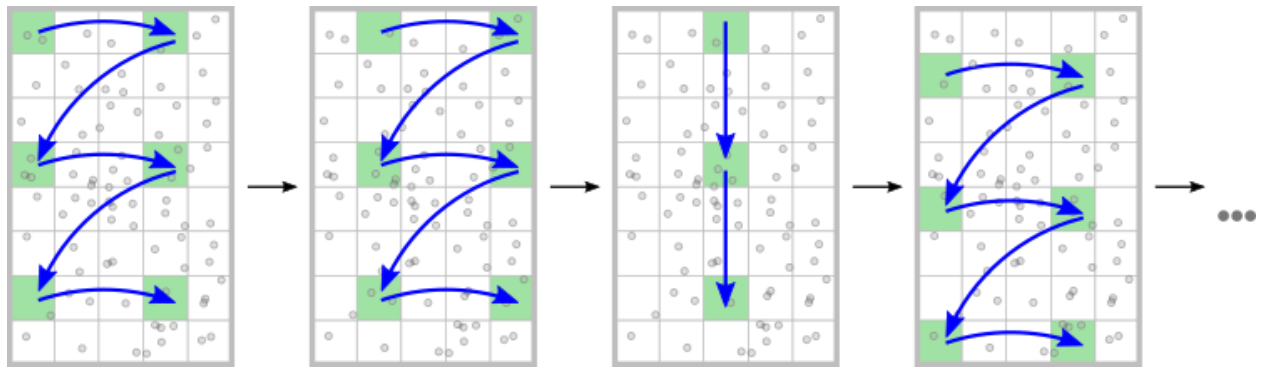


Рисунок 1 – Процесс обхода ячеек.

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».

2. Анализ проблемы

2.1 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.

2.2 Первая версия

Был написан бенчмарк в упрощенном виде моделирующий алгоритм 1 (этап 2).
Создается 1000*1000 ячеек (с 5000 случайно выбранными непустыми ячейками).

```
std::vector<std::vector<Particle> > initCells(int size, double p)
{
    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) // непустые ячейки распределены случайно
            cells[i] = std::vector<Particle>(100, v[i]);
    return cells;
}

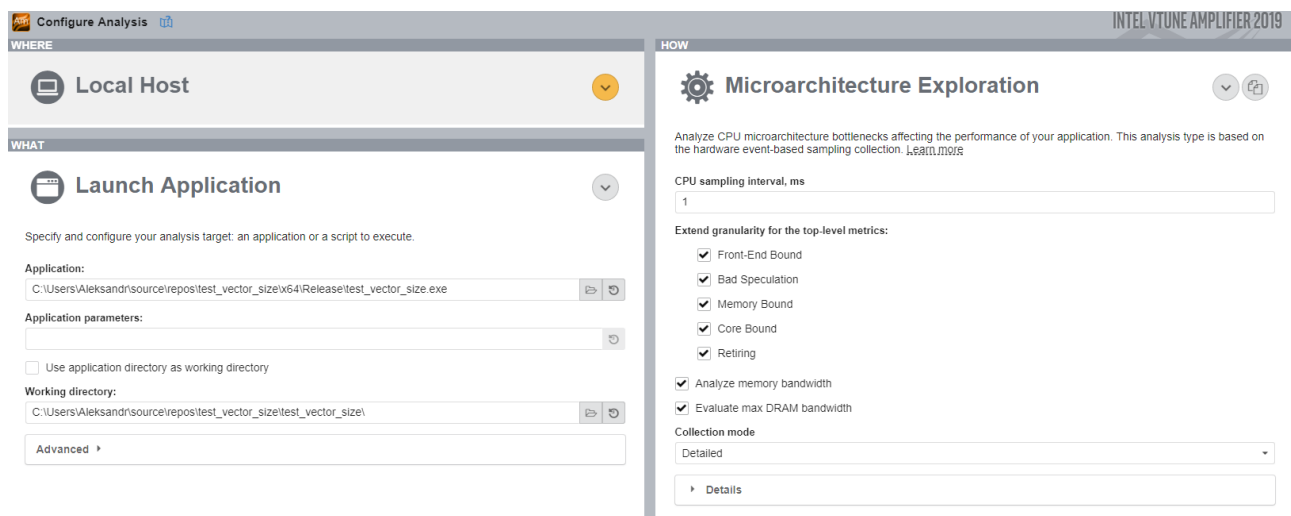
int main()
{
    const int size = 1000 * 1000; // число ячеек
    const int numIter = 20000; // количество итераций
    const double p = 0.005; // доля непустых ячеек равна половине процента (5000 непустых ячеек)

    std::vector<std::vector<Particle> > cells = initCells(size, p);
    f_check_size(cells, numIter);
    std::cout << cells[350].front();
    system("pause");
}
```

В `f_check_size()` выполняются вычисления с «частицами» в ячейке, при условии, что ячейка непустая.

```
void f_check_size(std::vector<std::vector<Particle> > &v, const int numIter)
{
    std::chrono::time_point<std::chrono::system_clock> start, end;
    int elapsed_mseconds = 0;
    const int size = v.size();
    start = std::chrono::system_clock::now();
    for (int i = 0; i < numIter; i++)
    {
#pragma omp parallel for
        for (int i = 0; i < size; i++)
        {
            if (v[i].empty() == false)
            {
                calculate(v[i]);
            }
        }
    }
    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";
}
```

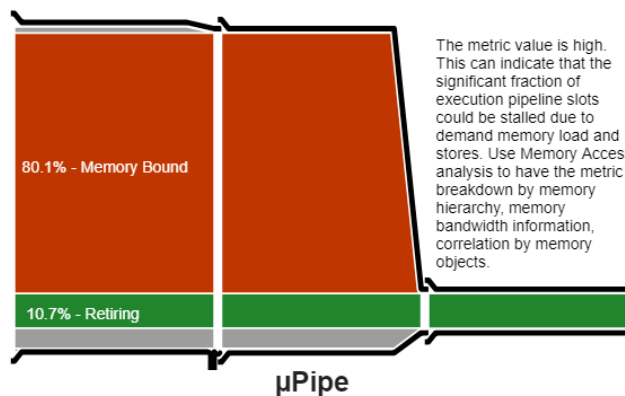
Общее время работы программы составило около 17 секунд. Для оценки узких мест использовался анализ VTune Amplifier 2019.



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).

Elapsed Time ^③: 17.070s

| | |
|--------------------------------------|-------------------------|
| Clockticks: | 446,836,498,350 |
| Instructions Retired: | 211,167,875,565 |
| CPI Rate ^② : | 2.116 |
| MUX Reliability ^② : | 1.000 |
| Retiring ^② : | 10.7% of Pipeline Slots |
| Front-End Bound ^② : | 1.9% of Pipeline Slots |
| Bad Speculation ^② : | 1.1% of Pipeline Slots |
| Back-End Bound ^② : | 86.3% of Pipeline Slots |
| Memory Bound ^② : | 80.1% of Pipeline Slots |
| L1 Bound ^② : | 0.7% of Clockticks |
| L2 Bound ^② : | 32.2% of Clockticks |
| L3 Bound ^② : | 1.4% of Clockticks |
| DRAM Bound ^② : | 42.9% of Clockticks |
| Memory Bandwidth ^② : | 77.9% of Clockticks |
| Memory Latency ^② : | 13.7% of Clockticks |
| LLC Miss ^② : | 25.7% of Clockticks |
| Store Bound ^② : | 0.0% of Clockticks |
| Core Bound ^② : | 6.2% of Pipeline Slots |
| Average CPU Frequency ^② : | 4.4 GHz |
| Total Thread Count: | 1 |
| Paused Time ^② : | 0s |



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

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.

2.3 Вторая версия программы

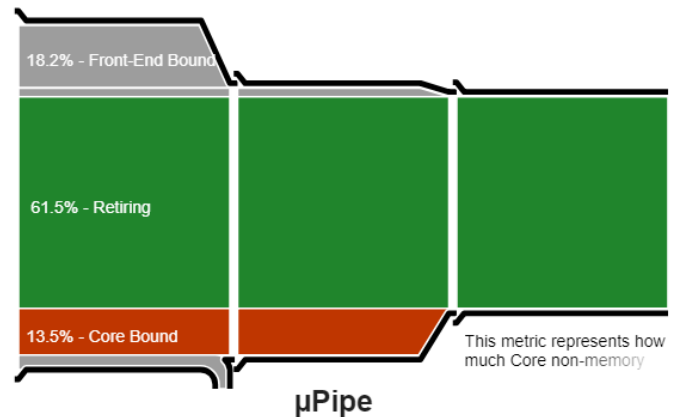
```
void f_check_size_bitset(std::vector<std::vector<Particle> > &v, const int numIter)
{
    std::chrono::time_point<std::chrono::system_clock> start, end;
    std::bitset<1000 * 1000> bs;
    int elapsed_mseconds = 0;
    const int size = v.size();
    start = std::chrono::system_clock::now();
    #pragma omp parallel for
    for (int i = 0; i < size; i++)
    {
        bs[i] = !v[i].empty();
    }

    for (int i = 0; i < numIter; i++)
    {
        #pragma omp parallel for
        for (int i = 0; i < size; i++)
        {
            if (bs[i])
                calculate(v[i]);
        }
    }
    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";
}
```

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

Elapsed Time[Ⓜ]: 3.293s

| | |
|--|-------------------------|
| Clockticks: | 82,584,000,000 |
| Instructions Retired: | 241,228,900,000 |
| CPI Rate [Ⓜ] : | 0.342 |
| MUX Reliability [Ⓜ] : | 0.996 |
| Retiring [Ⓜ] : | 61.5% of Pipeline Slots |
| Front-End Bound [Ⓜ] : | 18.2% of Pipeline Slots |
| Bad Speculation [Ⓜ] : | 4.3% of Pipeline Slots |
| Back-End Bound [Ⓜ] : | 16.0% of Pipeline Slots |
| Memory Bound [Ⓜ] : | 2.5% of Pipeline Slots |
| L1 Bound [Ⓜ] : | 2.4% of Clockticks |
| L2 Bound [Ⓜ] : | 1.7% of Clockticks |
| L3 Bound [Ⓜ] : | 3.5% of Clockticks |
| DRAM Bound [Ⓜ] : | 0.7% of Clockticks |
| Store Bound [Ⓜ] : | 0.0% of Clockticks |
| Core Bound [Ⓜ] : | 13.5% of Pipeline Slots |
| Divider [Ⓜ] : | 0.0% of Clockticks |
| Port Utilization [Ⓜ] : | 44.9% of Clockticks |
| Cycles of 0 Ports Utilized [Ⓜ] : | 0.4% of Clockticks |
| Cycles of 1 Port Utilized [Ⓜ] : | 17.0% of Clockticks |
| Cycles of 2 Ports Utilized [Ⓜ] : | 27.4% of Clockticks |
| Cycles of 3+ Ports Utilized [Ⓜ] : | 40.7% of Clockticks |
| Vector Capacity Usage (FPU) [Ⓜ] : | 94.5% |
| Average CPU Frequency [Ⓜ] : | 4.4 GHz |
| Total Thread Count: | 8 |
| Paused Time [Ⓜ] : | 0s |



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

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.

```

void f_check_size_vector(std::vector<std::vector<Particle> > &v, const int numIter)
{
    std::chrono::time_point<std::chrono::system_clock> start, end;
    std::vector<int> 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++)
    {
        bs[i] = !v[i].empty();
    }

    for (int i = 0; i < numIter; i++)
    {
#pragma omp parallel for
        for (int i = 0; i < size; i++)
        {
            if (bs[i])
                calculate(v[i]);
        }
    }

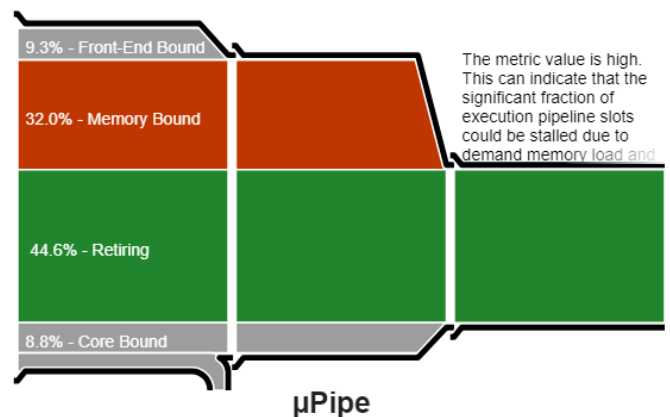
    end = std::chrono::system_clock::now();
    elapsed_mseconds += std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
    std::cout << "Time: " << elapsed_mseconds << " ms\n";
}

```

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.

Elapsed Time [Ⓢ]: 3.522s

| | | |
|--------------------------------------|------------------------|-------------------|
| Clockticks: | 94,015,844,745 | |
| Instructions Retired: | <u>161,717,509,545</u> | |
| CPI Rate [Ⓢ] : | 0.581 | |
| MUX Reliability [Ⓢ] : | 1.000 | |
| Retiring [Ⓢ] : | 44.6% | of Pipeline Slots |
| Front-End Bound [Ⓢ] : | 9.3% | of Pipeline Slots |
| Bad Speculation [Ⓢ] : | 5.3% | of Pipeline Slots |
| Back-End Bound [Ⓢ] : | 40.8% | of Pipeline Slots |
| Memory Bound [Ⓢ] : | 32.0% | of Pipeline Slots |
| L1 Bound [Ⓢ] : | 2.6% | of Clockticks |
| L2 Bound [Ⓢ] : | 9.4% | of Clockticks |
| L3 Bound [Ⓢ] : | 6.3% | of Clockticks |
| Contested Accesses [Ⓢ] : | 3.3% | of Clockticks |
| Data Sharing [Ⓢ] : | 0.9% | of Clockticks |
| L3 Latency [Ⓢ] : | <u>75.6%</u> | of Clockticks |
| SQ Full [Ⓢ] : | 0.7% | of Clockticks |
| DRAM Bound [Ⓢ] : | 8.7% | of Clockticks |
| Store Bound [Ⓢ] : | 0.0% | of Clockticks |
| Core Bound [Ⓢ] : | 8.8% | of Pipeline Slots |
| Average CPU Frequency [Ⓢ] : | 4.4 GHz | |
| Total Thread Count: | 1 | |
| Paused Time [Ⓢ] : | 0s | |



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

2.4 Третья версия программы

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

```
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;
};
```

Главный вычислительный цикл следующий принял вид:

```
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]);
        }
    }
}
```

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











3. Выводы

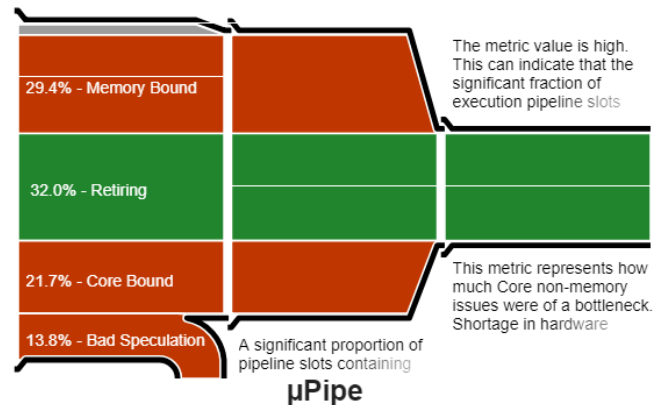
Анализ с помощью Amplifier показал, что общее число успешно выполненных инструкций уменьшилось со 160 млн. до 34 млн., проблемы с оперативной памятью почти полностью «перешли» на кэш L3.

Elapsed Time [?]: 1.000s 

| | | |
|---|--|-------------------|
| CPU Time [?] : | 5.172s | |
| Memory Bound [?] : | 29.0%  | of Pipeline Slots |
| L1 Bound [?] : | 1.7% | of Clockticks |
| L2 Bound [?] : | 3.5% | of Clockticks |
| L3 Bound [?] : | 15.3%  | of Clockticks |
| DRAM Bound [?] : | 4.0% | of Clockticks |
| DRAM Bandwidth Bound [?] : | 0.0% | of Elapsed Time |
| Loads: | 7,131,113,927 | |
| Stores: | 3,450,403,509 | |
| LLC Miss Count [?] : | 4,200,294 | |
| Average Latency (cycles) [?] : | 27 | |
| Total Thread Count: | 9 | |
| Paused Time [?] : | 0s | |

Elapsed Time [?]: 0.936s

| | | |
|--|--|-------------------|
| Clockticks: | 22,968,651,390 | |
| Instructions Retired: | 34,481,891,205 | |
| CPI Rate [?] : | 0.666 | |
| MUX Reliability [?] : | 1.000 | |
| Retiring [?] : | 32.0% | of Pipeline Slots |
| Front-End Bound [?] : | 3.1% | of Pipeline Slots |
| Bad Speculation [?] : | 13.8%  | of Pipeline Slots |
| Branch Mispredict [?] : | 13.6%  | of Pipeline Slots |
| Machine Clears [?] : | 0.2% | of Pipeline Slots |
| Back-End Bound [?] : | 51.1%  | of Pipeline Slots |
| Memory Bound [?] : | 29.4%  | of Pipeline Slots |
| L1 Bound [?] : | 3.3% | of Clockticks |
| L2 Bound [?] : | 4.5% | of Clockticks |
| L3 Bound [?] : | 16.6%  | of Clockticks |
| Contested Accesses [?] : | 4.9% | of Clockticks |
| Data Sharing [?] : | 3.0% | of Clockticks |
| L3 Latency [?] : | 100.0%  | of Clockticks |
| SQ Full [?] : | 2.3%  | of Clockticks |
| DRAM Bound [?] : | 3.4% | of Clockticks |
| Store Bound [?] : | 0.2% | of Clockticks |
| Core Bound [?] : | 21.7%  | of Pipeline Slots |
| Divider [?] : | 0.0% | of Clockticks |
| Port Utilization [?] : | 20.7%  | of Clockticks |
| Cycles of 0 Ports Utilized [?] : | 1.1% | of Clockticks |
| Cycles of 1 Port Utilized [?] : | 19.6% | of Clockticks |
| Cycles of 2 Ports Utilized [?] : | 18.4% | of Clockticks |
| Cycles of 3+ Ports Utilized [?] : | 27.7% | of Clockticks |
| Vector Capacity Usage (FPU) [?] : | 94.3%  | |
| Average CPU Frequency [?] : | 4.3 GHz | |
| Total Thread Count: | 1 | |
| Paused Time [?] : | 0s | |



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

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

Vectorization[?]: 92.4% of Packed FP Operations[?]

Instruction Mix:

| | | |
|---|---------|------------|
| SP FLOPs [?] : | 0.0% | of uOps |
| Packed [?] : | 27.4% | from SP FP |
| 128-bit [?] : | 13.0% 🚩 | from SP FP |
| 256-bit [?] : | 14.4% | from SP FP |
| Scalar [?] : | 72.6% 🚩 | from SP FP |
| DP FLOPs [?] : | 20.3% | of uOps |
| x87 FLOPs [?] : | 0.0% | of uOps |
| Non-FP [?] : | 79.7% | of uOps |
| FP Arith/Mem Wr Instr. Ratio [?] : | 1.835 | |

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.

4. 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();
}

```