

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра вычислительной техники

ОТЧЕТ
по лабораторной работе № 2
по дисциплине «Параллельные алгоритмы и системы»
Тема: Оптимизация программы

Студент гр 9306

Преподаватель

Евдокимов О.В.

Пазников А. А.

Санкт-Петербург
2023

Оглавление

| | |
|----------------------------------------------------------|----|
| Цель работы..... | 3 |
| Задание..... | 3 |
| Ход работы..... | 3 |
| Описание алгоритма. Базовая реализация..... | 3 |
| Код программы..... | 4 |
| Многопоточная реализация..... | 5 |
| Код программы..... | 6 |
| Оптимизация при помощи векторизации..... | 7 |
| Код программы..... | 8 |
| Использование средств профилирования..... | 9 |
| Код программы..... | 9 |
| Оптимизация работы конвейера. Разворачивание циклов..... | 10 |
| Код программы..... | 10 |
| Код программы..... | 11 |
| Выводы..... | 11 |

Цель работы

Изучение методов оптимизации программ для более полного использования аппаратных мощностей системы.

Задание

Оптимизировать программу выполняющую вычисление числа Пи.

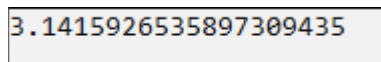
Ход работы

Описание алгоритма. Базовая реализация.

Для вычисления воспользуемся следующей формулой, позволяющей вычислить число π с различно точностью зависящей от параметра N :

$$\pi = \left(\frac{4}{1+x_0^2} + \frac{4}{1+x_1^2} + \dots + \frac{4}{1+x_{N-1}^2} \right) * \frac{1}{N}, \text{ где } x_i = (i+0,5) * \frac{1}{N}, i = \overline{0, N-1}$$

На рисунке 1 приведен пример рассчитанного значения при параметре $N=10000000$.



3.1415926535897309435

Рисунок 1 — демонстрация работы алгоритма.

Проведем измерения времени вычисления данной формулы, запустив вычисление 128 раз. Также проведем тестирование и анализ влияния различных компиляторов на скорость работы программы. Будем использовать три различных компилятора GCC (Msys64 MinGW64 GNU 12.2.0), Clang (Visual Studio clang_cl_x64_x64 Clang 15.0.1), MSVC (MSVC 19.35.32216.1) с ключами оптимизации -O0. На рисунке 1 приведены график ящиков с усами времени с удаленными выбросами (слева) и столбчатая диаграмма математического ожидания (справа). По данным графикам мы видим, что MinGW ушел в большой отрыв, примерно пятикратный. Удивительно странный результат.

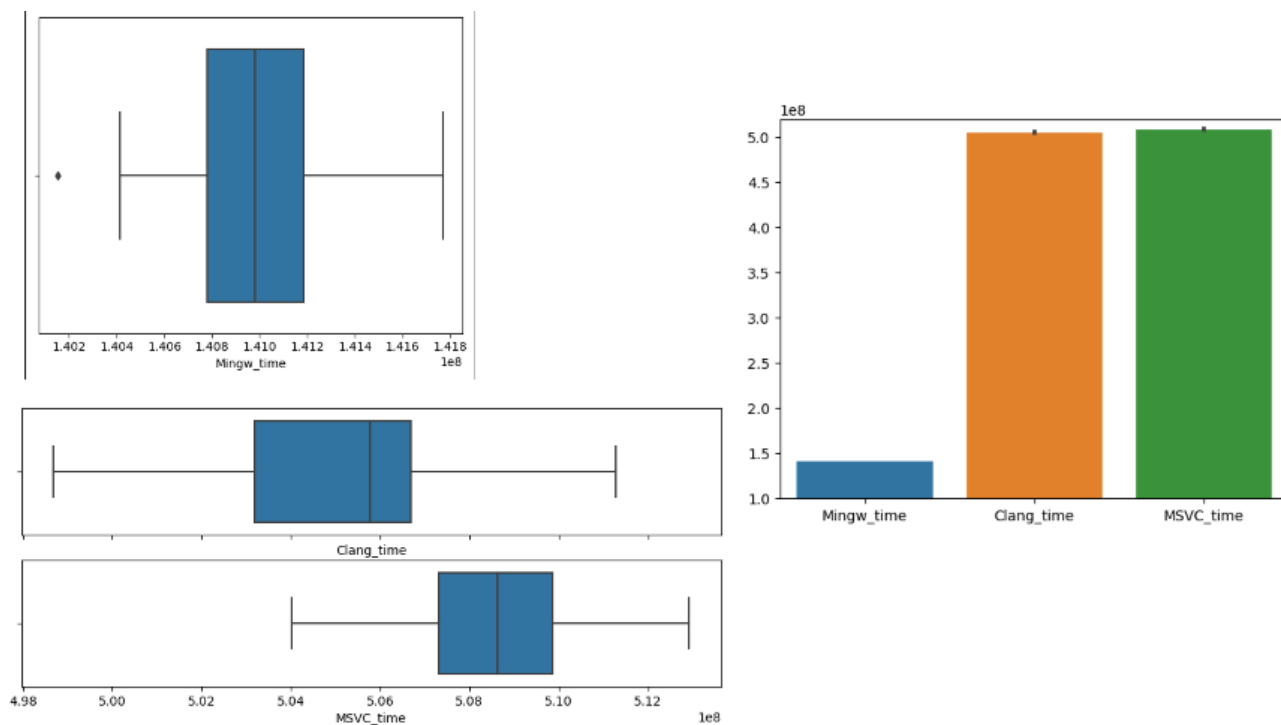


Рисунок 2 — графики со статистиками времени для базовой реализации.

Код программы

Приведем ниже код программы выполняющей ранее описанные действия. Результаты измерения времени каждой итерации вычислений выводятся в .csv файл. Данная программа сразу написана так, чтобы в дальнейшем мы могли проверять на корректность новые реализации (x_imp) сравнивая ответ с данной реализацией (x_base от $calc_base$)

```
constexpr long long N = 10000000;
constexpr int max_iter = 128;

double calc_base()
{
    double sum = 0;
    for(long i = 0; i < N; ++i)
    {
        sum += 4. / (1. + std::pow((i + 0.5) / N, 2.));
    }
    sum /= N;

    return sum;
}

bool is_eq(double l, double r)
{
    double epsilon = 0.1;
    bool res = true;

    double d = fabs(l - r);
```

```

        if (!(d < epsilon))
            res = false;

        return res;
}

int main()
{
    srand(std::chrono::system_clock::now().time_since_epoch().count());

    std::fstream file;
    file.open("TimeMeasure.csv", std::ios_base::out | std::ios_base::trunc);
    file << "time\n";

    //double x_base = calc_base();
    //double x_imp = calc_omp();
    //
    //if (!is_eq(res_base, res_imp))
    //{
    //    std::cout << "Test not pass\n";
    //    throw std::runtime_error("Test not pass");
    //}

    for (int cur_iter = 0; cur_iter < max_iter; ++cur_iter)
    {
        std::cout << cur_iter << std::endl;

        std::chrono::time_point<std::chrono::high_resolution_clock> t1 =
std::chrono::high_resolution_clock::now();

        double res_imp = calc_base();

        auto elapsed_time = std::chrono::high_resolution_clock::now() -
t1;
        file << elapsed_time.count() << "\n";
    }

    file.close();

    return 0;
}

```

Многопоточная реализация

Дополним нашу программу распараллеливанием, чтобы использовать имеющиеся на устройстве ядра процессора. Для этого будем использовать достаточно популярную библиотеку OpenMP.

Все что нам нужно для этого сделать добавить директиву `#pragma omp parallel for`. Затем вместо того чтобы вручную писать суммирование частных сумм каждого процессора (редукцию) с использованием критической секции или атомарной операцией. Мы можем указать дополнительную директиву `reduction(+:sum)`, которая сделает все за нас.

Давайте проведем измерения времени. Эксперимент будем проводить для трех ранее указанных компиляторов с их версиями OpenMp: Clang (5.0), MinGW (4.5), MSVC (2.0). Без явного указания количества потоков (автоматический выбор OpenMP) получили результаты

представленные на рисунке 3. По данным графикам мы видим, что производительность улучшилась на порядок.

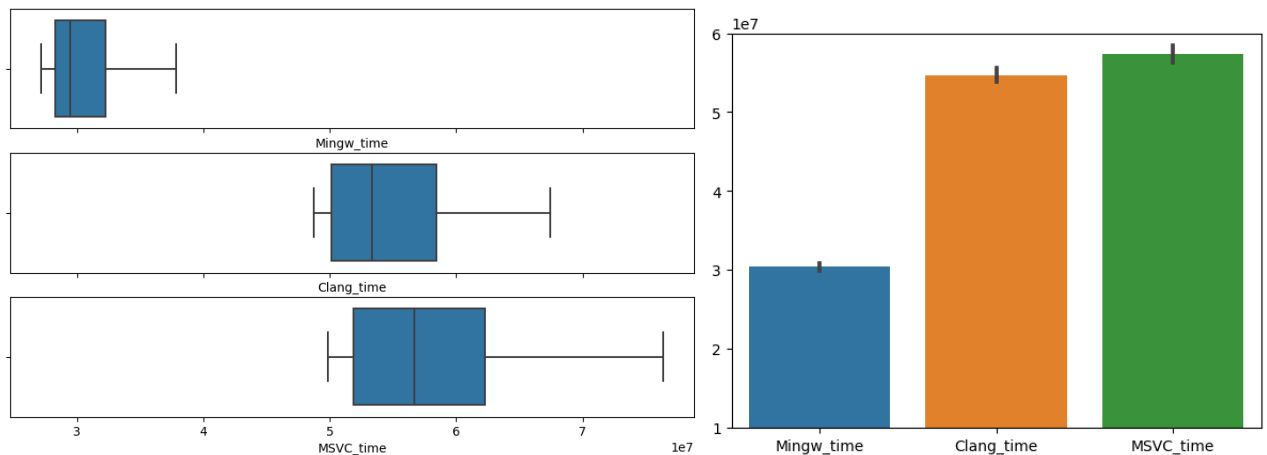


Рисунок 3 — графики статистик много поточной реализации.

Давайте проведем эксперименты по количеству потоков. В качестве компилятора выберем MinGW, как показывающий наилучший результат. На рисунке 4 показан график изменения времени вычисления (слева) в зависимости от количества явно указанных потоков для блока `parallel for` и относительный прирост, справа. Как мы видим по данным графикам, наилучшая производительность достигается при использовании 12 потоков, что логично, поскольку система на, которой запускается программа, тоже имеет 12 потоков процессора.

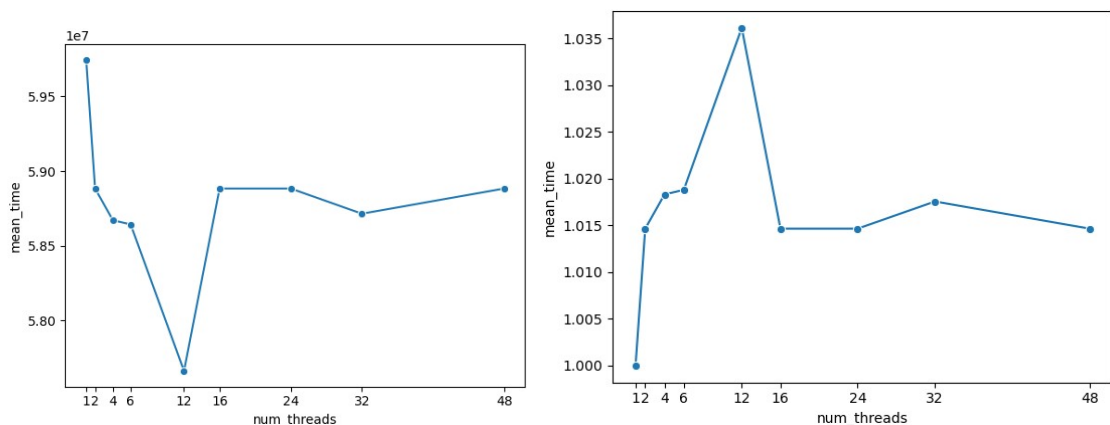


Рисунок 4 — график эффективности относительно количества потоков.

Код программы

```
double calc_omp()
{
    double sum = 0;
    #pragma omp parallel for num_threads(24) reduction(+:sum)
    for(long i = 0; i < N; ++i)
    {
        sum += 4. / (1 + std::pow((i + 0.5) / N, 2.));
    }
    sum /= N;
    return sum;
}
```

}

Оптимизация при помощи векторизации

В современных процессорах существуют инструкции позволяющие выполнять одинаковые действия над несколькими значениями одновременно. Использование таких инструкций должно увеличить показатели времени выполнения давайте это проверим. Использовать будем AVX инструкции поскольку требуются числа с плавающей точкой двойной точности (double).

Векторизацию мы добавим вместе с много поточностью. Однако теперь нам нужно отказаться от директивы parallel for и перейти к просто директиве parallel, поскольку нужно выделять диапазоны самостоятельно, а вот reduction можно оставить.

Для вычислений мы выделяем ряд констант в векторных регистрах: содержащую N, 1 и 4. Делим работу поровну между потоками. Затем назначаем каждому потоку начало его диапазона и конец, для последнего потока концом будет N-1, для обработки случаев если N не кратно количеству потоков. Нацело делим начало и конец на количество элементов в векторе это будут новые групповые начало и конец. Затем в цикле по группам выполняем установку значений вектора, деление на N, возведение в квадрат, прибавляем 1, делим 4 на полученное, аккумулируем результат в векторной сумме.

По окончании группового цикла нужно провести редукцию вектора суммы однако в случае использования AVX нельзя воспользоваться двойным вызовом инструкции горизонтальной суммы (hadd), как в случае SSE. Поскольку данная инструкция работает иначе суммируя младшие и старшие части по отдельности. После проведения одной горизонтальной суммы просуммируем первый и третий элемент, так и получим значение суммы.

Проведем измерения времени, результаты представлены на рисунке 5. Как мы можем видеть добавление векторизации дало трехкратный прирост производительности.

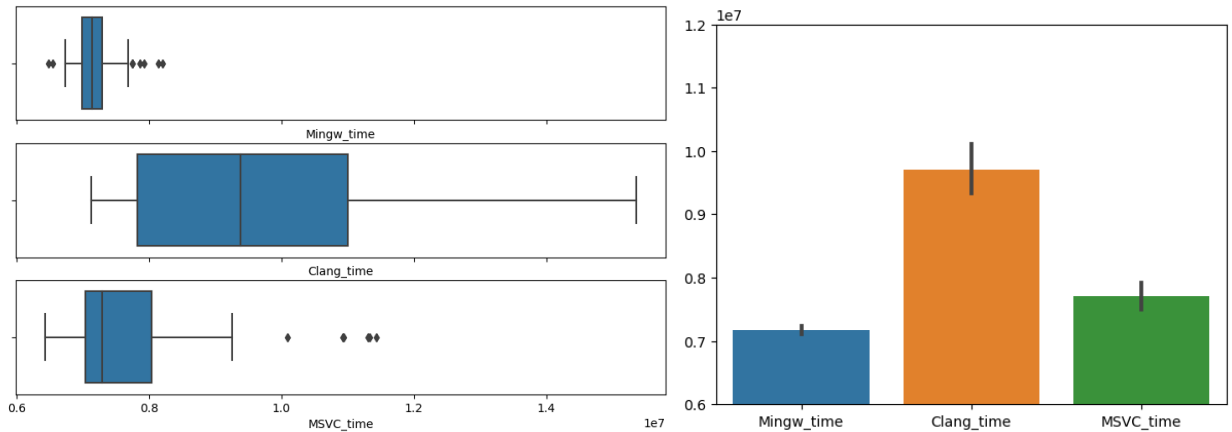


Рисунок 5 — графики времени выполнения с векторизацией.

Код программы

```
double calc_vec_256()
{
    const int count_of_double_in_reg = sizeof(__m256) / sizeof(double);
    const __m256d c_N = _mm256_set1_pd(N);
    const __m256d c_1 = _mm256_set1_pd(1);
    const __m256d c_4 = _mm256_set1_pd(4);
    double sh_sum = 0;
#pragma omp parallel num_threads(12) reduction(+:sh_sum)
    {
        __m256d v_sum = _mm256_set1_pd(0);
        const int nthreads = omp_get_num_threads();

        int thread_id = omp_get_thread_num();
        long long items_per_thread = N / nthreads;
        long long lb = thread_id * items_per_thread;
        long long ub = (thread_id == nthreads - 1) ? (N - 1) : (lb +
items_per_thread - 1);

        long end = ub / count_of_double_in_reg;
        long g = lb / count_of_double_in_reg;
        for (; g < end; ++g)
        {
            long i = g * count_of_double_in_reg;
            __m256d c = _mm256_set_pd(i + 0.5, i + 1.5, i + 2.5, i +
3.5);

            __m256d div = _mm256_div_pd(c, c_N);
            __m256d sqr = _mm256_mul_pd(div, div);
            __m256d sum = _mm256_add_pd(c_1, sqr);
            __m256d res = _mm256_div_pd(c_4, sum);

            v_sum = _mm256_add_pd(v_sum, res);
        }

        v_sum = _mm256_hadd_pd(v_sum, v_sum);
        sh_sum += reinterpret_cast<double*>(&v_sum)[0] +
reinterpret_cast<double*>(&v_sum)[2];

        if (thread_id == nthreads - 1)
        {
            for (long i = g * count_of_double_in_reg; i < N; ++i)
            {
                sh_sum += 4. / (1 + std::pow((i + 0.5) / N,
2.));
            }
        }
    }
}
```



```

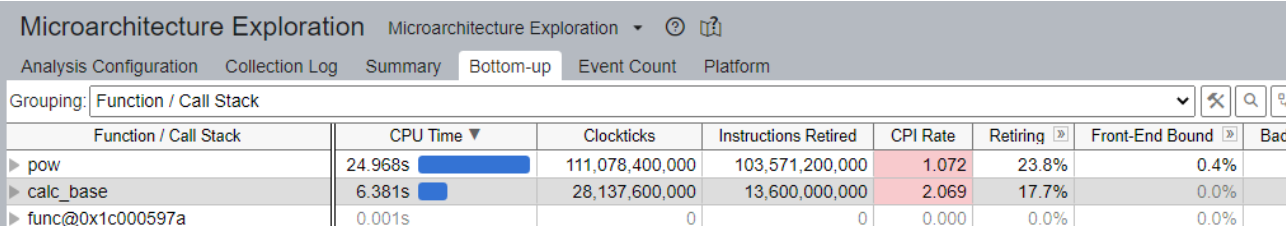
    }
}
sh_sum /= N;
return sh_sum;
}

```

Использование средств профилирования

Давайте вернемся к базовой реализации и проанализируем нашу программу при помощи средства профилирования Intel Vtune Profiler. В качестве компилятора будем использовать MSVC как наиболее родной для системы Windows.

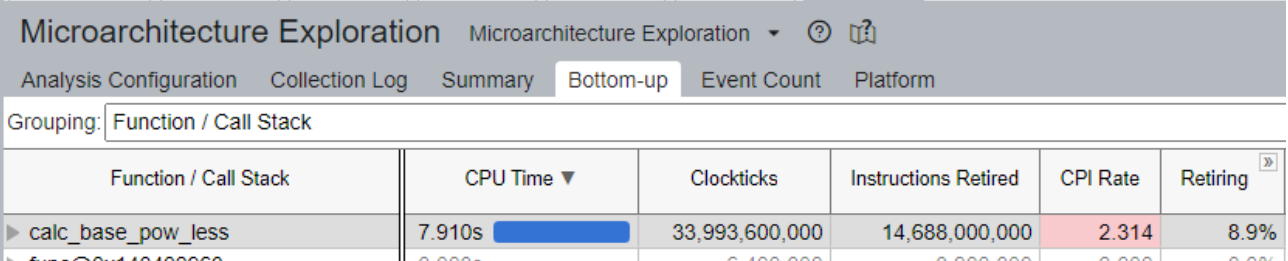
Запустим анализ микроархитектурных показателей, рисунок с таблицей результатов представлен на рисунке 6. Как мы можем видеть использовать операцию возведения в степень вероятно было ошибкой, поскольку ее вычисление занимает наибольшее время.



| Function / Call Stack | CPU Time | Clockticks | Instructions Retired | CPI Rate | Retiring | Front-End Bound | Back-End Bound |
|-----------------------|----------|-----------------|----------------------|----------|----------|-----------------|----------------|
| pow | 24.968s | 111,078,400,000 | 103,571,200,000 | 1.072 | 23.8% | 0.4% | 0.0% |
| calc_base | 6.381s | 28,137,600,000 | 13,600,000,000 | 2.069 | 17.7% | 0.0% | 0.0% |
| func@0x1c000597a | 0.001s | 0 | 0 | 0.000 | 0.0% | 0.0% | 0.0% |

Рисунок 6 — таблица результатов профилирования 1

Давайте уберем из кода вызов функции pow на простое умножение. Назовем эту реализацию calc_base_pow_less. Таблица из профилировщика приведена на рисунке 7. Как мы видим время выполнения функции вычисления уменьшилось более чем в 2 раза. Программа стала работать заметно быстрее. Возможно именно из-за такой мини оптимизации MinGW и обходил остальные компиляторы.



| Function / Call Stack | CPU Time | Clockticks | Instructions Retired | CPI Rate | Retiring | Front-End Bound | Back-End Bound |
|-----------------------|----------|----------------|----------------------|----------|----------|-----------------|----------------|
| calc_base_pow_less | 7.910s | 33,993,600,000 | 14,688,000,000 | 2.314 | 8.9% | 0.0% | 0.0% |
| func@0x1c000597a | 0.001s | 0 | 0 | 0.000 | 0.0% | 0.0% | 0.0% |

Рисунок 7 — таблица результатов профилирования 2

Код программы

```

double calc_base_pow_less()
{
    double sum = 0;
    for (long i = 0; i < N; ++i)
    {
        double div = (i + 0.5) / N;

        sum += 4. / (1. + div*div);
    }
    sum /= N;
}

```

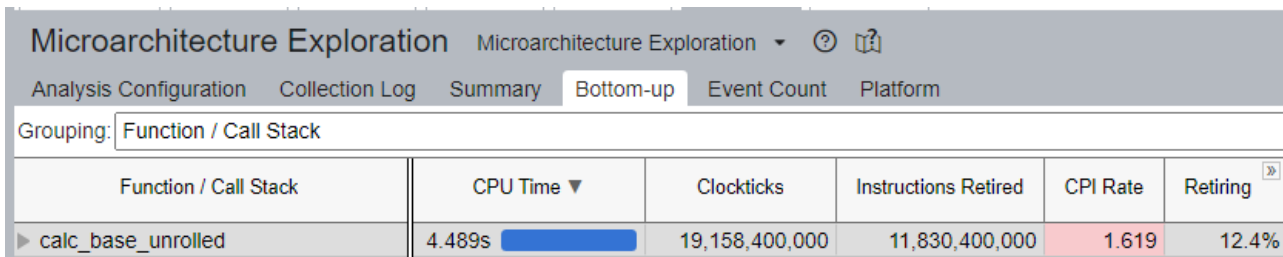
```

    return sum;
}

```

Оптимизация работы конвейера. Разворачивание циклов

По рисункам с прошлой части мы можем видеть, что у функции имеется CPI (Cycles Per Instruction | циклы на инструкцию) = 2,3. Чем больше это значение тем хуже. На инструкцию может тратиться меньше циклов если инструкции будут хорошо ложиться на конвейер без дыр. Давайте попробуем увеличить это значение развернув цикл на 4 шага вперед. Результаты проектировщика приведены на рисунке 8. Как мы можем видеть CPI уменьшился на 0,4, время выполнения приложения также уменьшилось в почти два раза.



The screenshot shows the 'Microarchitecture Exploration' window with the 'Bottom-up' tab selected. The 'Grouping' is set to 'Function / Call Stack'. The table below displays performance data for the 'calc_base_unrolled' function.

| Function / Call Stack | CPU Time ▼ | Clockticks | Instructions Retired | CPI Rate | Retiring ▸ |
|-----------------------|------------|----------------|----------------------|----------|------------|
| ► calc_base_unrolled | 4.489s | 19,158,400,000 | 11,830,400,000 | 1.619 | 12.4% |

Рисунок 8 — таблица результатов профилирования 3

Код программы

```

double calc_base_unrolled()
{
    double sum = 0;

    for (long g = 0; g < N/4; ++g)
    {
        long i = g*4;

        double div0 = (i + 0.5) / N;
        double div1 = (i + 1.5) / N;
        double div2 = (i + 2.5) / N;
        double div3 = (i + 3.5) / N;

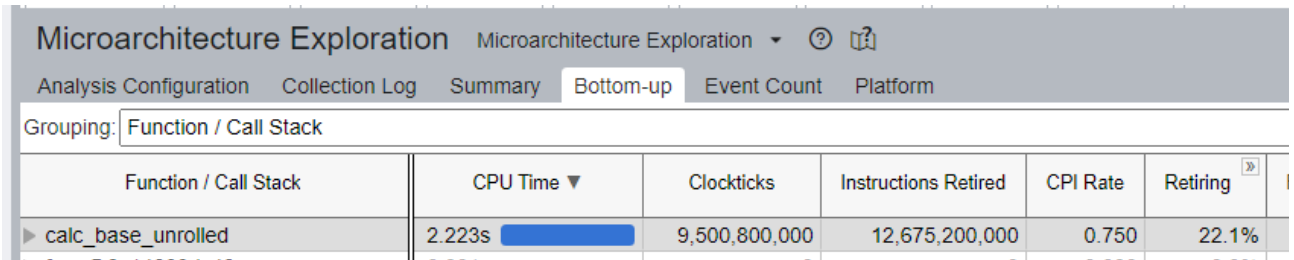
        sum += 4. / (1. + div0*div0);
        sum += 4. / (1. + div1*div1);
        sum += 4. / (1. + div2*div2);
        sum += 4. / (1. + div3*div3);
    }
    sum /= N;

    return sum;
}

```

Подумав над реализацией еще раз получилось уменьшить CPI до 0.7 и время еще. Для этого всего лишь надо было добавить переменную частной суммы внутри цикла, которая затем добавляется к основной сумме. Вероятно, это уменьшает зависимость данных между двумя итерациями и уменьшает количество простоев конвейера по зависимости. Такой вывод можно сделать по уменьшению показателя Memory Bound с 60% до 22%. На рисунке 9 приведена таблица из профилировщика. Данное исправление дало нам двух кратное

уменьшение времени выполнения.



The screenshot shows the 'Microarchitecture Exploration' interface with the 'Bottom-up' tab selected. The 'Grouping' is set to 'Function / Call Stack'. The table below displays the performance metrics for the 'calc_base_unrolled' function.

| Function / Call Stack | CPU Time ▼ | Clockticks | Instructions Retired | CPI Rate | Retiring |
|-----------------------|------------|---------------|----------------------|----------|----------|
| ► calc_base_unrolled | 2.223s | 9,500,800,000 | 12,675,200,000 | 0.750 | 22.1% |

Рисунок 9 — таблица результатов профилирования 4

Код программы

```
double calc_base_unrolled()
{
    double sum = 0;

    for (long g = 0; g < N/4; ++g)
    {
        double p_sum = 0;
        long i = g*4;

        double div0 = (i + 0.5) / N;
        double div1 = (i + 1.5) / N;
        double div2 = (i + 2.5) / N;
        double div3 = (i + 3.5) / N;

        p_sum += 4. / (1. + div0*div0);
        p_sum += 4. / (1. + div1*div1);
        p_sum += 4. / (1. + div2*div2);
        p_sum += 4. / (1. + div3*div3);

        sum+=p_sum;
    }
    sum /= N;

    return sum;
}
```

Выводы

В ходе выполнения лабораторной работы мы изучили различные методы оптимизации, такие как: многопоточность, векторизация, разворачивание циклов. При исследовании многопоточности мы узнали что лучшая производительность достигается при 12 потоках. При исследовании векторизации мы изучили отличия AVX (double) от SSE. С помощью средств профилирования научились находить не эффективные части программ. При разворачивании циклов мы научились обнаруживать и исправлять зависимости по данным