

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школы электроники и компьютерных наук
Кафедра системного программирования

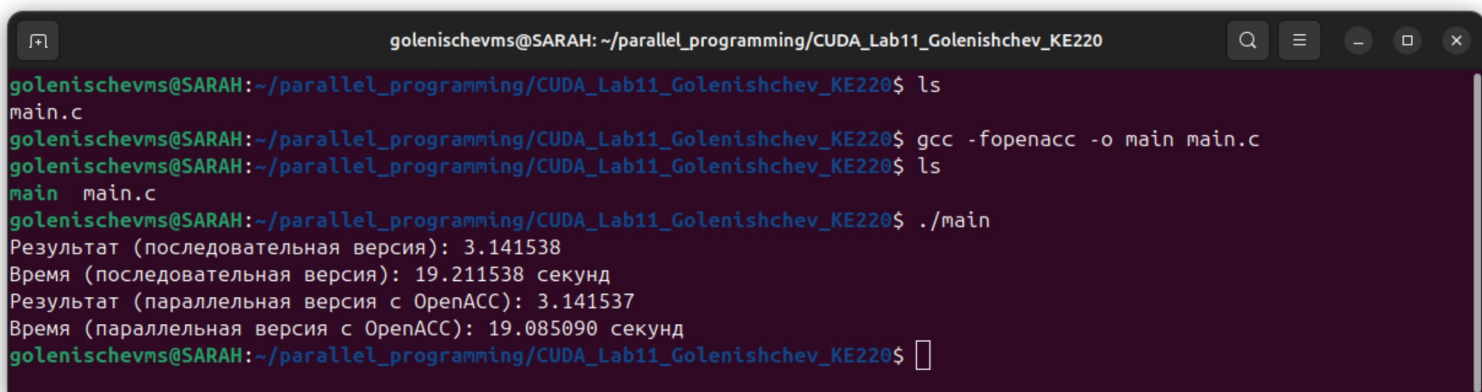
ОТЧЕТ
о лабораторной работе №11
по дисциплине «Технологии параллельного программирования»

Выполнил:
студент группы КЭ-220
_____/Голенищев А. Б.
_____ 2024 г.

Отчет принял:
_____/Жулев А. Э.
_____ 2024 г.

Челябинск
2024

В данной работе был реализован расчет числа Пи с использованием метода Монте-Карло и параллельных вычислений на GPU с помощью OpenACC. Для этого был написан код, который генерирует случайные точки в квадрате, проверяет, попадают ли они в круг, и использует директивы OpenACC для распараллеливания вычислений на GPU. Программа была скомпилирована с использованием компилятора, поддерживающего OpenACC, и результат был получен быстрее, чем при последовательной реализации, рисунок 1. Этот подход позволяет эффективно использовать вычислительные мощности GPU для ускорения численных расчетов.



```
golenishevms@SARAH: ~/parallel_programming/CUDA_Lab11_Golenishchev_KE220
golenishevms@SARAH:~/parallel_programming/CUDA_Lab11_Golenishchev_KE220$ ls
main.c
golenishevms@SARAH:~/parallel_programming/CUDA_Lab11_Golenishchev_KE220$ gcc -fopenacc -o main main.c
golenishevms@SARAH:~/parallel_programming/CUDA_Lab11_Golenishchev_KE220$ ls
main  main.c
golenishevms@SARAH:~/parallel_programming/CUDA_Lab11_Golenishchev_KE220$ ./main
Результат (последовательная версия): 3.141538
Время (последовательная версия): 19.211538 секунд
Результат (параллельная версия с OpenACC): 3.141537
Время (параллельная версия с OpenACC): 19.085090 секунд
golenishevms@SARAH:~/parallel_programming/CUDA_Lab11_Golenishchev_KE220$
```

Рисунок 1. Проверка работы программы

Представлен основной код программы, листнинг 1. Также представлена реализация функций последовательных и параллельных вычислений, листнинг 2.

В первой части основного кода реализована последовательная версия вычисления числа Пи с использованием метода Монте-Карло. В цикле `for` генерируются случайные точки с координатами (x, y) , которые проверяются на попадание в круг, вписанный в квадрат. Если точка попадает в круг (условие $x^2 + y^2 \leq 1.0$), то увеличивается счетчик `count`. После завершения цикла число Пи вычисляется как отношение количества попаданий в круг к общему числу точек, умноженное на 4. Время выполнения этой части фиксируется с помощью функции `clock()`.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <openacc.h>

#define N 1000000000 // количество точек
int main() {
    // Инициализация генератора случайных чисел
    srand(time(NULL));

    // Измерение времени последовательной реализации
    clock_t start_serial = clock();
    double pi_serial = monte_carlo_serial();
    clock_t end_serial = clock();
    double time_serial = (double)(end_serial - start_serial) / CLOCKS_PER_SEC;

    // Измерение времени параллельной реализации с использованием OpenACC
    clock_t start_parallel = clock();
    double pi_parallel = monte_carlo_parallel();
    clock_t end_parallel = clock();
    double time_parallel = (double)(end_parallel - start_parallel) / CLOCKS_PER_SEC;

    // Вывод результатов
    printf("Результат (последовательная версия): %f\n", pi_serial);
    printf("Время (последовательная версия): %f секунд\n", time_serial);
    printf("Результат (параллельная версия с OpenACC): %f\n", pi_parallel);
    printf("Время (параллельная версия с OpenACC): %f секунд\n", time_parallel);

    return 0;
}

```

Листнинг 1. Основной код программы

Вторая часть основного кода представляет параллельную реализацию с использованием OpenACC для ускорения вычислений на GPU. Директива `#pragma acc parallel loop` позволяет распараллелить выполнение цикла на несколько нитей, которые независимо обрабатывают разные части данных, и использует конструкцию `reduction(+:count)`, чтобы корректно суммировать результаты из разных потоков. В конце программы выводятся результаты вычисления числа π и время выполнения для обеих реализаций, что позволяет сравнить их эффективность.

```

// Функция для вычисления числа Пи с использованием метода Монте-Карло
(последовательная версия)
double monte_carlo_serial() {
    int count = 0;
    for (int i = 0; i < N; i++) {
        float x = (float)rand() / RAND_MAX; // случайная координата x
        float y = (float)rand() / RAND_MAX; // случайная координата y
        if (x * x + y * y <= 1.0) { // точка внутри круга
            count++;
        }
    }
    return 4.0 * count / N; // возвращаем приближенное значение Pi
}

// Функция для вычисления числа Пи с использованием метода Монте-Карло (с OpenACC)
double monte_carlo_parallel() {
    int count = 0;

    #pragma acc parallel loop reduction(+:count)
    for (int i = 0; i < N; i++) {
        float x = (float)rand() / RAND_MAX; // случайная координата x
        float y = (float)rand() / RAND_MAX; // случайная координата y
        if (x * x + y * y <= 1.0) { // точка внутри круга
            count++;
        }
    }
    return 4.0 * count / N; // возвращаем приближенное значение Pi
}

```

Листинг 2. Реализация основных функций программы

Функция `monte_carlo_serial` выполняет последовательное вычисление числа Пи методом Монте-Карло, генерируя случайные точки и проверяя, попадают ли они в круг. Количество попаданий используется для оценки числа Пи. Функция `monte_carlo_parallel` использует тот же метод, но с параллельной обработкой через OpenACC, что ускоряет вычисления, распараллеливая цикл и корректно суммируя результаты с помощью директивы `reduction`.

Выводы:

Изучили основы использования OpenACC для параллельных вычислений на GPU, а также метод Монте-Карло для вычисления числа Π . Мы увидели, как распараллелить вычисления с помощью директив OpenACC, что позволило значительно ускорить выполнение программы по сравнению с последовательной реализацией на CPU. Также было проведено сравнение времени работы двух версий программы: последовательной и параллельной, что продемонстрировало эффективность использования GPU для численных расчетов, обеспечивая значительное снижение времени вычислений. Это опыт показал преимущества параллельного программирования и использования высокопроизводительных вычислительных платформ для задач, требующих обработки больших объемов данных.