Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Архитектура вычислительных систем»

**ОТЧЕТ**

к лабораторной работе №1-3

на тему:

**«ПРОЦЕССОРЫ РАЗЛИЧНЫХ СЕМЕЙСТВ И ПОКОЛЕНИЙ В РЕЖИМЕ ОДНОГО И МНОЖЕСТВА ЯДЕР. ГРАФИЧЕСКИЕ ПРОЦЕССОРЫ РАЗЛИЧНЫХ СЕМЕЙСТВ И ПОКОЛЕНИЙ»**

БГУИР 6-05-0612-02 67

|  |
| --- |
| Выполнил студент группы 353503  КОХАН Артём Игоревич |
|  |
| (дата, подпись студента) |
| Проверил ассистент каф. информатики  КАЛИНОВСКАЯ Анастасия Александровна |
|  |
| (дата, подпись преподавателя) |

Минск 2025

# 1 Индивидуальное задание

**Зaдание**

1 Осуществить методом математического выполнения функции согласно варианту задания. Значение аргумента x изменяется от a до b с шагом h. Для каждого x найти значения функции Y(x), суммы S(x) и число итераций n, при котором достигается требуемая точность ε = |Y(x)-S(x)|. Результат вывести в виде таблицы. Значения a, b, h и ε вводятся с клавиатуры.

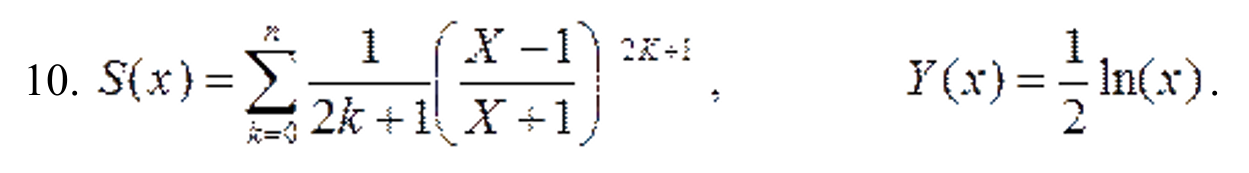


Рисунок 1 – Заданные функции

2 Осуществить написание программы на любом удобном языке программирования с вставками кода ассемблера для выполнения задачи на i количестве итерации (i>5000) для получения достоверных результатов эксперимента.

Выполнение задачи должно осуществляться в операционной системе без графической оболочки.

3 Результатом будут графики нагрузки на одно ядро процессора и на на все ядра процессора. График нагрузки на GPU.

По оси Х – время выполнения, по оси Y – количество итераций (i).

Intel – два графика на одно ядро, с функцией Hyper-Threading и без.

AMD – с функцией SMT и без.

ARM – одно низко производительное ядро и одно высокопроизводительное ядро.

# 2 Выполнение работы

Для выполнения задания был написан код на языке С++, который вычисляет логарифмическую функцию двумя способами: аналитически через стандартную математическую функцию log(x*)* и приближенно с помощью разложения в ряд. В программе используется вставка на языке ассемблера для процессоров архитектуры **x86** с использованием FPU (Floating Point Unit), которая является частью процессора и предназначена для выполнения операций с числами с плавающей запятой. FPU специально оптимизирован для работы с числами с плавающей запятой, поэтому операции на нем выполняются более эффективно.

#include <iostream>

#include <fstream>

#include <chrono>

#include <cmath>

using namespace std;

using namespace std::chrono;

double solveS(double x, double epsilon, int &iterations)

{

double result = 0.0;

double term;

double base = (x - 1) / (x + 1);

double base\_pow = base;

iterations = 0;

while (true)

{

term = 1.0 / (2 \* iterations + 1) \* base\_pow;

++iterations;

if (fabs(term) < epsilon)

break;

// Ассемблерная вставка для сложения (добавление term к result)

asm volatile(

"fldl %1;" // Загружаем term в FPU стек

"faddl %0;" // Складываем с result (верх стека + result)

"fstpl %0;" // Записываем обратно в result

: "=m"(result)

: "m"(term));

base\_pow \*= base \* base; // (2i+1..2i+3..2i+5,,2i+7..)

}

return result;

}

double solveY(double x){

return log(x) / 2;

}

int main()

{

double a, b, h, epsilon;

cout << "Введите a: ";

cin >> a;

cout << "Введите b: ";

cin >> b;

cout << "Введите шаг h: ";

cin >> h;

cout << "Введите epsilon: ";

cin >> epsilon;

ofstream outputFile("results.txt");

if (!outputFile)

{

cerr << "Ошибка открытия файла!" << endl;

return 1;

}

cout << " x | Y(x) | S(x) | Итерации | Время (сек) " << endl;

cout << "-------------------------------------------------" << endl;

for (double x = a; x <= b; x += h)

{

if(x <= 0) continue;

if(h <= 0 && epsilon <= 0) break;

int iterations = 0;

double resultY = solveY(x);

auto start = high\_resolution\_clock::now();

double resultS = solveS(x, epsilon, iterations);

auto end = high\_resolution\_clock::now();

duration<double> elapsedTime = end - start;

cout << x << " | " << resultY << " | " << resultS << " | " << iterations << " | " << elapsedTime.count() << endl;

outputFile << iterations << " | " << elapsedTime.count() << endl;

}

outputFile.close();

return 0;

}

Данный код был запущен в 4 режимах: одноядерный с включённым/выключенным режимом SMT, многоядерный с включённым/выключенным режимом SMT.

Для того что бы запустить на одном физическом ядре, с помощью команды *cat /sys/devices/system/cpu/cpu\*/topology/core\_id* выясним какие потоки соответствуют каким физическим ядрам.

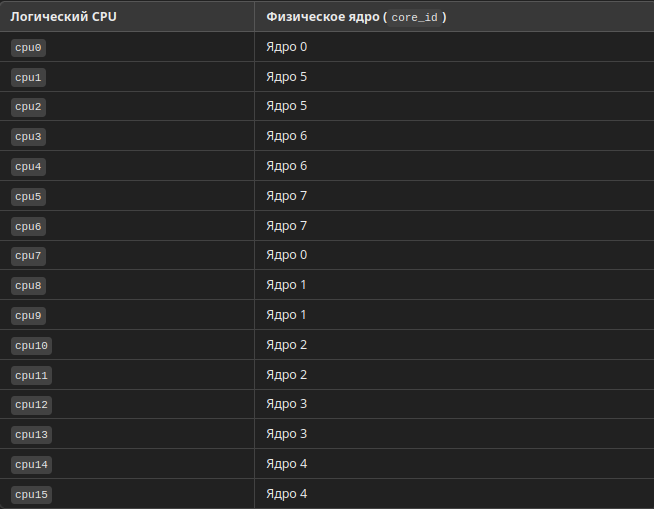


Рисунок 2 – Соответствие ядер и потоков

Например сначала будем запускать программу на первом физическом ядре, с помощью потоков 8,9: *taskset -c 8,9 ./main.*

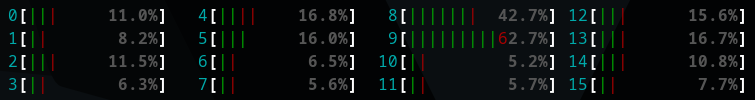


Рисунок 3 – Работа потоков 8,9 во время выполнения

В результате значений построим график зависимости времени выполнения программы от количества итераций.



Рисунок 4 – График выполнения в одноядерном режиме с включенным SMT

Дальше выключим режим SMT для выполнения программы в режиме в одноядерном режиме с выключенным SMT: *echo off | sudo tee /sys/devices/system/cpu/smt/control.* Запустим нашу программу *taskset -c 8 ./main.*

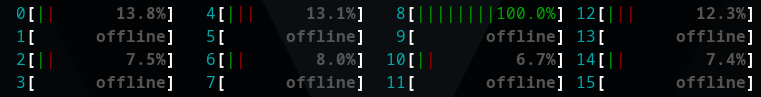
**

Рисунок 5 – Работа потока 8 во время выполнения

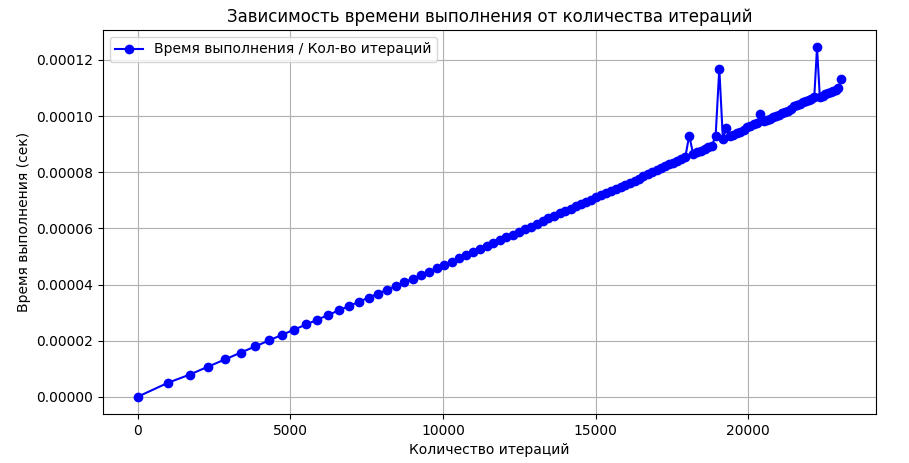


Рисунок 6 – График выполнения в одноядерном режиме с выключенным SMT

Аналогичные действия выполним в многоядерном режим, только явно укажем запуск на всех ядрах: *taskset -c 0-15 ./main.*

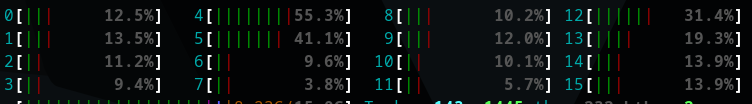


Рисунок 7 – Работа потоков 0-15 во время выполнения

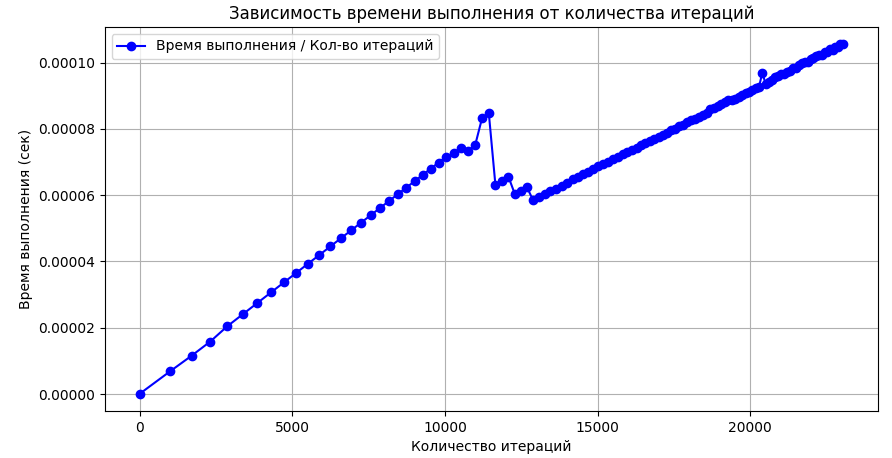


Рисунок 8 – График выполнения во многоядерном режиме с включенным SMT

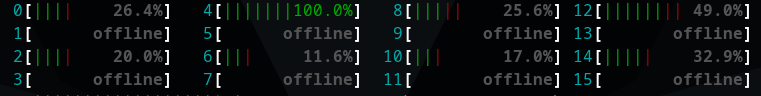


Рисунок 9 – Работа потоков 0, 2, 4, 6, 8, 10, 12, 14 во время выполнения

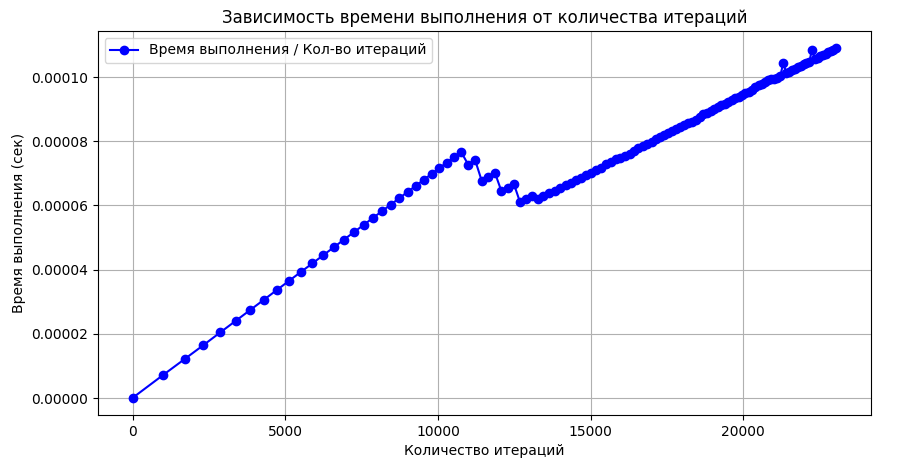


Рисунок 10 – График выполнения во многоядерном режиме с выключенным SMT

Соберём и запустим код предствленный ниже на CUDA: *nvcc -arch=sm\_86 main.cu -o main && ./main.*

#include <iostream>

#include <fstream>

#include <chrono>

#include <cmath>

#include <cuda\_runtime.h>

using namespace std;

using namespace std::chrono;

\_\_global\_\_ void solveSKernel(double x, double epsilon, double \*result, int \*converged, int maxIterations)

{

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

if (idx >= maxIterations || \*converged <= idx)

return; // Прерываем выполнение, если достигнута сходимость

double base = (x - 1) / (x + 1);

double term = 1.0 / (2 \* idx + 1) \* pow(base, 2 \* idx + 1);

// Атомарное сложение для накопления результата

atomicAdd(result, term);

// Проверяем условие сходимости

if (fabs(term) < epsilon)

{

atomicMin(converged, idx + 1); // Обновляем минимальное количество итераций

}

}

double solveS(double x, double epsilon, int &iterations)

{

const int maxIterations = 100000; // Ограничение на количество итераций

double result = 0.0;

int converged = maxIterations;

// Выделяем память на устройстве

double \*d\_result;

int \*d\_converged;

cudaMalloc((void \*\*)&d\_result, sizeof(double));

cudaMalloc((void \*\*)&d\_converged, sizeof(int));

// Инициализируем значения на устройстве

cudaMemset(d\_result, 0, sizeof(double));

cudaMemcpy(d\_converged, &converged, sizeof(int), cudaMemcpyHostToDevice);

// Конфигурация CUDA

int threadsPerBlock = 256;

int blocksPerGrid = (maxIterations + threadsPerBlock - 1) / threadsPerBlock;

// Запуск CUDA kernel

solveSKernel<<<blocksPerGrid, threadsPerBlock>>>(x, epsilon, d\_result, d\_converged, maxIterations);

// Синхронизация устройства

cudaDeviceSynchronize();

// Копируем результат обратно на хост

cudaMemcpy(&result, d\_result, sizeof(double), cudaMemcpyDeviceToHost);

cudaMemcpy(&converged, d\_converged, sizeof(int), cudaMemcpyDeviceToHost);

// Освобождаем память на устройстве

cudaFree(d\_result);

cudaFree(d\_converged);

// Устанавливаем количество итераций

iterations = converged;

return result;

}

// Вычисление Y(x) (не требует распараллеливания)

double solveY(double x)

{

return log(x) / 2;

}

int main()

{

double a, b, h, epsilon;

cout << "Введите a: ";

cin >> a;

cout << "Введите b: ";

cin >> b;

cout << "Введите шаг h: ";

cin >> h;

cout << "Введите epsilon: ";

cin >> epsilon;

ofstream outputFile("results.txt");

if (!outputFile)

{

cerr << "Ошибка открытия файла!" << endl;

return 1;

}

cout << " x | Y(x) | S(x) | Итерации | Время (сек) " << endl;

cout << "-------------------------------------------------" << endl;

for (double x = a; x <= b; x += h)

{

if (x <= 0)

continue;

if (h <= 0 && epsilon <= 0)

break;

int iterations = 0;

double resultY = solveY(x);

auto start = high\_resolution\_clock::now();

double resultS = solveS(x, epsilon, iterations);

auto end = high\_resolution\_clock::now();

duration<double> elapsedTime = end - start;

cout << x << " | " << resultY << " | " << resultS << " | " << iterations << " | " << elapsedTime.count() << endl;

outputFile << iterations << " | " << elapsedTime.count() << endl;

}

outputFile.close();

return 0;

}



Рисунок 11 – График выполнения для GPU при распараллеливании

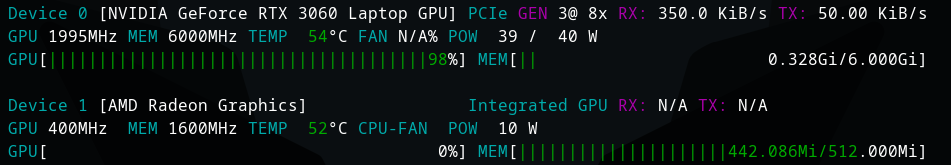


Рисунок 12– Работа GPU NVIDIA

# Вывод

В ходе выполнения практических работ была реализована интеграция ассемблерных вставок в проект на C++, что позволило глубже изучить особенности работы FPU и ассемблерных прерываний. Экспериментальная часть показала, что:

1 При запуске программы на одном ядре режим SMT обеспечивает незначительное ускорение вычислений. Также на графиках заметно использование режима SMT, примерно после 16000 итераций видно перераспределение ресурсов между потоками ядра.

2 При использовании множества ядер разница между режимами с включённым и выключенным SMT остаётся минимальной, но видно, что с включенным SMT рост времени выполнения более стабилен.

3 Запуск на GPU продемонстрировал немного иной результат. GPU обрабатывает множество потоков одновременно. Когда количество итераций увеличивается, вычисления распределяются по большому количеству ядер, и загрузка GPU остается практически одинаковой. Если задачи укладываются в доступные потоки, время выполнения остается почти постоянным. Первое измерение (1 итерация) занимает гораздо больше времени, что объясняется накладными расходами на запуск ядра CUDA, передачу данных в память GPU и синхронизацию потоков. После этого выполнение становится стабильным. Если задача эффективно распараллелена, GPU выполняет все вычисления за один или несколько циклов, и дальнейшее увеличение итераций практически не увеличивает время выполнения. Это объясняет, почему время стабилизируется.

На CPU время выполнения растет линейно с увеличением количества итераций. Это связано с тем, что CPU выполняет код последовательно (или с небольшой степенью параллелизма, если использует несколько потоков). В отличие от GPU, процессор не может запустить тысячи параллельных вычислений.