

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

## Итоговое задание

по дисциплине «Технологии параллельного программирования»

Проверил, доцент  
\_\_\_\_\_/Маковецкая Т. Ю.  
2024 г.

Автор работы  
студент группы КЭ-220  
\_\_\_\_\_/Голенищев А. Б.  
2024 г.

Работа защищена с оценкой  
(прописью, цифрой)

---

2024 г.

Челябинск  
2024

## Содержание

Теоретическое описание алгоритма БПФ .....	3
Практическая реализация .....	6
Синтетические данные.....	6
Данные из набора .....	7
Последовательные вычисления на процессоре .....	9
Использование FFTW .....	11
Собственная реализация алгоритма с параллельными вычислениями .....	12
Установка необходимых стандартов и библиотек.....	13
Настройка параметров сборки проекта.....	14
Основной код программы.....	15
Визуализация выходных данных.....	17
Оценка эффективности параллельного алгоритма .....	19
Расчет эффективности алгоритмов.....	21
Выводы .....	24
Библиографический список.....	25

## Теоретическое описание алгоритма БПФ

После второй мировой войны началась гонка вооружений, которая сопровождалась большим количеством испытаний ядерного оружия. После проведения испытаний водородных бомб, представители СССР и США - двух крупнейших ядерных держав заключили Договор о нераспространении ядерного оружия в 1968 г. [1], предполагающий запрет испытаний ядерного оружия под водой, в воздухе, на земле и в космосе. Данный договор не предусматривал подземные испытания ядерного оружия, т.к. существующие технические средства сейсмического контроля не позволяли отслеживать и контролировать такие испытания. Международной группе ученых, включая советских и американских, было поручено разработать техническое решение по мониторингу ядерных подземных испытаний в режиме реального времени. Группа математиков для анализа сигналов сейсмографов пробовала применить алгоритм дискретного преобразования Фурье. Исходные данные - временной ряд (сейсмограмма). Физический смысл разложения временного ряда сигнала в ряд Фурье - получении информации об амплитудном и частотном составе сигнала. Для преобразования в ряд Фурье мы умножаем сложную функцию сигнала на простые гармонические функции, затем интегрируем (считаем результирующую площадь под графиками результирующей функции), Рисунок 1. Так делаем для каждого значения умножаемой частоты.

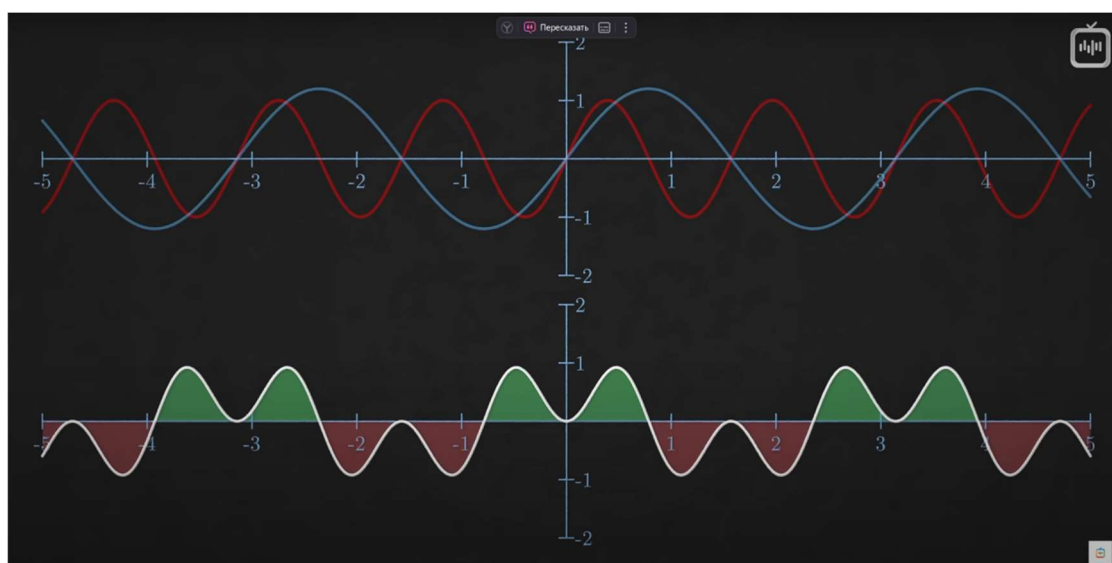


Рисунок 1. Умножение функции исследуемого сигнала на гармоническую

Реальные сигналы являются аналоговыми непрерывными величинами, поэтому мы можем обрабатывать с помощью вычислительных устройств только временные ряды. Для временных рядов применяется дискретное преобразование Фурье (ДПФ) и быстрое преобразование Фурье (БПФ). Преобразование Фурье отображает периодическую функцию  $f(t)$  с временной в частотную область в дискретной форме по следующей формуле (1). Дискретное преобразование Фурье предполагает умножение временного ряда из  $N$  значений на  $N$  гармонических функций, что требует высоких вычислительных затрат.

$$(1) F_k = \frac{1}{n} \sum_{j=0}^{n-1} f_j e^{-2\pi i \left(\frac{jk}{n}\right)}, k = 0, 1, \dots, n-1, f_k = f(x_k)$$

Американский математик Джон Кьюти, а также исследователь компании IBM Джеймс Кули в 1965 году опубликовали статью с решением проблемы производительности дискретного преобразования Фурье [2]. Быстрое преобразование Фурье является алгоритмом, полученным в ходе наблюдений Дж. Кьюти, который заметил, что значения периодических функций повторяются. Оно предполагает значительно меньшие вычислительные затраты, т.к. требуется перемножить не  $N^2$  гармонических функций, а  $N \log_2 N$ , где  $N$  - количество элементов выборки временного ряда. Это дает значительное преимущество, когда количество точек велико, Рисунок 2.

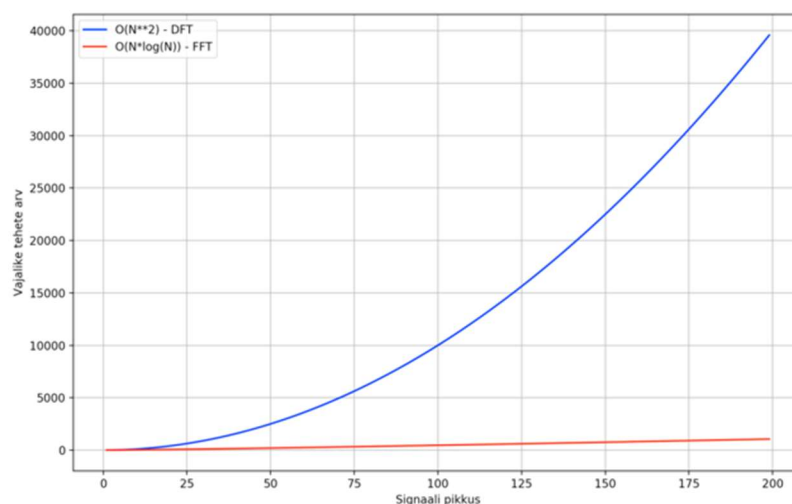


Рисунок 2. Количество вычислений от размера выборки

Для языка C++ существует хорошая реализация библиотеки для дискретного преобразования Фурье, которую разработали в 1999 г. исследователи из Массачусетского Технологического института [3]. Она носит название FFTW (Самое быстрое преобразование Фурье на Западе). Данная библиотека поддерживает MPI, а также OpenMP, Cilk, Rthreads. В данной работе будет показана реализация алгоритма БПФ с использованием OpenACC и OpenMP.

## Практическая реализация

В качестве набора данных для расчетов выбрали 2:

1. Синтетический набор данных (сами рассчитали выборку значений)
2. Взяли один файл `dart.csv` из набора данных [4], описание которого приведено в статье [5].

### Синтетические данные

```
using namespace std;

// Глобальные векторы для данных (std::vector)
vector<double> t, u, f, a;

// Искусственный сигнал
double my_signal(double t) {
    return 3 * cos(2 * M_PI * 3 * t + M_PI / 4) + 2 * sin(2 * M_PI * 7 * t - M_PI / 6) +
        1.5 * cos(2 * M_PI * 12 * t) + 0.8 * sin(2 * M_PI * 20 * t + M_PI / 3);
}

// Функция для генерации сигнала
void sample_signal(double (*func)(double), int m, vector<double> &x, vector<double> &y)
{
    x.clear();
    y.clear();

    double dt = 1.0 / m;

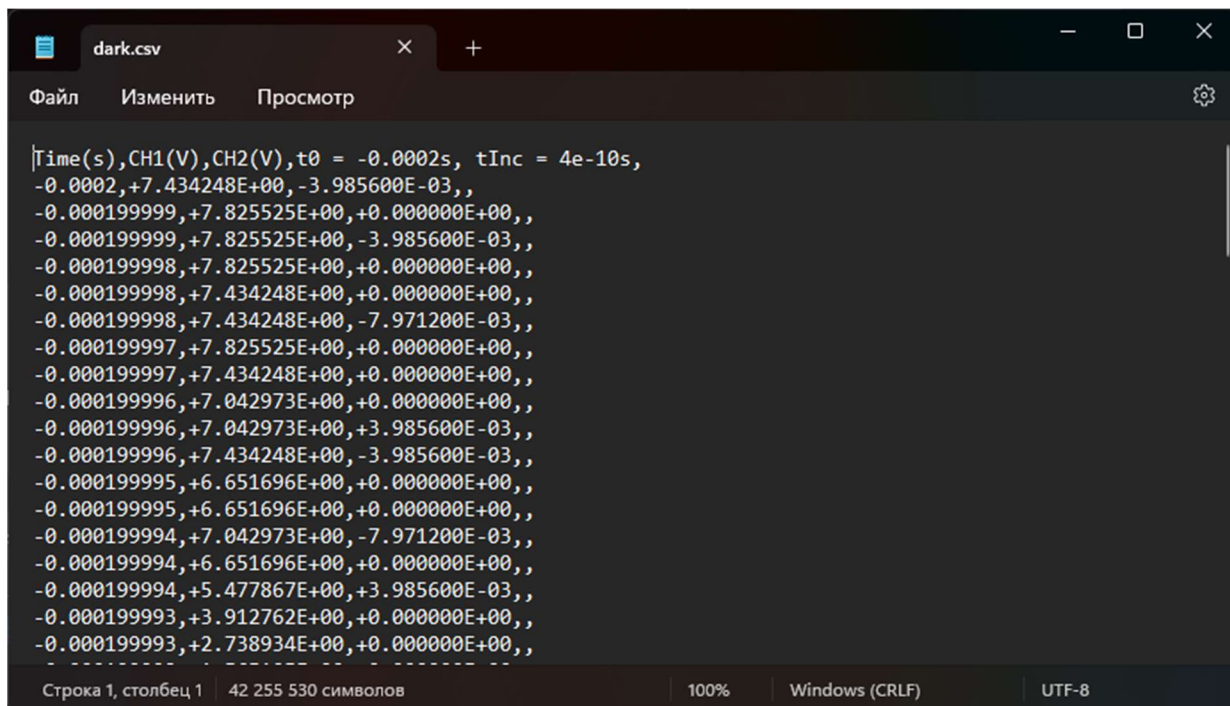
    for (int i = 0; i < m; ++i) {
        double time = i * dt;
        x.push_back(time);
        y.push_back(func(time));
    }
}
```

Листнинг 1. Реализация функций для создания искусственного временного ряда

Файл содержит следующие данные: время измерения, напряжение канала 1, напряжение канала 2 осциллографа. Осциллограф имеет частоту дискретизации 1 ГВыб/с, количество измерений в файле – 1000000 значений для обоих каналов (все замеры за 1 с проведены). Для констант и тригонометрических функций использована библиотека `math.h`.

## Данные из набора

Рассмотрим набор данных в текстовом редакторе, Рисунок 3.



```
|Time(s),CH1(V),CH2(V),t0 = -0.0002s, tInc = 4e-10s,  
-0.0002,+7.434248E+00,-3.985600E-03,,  
-0.000199999,+7.825525E+00,+0.000000E+00,,  
-0.000199999,+7.825525E+00,-3.985600E-03,,  
-0.000199998,+7.825525E+00,+0.000000E+00,,  
-0.000199998,+7.434248E+00,+0.000000E+00,,  
-0.000199998,+7.434248E+00,-7.971200E-03,,  
-0.000199997,+7.825525E+00,+0.000000E+00,,  
-0.000199997,+7.434248E+00,+0.000000E+00,,  
-0.000199996,+7.042973E+00,+0.000000E+00,,  
-0.000199996,+7.042973E+00,+3.985600E-03,,  
-0.000199996,+7.434248E+00,-3.985600E-03,,  
-0.000199995,+6.651696E+00,+0.000000E+00,,  
-0.000199995,+6.651696E+00,+0.000000E+00,,  
-0.000199994,+7.042973E+00,-7.971200E-03,,  
-0.000199994,+6.651696E+00,+0.000000E+00,,  
-0.000199994,+5.477867E+00,+3.985600E-03,,  
-0.000199993,+3.912762E+00,+0.000000E+00,,  
-0.000199993,+2.738934E+00,+0.000000E+00,,  
-----  
Строка 1, столбец 1 | 42 255 530 символов | 100% | Windows (CRLF) | UTF-8
```

Рисунок 3. Содержание файла dark.csv

Реализуем функцию чтения файла с указанием количества строк, листнинг 2. Данная функция работает с файлом благодаря библиотеке fstream, при чтении функция открывает файл с проверкой его открытия без ошибок (на случай, если путь к файлу указан неверно, например). Удаляет все значения из глобальных векторов, затем построчно считывает и заполняет их, переводя string в double. Чтение данных осуществляется пока line\_count не достигнет max\_lines.

```

bool read_csv(const string &filename, vector<double> &x, vector<double> &y, int
max_lines = -1)
{
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Ошибка: не удалось открыть файл " << filename << endl;
        return false;
    }
    x.clear();
    y.clear();
    string line;
    bool header_skipped = false;
    int line_count = 0; // Счётчик обработанных строк
    while (getline(file, line)) {
        if (!header_skipped) {
            header_skipped = true;
            continue; // Пропускаем первую строку (заголовок)
        }
        if (max_lines != -1 && line_count >= max_lines) {
            break; // Прекращаем чтение, если достигнут лимит строк
        }
        stringstream ss(line);
        string time_str, ch1_str;
        if (getline(ss, time_str, ',') && getline(ss, ch1_str, ',')) {
            try {
                x.push_back(stod(time_str));
                y.push_back(stod(ch1_str));
            } catch (const invalid_argument &e) {
                cerr << "Ошибка: некорректные данные в строке: " << line << endl;
                continue;
            }
        }
        ++line_count; // Увеличиваем счётчик строк
    }
    file.close();
    return true;
}

```

Листинг 2. Функция чтения данных из набора



## Последовательные вычисления на процессоре

Показана реализация функции для вычисления ДПФ без параллельных стандартов, листинг 3. Данная функция обрабатывает указанные ей векторы (в аргумент передаем глобальные  $u$  – с заполненными данными,  $f$ ,  $a$  – с пустыми, а также шаг дискретизации в секундах). Для вычисления комплексных значений используем тип `std::complex`. Представлена формула вычисления на каждом шаге (2).

$$(2) X_k = \sum_{n=0}^{N-1} u[n] \cdot e^{-\frac{i2\pi kn}{N}}$$

В формуле (2)  $u[n]$  – значение входного сигнала из временного ряда осциллограммы,  $k$  – индекс вычисляемой частоты,  $e^{-\frac{i2\pi kn}{N}}$  – комплексный коэффициент вращения (По формуле Эйлера  $e^{-\frac{i2\pi kn}{N}} = \cos\left(-\frac{2\pi kn}{N}\right) + i\sin\left(-\frac{2\pi kn}{N}\right)$ ),  $i$  – мнимая единица. В цикле для каждого значения  $u[n]$  вычислим угол фазу сигнала  $\text{angle}$ , как аргумент тригонометрических функций. Каждое  $u[n]$  умножаем на комплексное выражение, затем складываем частичные суммы («интегрируем») – находим значение амплитуды для данной частоты. Шаг частоты сигнала вычислется по формуле (3), где  $N$  – количество шагов во временном интервале,  $\Delta t$  – шаг по времени, его можно посмотреть в файле с данными или вычислить из частоты дискретизации осциллографа как обратную ей величину. Все вычисленные значения добавляем в конец векторов (динамических массивов).

$$(3) \Delta f = \frac{1}{N \cdot \Delta t}$$

```

void serial_compute_fft(const vector<double> &u, vector<double> &frequencies,
vector<double> &amplitudes, double delta_t) {
    int N = u.size();
    vector<complex<double>> fft_result(N);

    for (int k = 0; k < N; ++k) {
        complex<double> sum(0.0, 0.0);
        for (int n = 0; n < N; ++n) {
            double angle = -2.0 * M_PI * k * n / N;
            sum += u[n] * complex<double>(cos(angle), sin(angle));
        }
        fft_result[k] = sum;
    }

    double freq_step = 1.0 / (N * delta_t);

    for (int k = 0; k < N; ++k) {
        frequencies.push_back(k * freq_step);
        amplitudes.push_back(abs(fft_result[k]) / N);
    }
}

```

Листнинг 3. Реализация последовательного алгоритма ДПФ

## Использование FFTW

Напишем функцию с теми же входными данными, что и последовательная. Нужно подготовить динамические массивы сперва для использования с этой библиотекой – используются собственные классы и функции этой библиотеки. Запуск вычислений происходит с помощью функции `fftw_execute()`, аргументов которой служит `fftw_plan` – поля объекта являются параметрами для расчетов. В данном случае мы указываем конструктор для одномерного ДПФ (задаем в нем количество наших точек, входной и выходной массивы, тип преобразования (прямое/обратное) и флаг оптимизации (`FFTW_ESTIMATE` – план создается быстро без оптимизации)).

```
void fftw_compute_fft(const vector<double> &u, vector<double> &frequencies,
vector<double> &amplitudes, double delta_t) {
    int N = u.size();

    // Выделяем память для FFTW
    fftw_complex *in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    fftw_complex *out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
    fftw_plan plan = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);

    // Инициализируем входные данные
    for (int i = 0; i < N; ++i) {
        in[i][0] = u[i]; // Реальная часть
        in[i][1] = 0.0;  // Мнимая часть
    }

    // Выполняем FFT
    fftw_execute(plan);

    double freq_step = 1.0 / (N * delta_t);

    for (int k = 0; k < N; ++k) {
        frequencies.push_back(k * freq_step); // Частота
        amplitudes.push_back(sqrt(out[k][0] * out[k][0] + out[k][1] * out[k][1]) / N);
    }

    // Освобождаем ресурсы FFTW
    fftw_destroy_plan(plan);
    fftw_free(in);
    fftw_free(out);
}
```

Листинг 4. Функция, которая использует реализацию вычислений БПФ в FFTW

## Собственная реализация алгоритма с параллельными вычислениями

Идея распараллеливания в собственной реализации состоит в следующем: нам необходимо использовать те инструменты, которые предполагают использование CPU, поскольку во встраиваемых системах и многих ПК отсутствует дискретная видеокарта. По этой причине решено использовать два стандарта распараллеливания программ – OpenACC и OpenMP. Представлен код, листинг 5.

```
void my_compute_fft(const vector<double> &u, vector<double> &frequencies,
vector<double> &amplitudes, double time_step) {
    int N = u.size();
    vector<complex<double>> fft_result(N);

#pragma acc data copyin(u[0:N]) copyout(fft_result[0:N])
#pragma omp parallel for
    for (int k = 0; k < N; ++k) {
        complex<double> sum(0.0, 0.0);

        // Параллельная обработка по n с использованием OpenACC
#pragma acc parallel loop reduction(+:sum)
        for (int n = 0; n < N; ++n) {
            double angle = -2.0 * M_PI * k * n / N;
            sum += u[n] * complex<double>(cos(angle), sin(angle));
        }

        fft_result[k] = sum;
    }

    // Рассчитываем частоты и амплитуды
    double freq_step = 1.0 / (N * time_step);
    for (int k = 0; k < N; ++k) {
        frequencies.push_back(k * freq_step);
        amplitudes.push_back(abs(fft_result[k]) / N);
    }
}
```

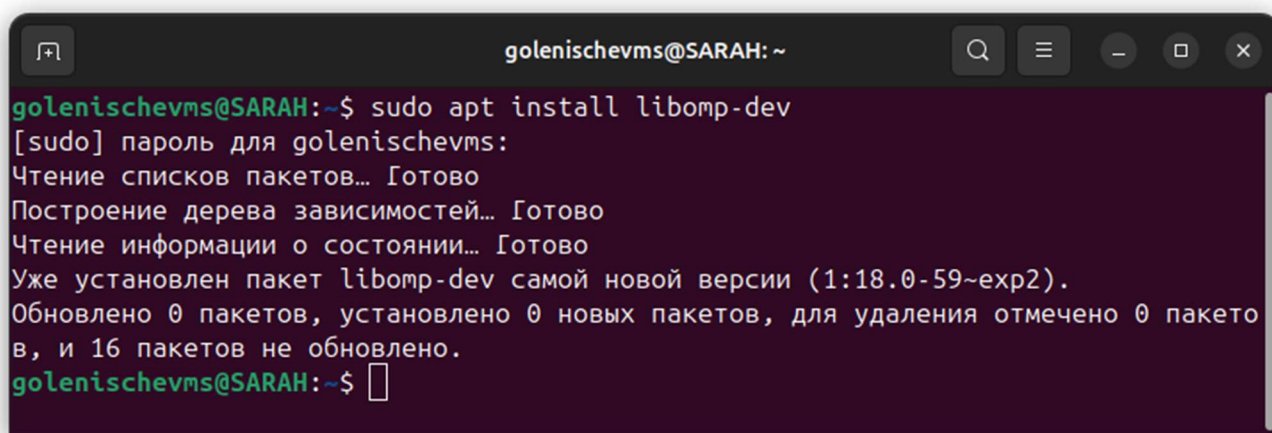
Листинг 5. Код собственной распараллеливающей функции для вычисления ДПФ

В моей реализации OpenMP используется для распараллеливания цикла вычислений на процессоре – задействуя по умолчанию максимальное число его потоков (АЛУ процессора выполняет математические операции быстро – поэтому для распараллеливания цикла является оптимальным решением). OpenACC предполагает, что данные могут быть загружены в видеокарту, количество процессоров в которой

значительно больше. Вычислить сумму сразу проще. Мы загружаем исходные данные в ускорительное устройство (видеокарту, которую использует OpenACC). Затем мы с потоках OpenMP распараллеливаем вычисления внутреннего цикла с помощью OpenACC. Таким образом – основные вычисления выборки мы выполняем на графическом ускорителе NVIDIA, а запись обработки рассчитанных значений (суммы) сохраняют потоки CPU.

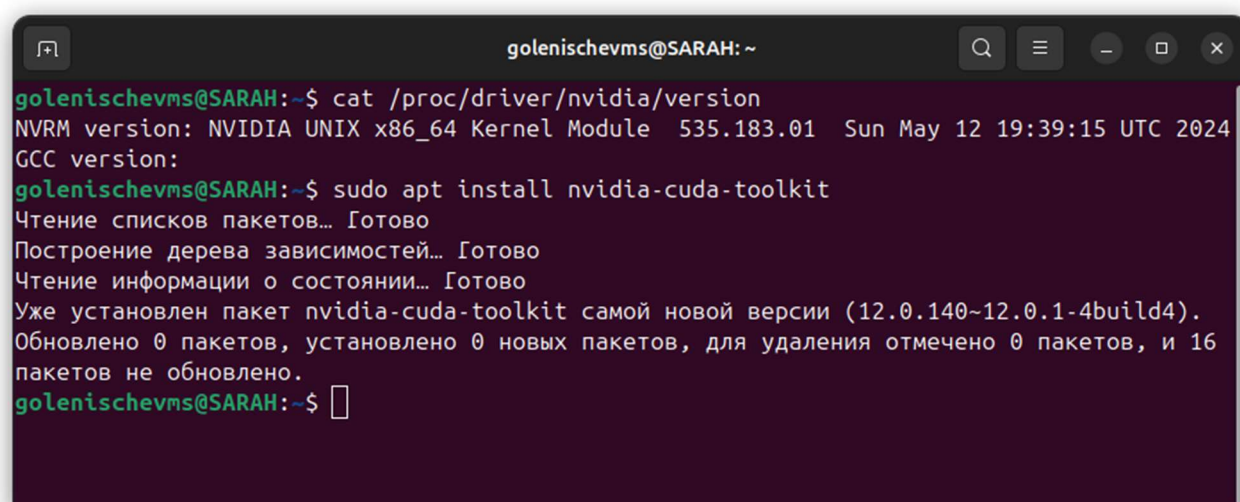
## Установка необходимых стандартов и библиотек

Установим OpenMP в Ubuntu, Рисунок 4. Установим также OpenACC (входит в Cuda Toolkit), Рисунок 5. Установим библиотеку FFTW, Рисунок 6.



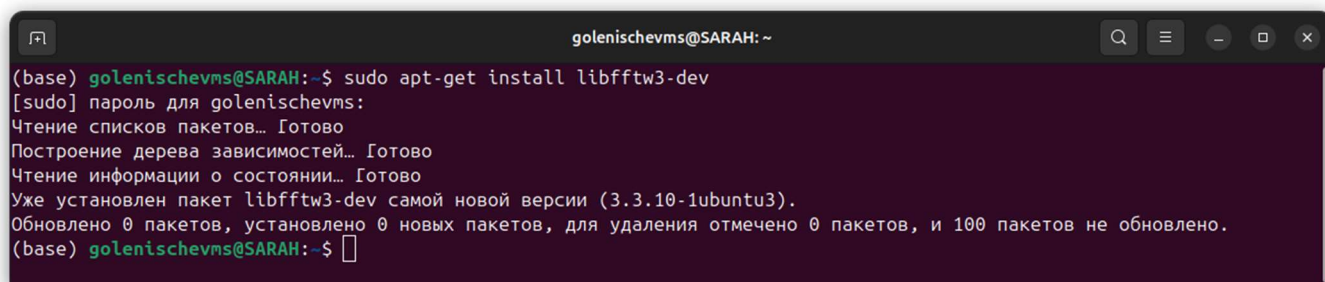
```
golenischevms@SARAH: ~  
golenischevms@SARAH:~$ sudo apt install libomp-dev  
[sudo] пароль для golenischevms:  
Чтение списков пакетов... Готово  
Построение дерева зависимостей... Готово  
Чтение информации о состоянии... Готово  
Уже установлен пакет libomp-dev самой новой версии (1:18.0-59~exp2).  
Обновлено 0 пакетов, установлено 0 новых пакетов, для удаления отмечено 0 пакетов,  
и 16 пакетов не обновлено.  
golenischevms@SARAH:~$
```

Рисунок 4. Установка OpenMP с помощью apt



```
golenischevms@SARAH: ~  
golenischevms@SARAH:~$ cat /proc/driver/nvidia/version  
NVRM version: NVIDIA UNIX x86_64 Kernel Module  535.183.01  Sun May 12 19:39:15 UTC 2024  
GCC version:  
golenischevms@SARAH:~$ sudo apt install nvidia-cuda-toolkit  
Чтение списков пакетов... Готово  
Построение дерева зависимостей... Готово  
Чтение информации о состоянии... Готово  
Уже установлен пакет nvidia-cuda-toolkit самой новой версии (12.0.140~12.0.1-4build4).  
Обновлено 0 пакетов, установлено 0 новых пакетов, для удаления отмечено 0 пакетов,  
и 16 пакетов не обновлено.  
golenischevms@SARAH:~$
```

Рисунок 5. Установка инструментов для вычислений на GPU



```
golenischevms@SARAH: ~  
(base) golenischevms@SARAH:~$ sudo apt-get install libfftw3-dev  
[sudo] пароль для golenischevms:  
Чтение списков пакетов... Готово  
Построение дерева зависимостей... Готово  
Чтение информации о состоянии... Готово  
Уже установлен пакет libfftw3-dev самой новой версии (3.3.10-1ubuntu3).  
Обновлено 0 пакетов, установлено 0 новых пакетов, для удаления отмечено 0 пакетов, и 100 пакетов не обновлено.  
(base) golenischevms@SARAH:~$
```

Рисунок 6. Установка FFTW с помощью apt

## Настройка параметров сборки проекта

Для запуска проекта с OpenMP и OpenACC необходимо добавить флаги компилятора в \*.pro файл настроек сборки проекта qmake (Qt), а также добавить пути к библиотекам компилятора с поддержкой OpenACC, флаги библиотеки, листнинг 6.

```
TEMPLATE = app  
CONFIG += console c++17  
CONFIG -= app_bundle  
CONFIG -= qt  
  
# Флаги компилятора и линковки для OpenMP  
QMAKE_CXXFLAGS += -fopenmp  
QMAKE_LFLAGS += -fopenmp  
  
# Пути к заголовкам и библиотекам OpenACC  
INCLUDEPATH += /snap/freecad/1202/usr/lib/gcc/x86_64-linux-gnu/11/include  
LIBS += -L/snap/freecad/1202/usr/lib/gcc/x86_64-linux-gnu/11/lib64  
  
# Пути к заголовкам и библиотекам FFTW  
INCLUDEPATH += /path/to/fftw/include  
LIBS += -L/path/to/fftw/lib -lfftw3 -lm  
  
# Источник кода  
SOURCES += \  
    main.cpp
```

Листнинг 6. Содержимое \*.pro файла проекта с OpenMP и OpenACC в QT

## Основной код программы

В основном коде программы мы реализовали как возможность использования как синтетических данных, так и выборки из файла с фиксированным количеством значений. Выборка обрабатывается тремя исследуемыми методами, затем результаты записываются в CSV файлы, для дальнейшей обработки/визуализации данных. Подсчитываем временные затраты на выполнения каждой функции вычислений ДПФ.

Показан пример вывода программы, Рисунок 7. Представлен код программы, листинг 7.

```
CalculationFFT_Golenishchev_KE220 X
17:33:38: Starting /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220...
Загружаю данные из файла
Данные загрузил
Размер вектора данных (количество значений напряжения и времени): 552
Вычисляю последовательно БПФ
Время serial_compute_fft: 5 мс
Вычисляю БПФ с помощью OpenMP и OpenACC
Время my_compute_fft: 17 мс
Вычисляю БПФ с помощью средств библиотеки FFTW
Время fftw_compute_fft: 0 мс
Готово.
17:33:38: /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220 exited with code 0
```

Рисунок 7. Вывод программы с фиксацией временных затрат на вычисление ДПФ  
каждым исследуемым методом

Проведены несколько тестов с различным количеством значений временного ряда от 500 до 500 000 точек (в файле 1 000 000 точек), Рисунок 8.

```
main.cpp @ CalculationFFT_Golenishchev_KE220 - Qt Creator
101 y.clear();
102
103 double dt = 1.0 / m;
104
105 for (int i = 0; i < m; ++i) {
106     double time = i * dt;
107     x.push_back(time);
108     y.push_back(func(time));
109 }
110
111

Вывод приложения
CalculationFFT_Golenishchev_KE220 X
08:56:03: Starting /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220...
Загружаю данные из файла
Данные загрузил
Размер вектора данных (количество значений напряжения и времени): 500
Вычисляю последовательно БПФ
Время serial_compute_fft: 5181 мс
Вычисляю БПФ с помощью OpenMP и OpenACC
Время my_compute_fft: 1070 мс
Вычисляю БПФ с помощью средств библиотеки FFTW
Время fftw_compute_fft: 218 мс
Готово.
08:56:03: /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220 exited with code 0
08:56:10: Starting /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220...
Загружаю данные из файла
Данные загрузил
Размер вектора данных (количество значений напряжения и времени): 1000
Вычисляю последовательно БПФ
Время serial_compute_fft: 19031 мс
Вычисляю БПФ с помощью OpenMP и OpenACC
Время my_compute_fft: 2518 мс
Вычисляю БПФ с помощью средств библиотеки FFTW
Время fftw_compute_fft: 301 мс
Готово.
08:56:10: /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220 exited with code 0
08:56:16: Starting /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220...
Загружаю данные из файла
Данные загрузил
Размер вектора данных (количество значений напряжения и времени): 5000
Вычисляю последовательно БПФ
Время serial_compute_fft: 470399 мс
Вычисляю БПФ с помощью OpenMP и OpenACC
Время my_compute_fft: 52481 мс
Вычисляю БПФ с помощью средств библиотеки FFTW
Время fftw_compute_fft: 572 мс
Готово.
08:56:17: /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220 exited with code 0
08:56:21: Starting /home/golenishevms/CalculationFFT_Golenishchev_KE220/build/MPI_Lab-Debug/CalculationFFT_Golenishchev_KE220...
Загружаю данные из файла
Данные загрузил
Размер вектора данных (количество значений напряжения и времени): 10000
```

Рисунок 8. Результаты тестовых запусков с временными рядами различной длины

```

int main() {
    // Тестирование на синтетических данных
    // cout << "Генерирую данные" << endl;
    // const int m = 1000;
    // sample_signal(my_signal, m, t, u);
    // Тестирование на данных из файла
    cout << "Загружаю данные из файла" << endl;
    read_csv("/home/golenischevms/CalculationFFT_Golenishchev_KE220/input_data/dark.
csv", t, u, 500000);
    cout << "Данные загрузил" << endl;
    double time_step = 4e-10;
    vector<double> f_serial, a_serial;
    vector<double> f_my, a_my;
    vector<double> f_fftw, a_fftw;
    cout << "Размер вектора данных (количество значений напряжения и времени): " <<
u.size() << endl;
    cout << "Вычисляю последовательно БПФ" << endl;
    auto start = chrono::high_resolution_clock::now();
    serial_compute_fft(u, f_serial, a_serial, time_step);
    auto end = chrono::high_resolution_clock::now();
    cout << "Время serial_compute_fft: " <<
chrono::duration_cast<chrono::microseconds>(end - start).count() << " мкс" << endl;
    write_csv("/home/golenischevms/CalculationFFT_Golenishchev_KE220/output_data/ser
ial_results.csv", t, u, f_serial, a_serial);
    cout << "Вычисляю БПФ с помощью OpenMP и OpenACC" << endl;
    start = chrono::high_resolution_clock::now();
    my_compute_fft(u, f_my, a_my, time_step);
    end = chrono::high_resolution_clock::now();
    cout << "Время my_compute_fft: " <<
chrono::duration_cast<chrono::microseconds>(end - start).count() << " мкс" << endl;
    write_csv("/home/golenischevms/CalculationFFT_Golenishchev_KE220/output_data/my_
results.csv", t, u, f_my, a_my);
    cout << "Вычисляю БПФ с помощью средств библиотеки FFTW" << endl;
    start = chrono::high_resolution_clock::now();
    fftw_compute_fft(u, f_fftw, a_fftw, time_step);
    end = chrono::high_resolution_clock::now();
    cout << "Время fftw_compute_fft: " <<
chrono::duration_cast<chrono::microseconds>(end - start).count() << " мкс" << endl;
    write_csv("/home/golenischevms/CalculationFFT_Golenishchev_KE220/output_data/fft
w_results.csv", t, u, f_fftw, a_fftw);

    cout << "Готово." << endl;
    return 0;
}

```

Листинг 7. Основной код программы



## Визуализация выходных данных

На C++ разработано приложение QtWidgets, которое позволяет обработанные выборки визуализировать с помощью библиотеки QCustomPlot, поддерживающей OpenGL, Рисунки 9-11.



Рисунок 9. Визуализация результатов обработки OpenACC + OpenMP

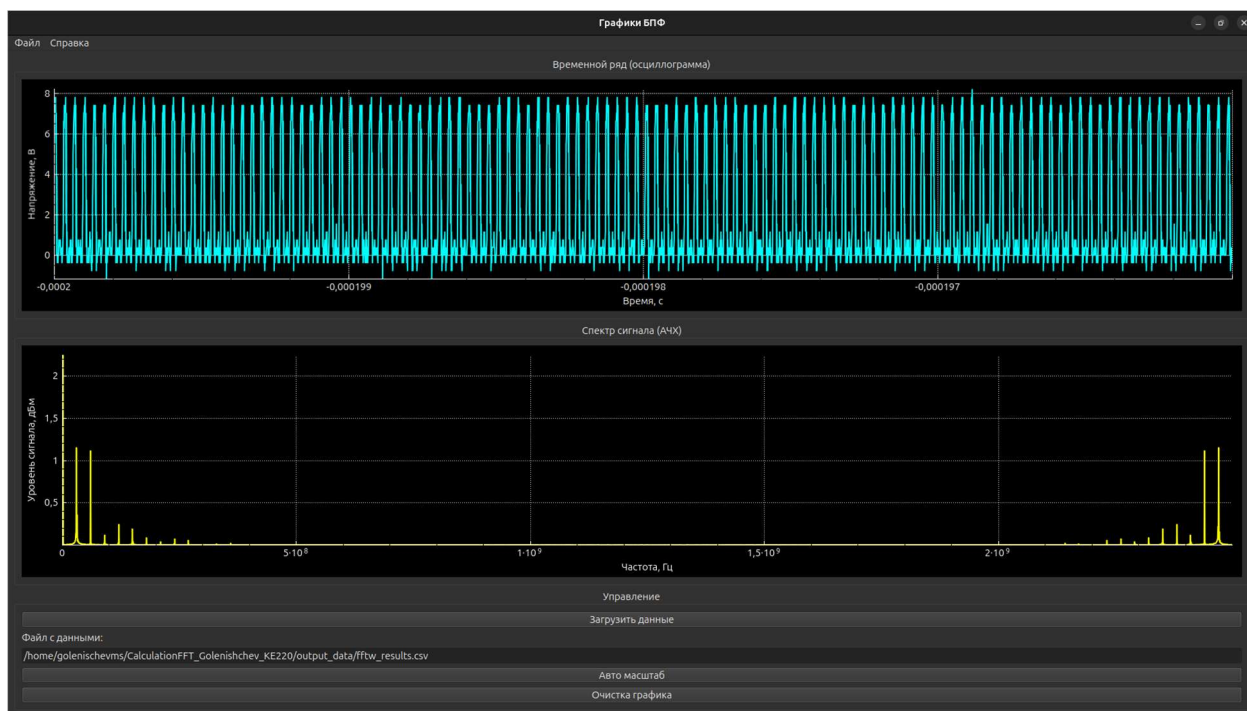


Рисунок 10. Визуализация результатов обработки FFTW

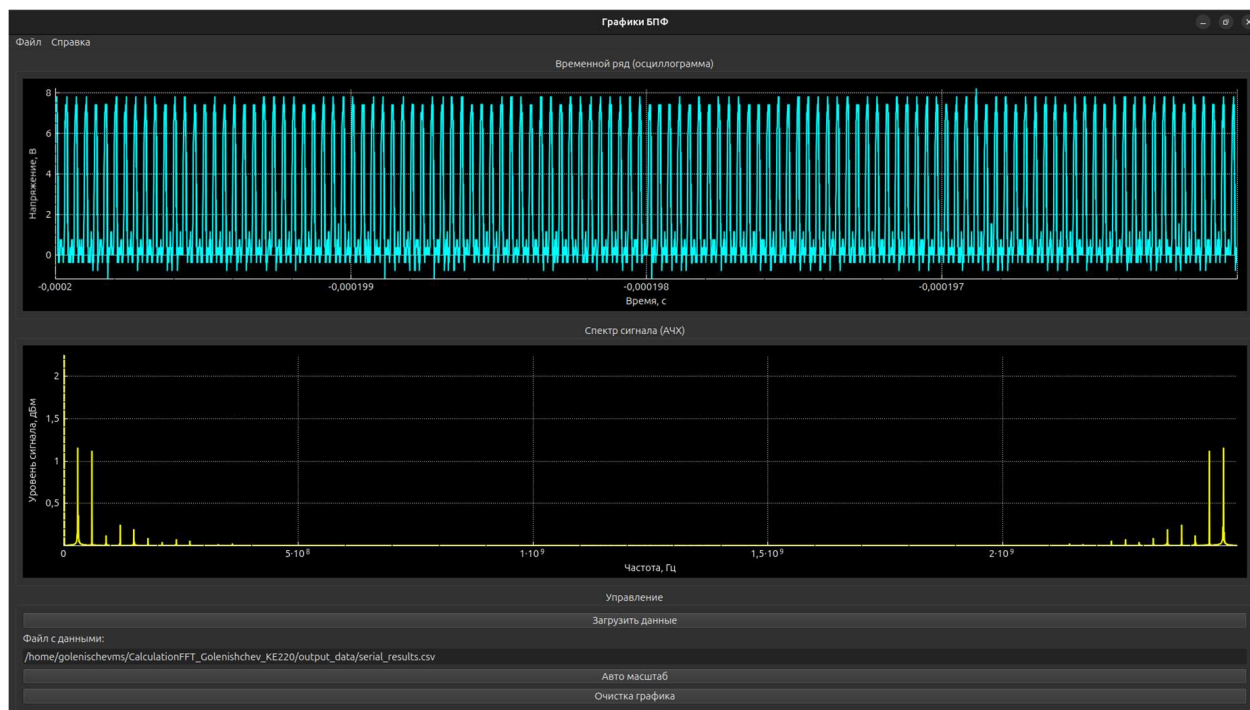


Рисунок 11. Визуализация результатов обработки последовательных вычислений

Построенные графики позволяют оценить результаты работы алгоритмов.

## Оценка эффективности параллельного алгоритма

Представлена таблица, состоящая из рассчитанных значений временных затрат в ходе вычислений тремя методами, Таблица 1.

Таблица 1 – Время расчета БПФ (в мкс) исследуемыми стандартами

Количество выборок	Последовательные вычисления на CPU	OpenACC + OpenMP	FFTW
500	5181	1070	276
1000	19931	2518	301
5000	470399	52401	572
10000	1862332	196433	725
50000	46004596	3789500	2506
100000	187269098	16958167	4788
500000	7473433645	649621625	22190

Визуализируем результаты с помощью matplotlib, Рисунок 12.

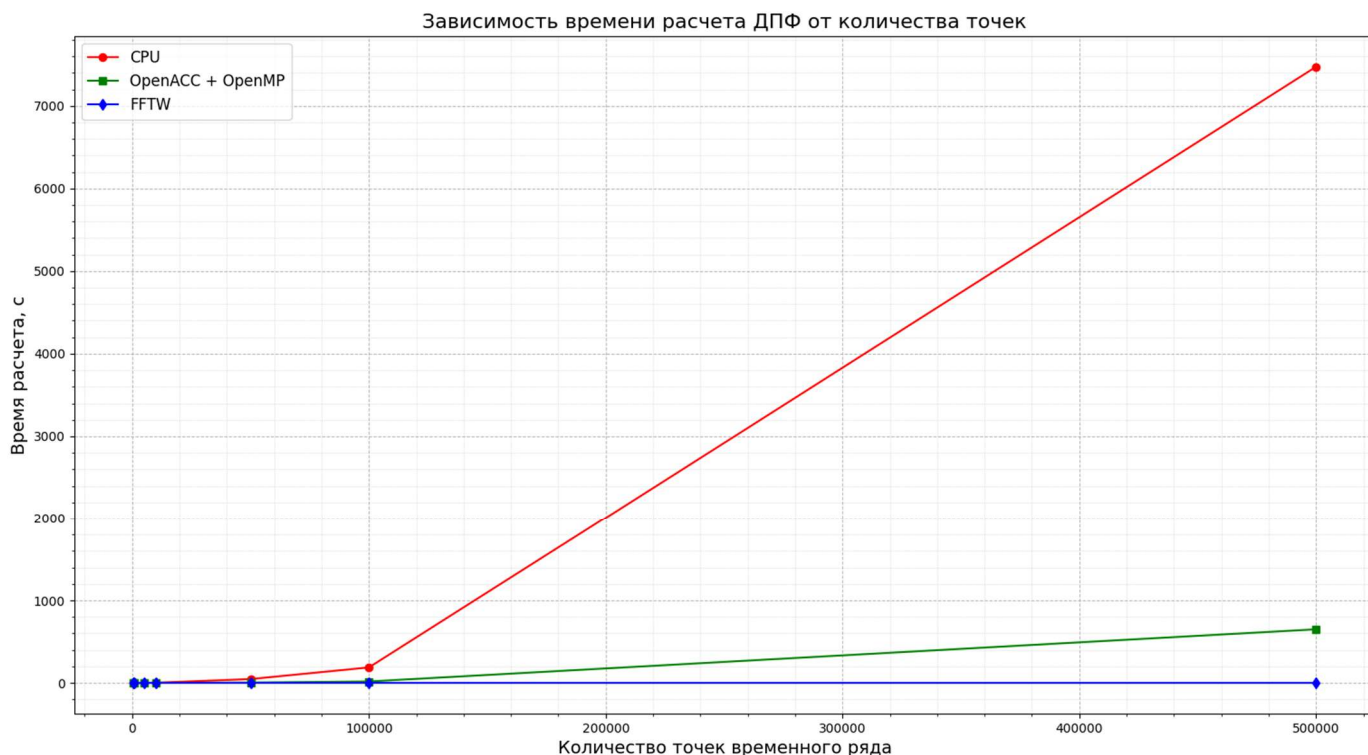


Рисунок 12. График затраченного времени при вычислениях

Представлен код программы для построения графика сравнения результатов временных затрат, листнинг 8.

```
import matplotlib.pyplot as plt
import numpy as np
# Данные
samples = np.array([500, 1000, 5000, 10000, 50000, 100000, 500000])
cpu_times = np.array([5181, 19931, 470399, 1862332, 46004596, 187269098,
7473433645]) / 1000000 # Перевод в секунды
openacc_times = np.array([1070, 2518, 52401, 196433, 3789500, 16958167, 649621625])
/ 1000000
fftw_times = np.array([276, 301, 572, 725, 2506, 4788, 22190]) / 1000000
# Построение графика
plt.plot(samples, cpu_times, 'o-r', label='CPU') # Красный кружок
plt.plot(samples, openacc_times, 's-g', label='OpenACC + OpenMP') # Зеленый квадрат
plt.plot(samples, fftw_times, 'd-b', label='FFTW') # Синий ромб
# Настройка осей
plt.xlabel('Количество точек временного ряда', fontsize=14)
plt.ylabel('Время расчета, с', fontsize=14)
plt.title('Зависимость времени расчета ДПФ от количества точек', fontsize=16)
# Сетка
plt.grid(which='major', linestyle='--', linewidth=0.7, alpha=0.9) # Основная сетка
plt.grid(which='minor', linestyle=':', linewidth=0.5, alpha=0.7) # Дополнительная
сетка
plt.minorticks_on() # Включить дополнительные деления
# Легенда
plt.legend(fontsize=12, loc='upper left')
# Показ графика
plt.show()
```

Листнинг 8. Код программы для построения графика на Python

Представлены характеристики компьютера, на котором проводились расчеты, Рисунок 13. Процессор – Intel Core i7-13700K (16 ядер 24 потока с тактовой частотой 3,4 ГГц и до 5,4 ГГц в режиме турбо), видеокарта NVIDIA GEFORCE RTX4060TI (2580 МГц, DLSS 3, шейдерные ядра Ada Lovelace производительностью 22 TFLOPS, ядра тарсировки лучей 3th Gen производительностью 51 TFLOPS, тензорные ядра 4th Gen производительностью 353 AI TOPS, 16 ГБ GDDR6 видеопамяти).

```

golenischevms@SARAH:~$ neofetch
      .-/+oosssso+/-.
      `:+ssssssssssssssssss+:`
    -+ssssssssssssssssssssyyssss+-
    .osssssssssssssssssssdMMMMyssssso.
    /ssssssssssshdmmNNmmyNMMMMhssssss/
    +ssssssssshmydMMMMMMMNddddyssssssss+
    /ssssssssshNMMMMyhhyyyyhmNMMMNhssssssss/
    .ssssssssdMMMNhssssssssshNMMMdssssssss.
    +ssssshhhyNMMNyssssssssssssyNMMMyssssssss+
    ossyNMMMNyMMhssssssssssssshmmhssssssso
    ossyNMMMNyMMhssssssssssssshmmhssssssso
    +ssssshhhyNMMNyssssssssssssyNMMMyssssssss+
    .ssssssssdMMMNhssssssssshNMMMdssssssss.
    /ssssssssshNMMMMyhhyyyyhdNMMMNhssssssss/
    +ssssssssdnydMMMMMMMNddddyssssssss+
    /ssssssssshdmmNNNmyNMMMMhssssss/
    .osssssssssssssssssssdMMMMyssssso.
    -+ssssssssssssssssssssyyssss+-
    `:+ssssssssssssssssss+:`
      .-/+oosssso+/-.

golenischevms@SARAH
-----
OS: Ubuntu 24.04.1 LTS x86_64
Host: MS-7D32 3.0
Kernel: 6.8.0-49-generic
Uptime: 3 hours, 47 mins
Packages: 2310 (dpkg), 26 (flatpak),
Shell: bash 5.2.21
Resolution: 1920x1080
DE: GNOME 46.0
WM: Mutter
WM Theme: Adwaita
Theme: Yaru-blue-dark [GTK2/3]
Icons: Yaru-blue [GTK2/3]
Terminal: gnome-terminal
CPU: 13th Gen Intel i7-13700K (24) @
GPU: Intel Raptor Lake-S GT1 [UHD Gr
GPU: NVIDIA GeForce RTX 4060 Ti 16GB
Memory: 4968MiB / 64079MiB

golenischevms@SARAH:~$

```

Рисунок 13. Характеристики ПК для тестирования алгоритмов.

## Расчет эффективности алгоритмов

Проведем расчет эффективности алгоритма: вычислим значения ускорения по формуле (4), где  $S_p(n)$  – параметр стоимости,  $T_{seq}(n)$  – время выполнения последовательного алгоритма для  $n$  выборок,  $T_p(n)$  – время выполнения параллельного алгоритма для  $n$  выборок (OpenACC + OpenMP или FFTW).

$$(4) S_p(n) = \frac{T_{seq}(n)}{T_p(n)}$$

Проведем расчет стоимости по формуле (5),  $C_p(n)$  – стоимость параллельного алгоритма,  $T_p$  – время выполнения параллельного алгоритма для  $n$  выборок,  $p$  – число потоков/процессов (расчет проведем для 8 процессов).

$$(5) C_p(n) = T_p \cdot p$$

Расчеты проведем с помощью программы на языке Python, листнинг 9.

```
import numpy as np
from prettytable import PrettyTable
import matplotlib.pyplot as plt
# Данные
samples = np.array([500, 1000, 5000, 10000, 50000, 100000, 500000])
time_cpu = np.array([5181, 19931, 470399, 1862332, 46004596, 187269098, 7473433645])
time_openacc = np.array([1070, 2518, 52401, 196433, 3789500, 16958167, 649621625])
time_fftw = np.array([276, 301, 572, 725, 2506, 4788, 22190])
# Число процессоров
p_openacc = 8 # Количество процессов для OpenACC + OpenMP
p_fftw = 8 # Количество процессов для FFTW
# Ускорение (Speedup)
speedup_openacc = time_cpu / time_openacc
speedup_fftw = time_cpu / time_fftw
# Стоимость (Cost)
cost_openacc = time_openacc * p_openacc
cost_fftw = time_fftw * p_fftw
# Формирование таблицы результатов
results_table = PrettyTable()
results_table.field_names = [
    "Количество выборок", "Ускорение OpenACC + OpenMP", "Ускорение FFTW", "Стоимость
OpenACC + OpenMP", "Стоимость FFTW"
]
for i in range(len(samples)):
    results_table.add_row([
        samples[i],
        f"{speedup_openacc[i]:.2f}",
        f"{speedup_fftw[i]:.2f}",
        f"{cost_openacc[i]:.2e}",
        f"{cost_fftw[i]:.2e}"
    ])
print("Таблица 2 - Результаты расчетов эффективности алгоритма:")
print(results_table)
```

Листнинг 9. Код программы для расчета ускорения и стоимости

Приведены результаты расчетов, программы Таблица 2.



Таблица 2 – Результаты расчетов эффективности и стоимости алгоритмов

Количество выборок	Ускорение OpenACC + OpenMP	Ускорение FFTW	Стоимость OpenACC + OpenMP	Стоимость FFTW
500	4.84	18.77	8.56e+03	1.10e+03
1000	7.92	66.22	2.01e+04	1.20e+03
5000	8.98	822.38	4.19e+05	2.29e+03
10000	9.48	2568.73	1.57e+06	2.90e+03
50000	12.14	18357.78	3.03e+07	1.00e+04
100000	11.04	39112.18	1.36e+08	1.92e+04
500000	11.50	336792.86	9.02e+08	8.88e+04

Визуализируем результаты вычислений, Рисунки 14-15.

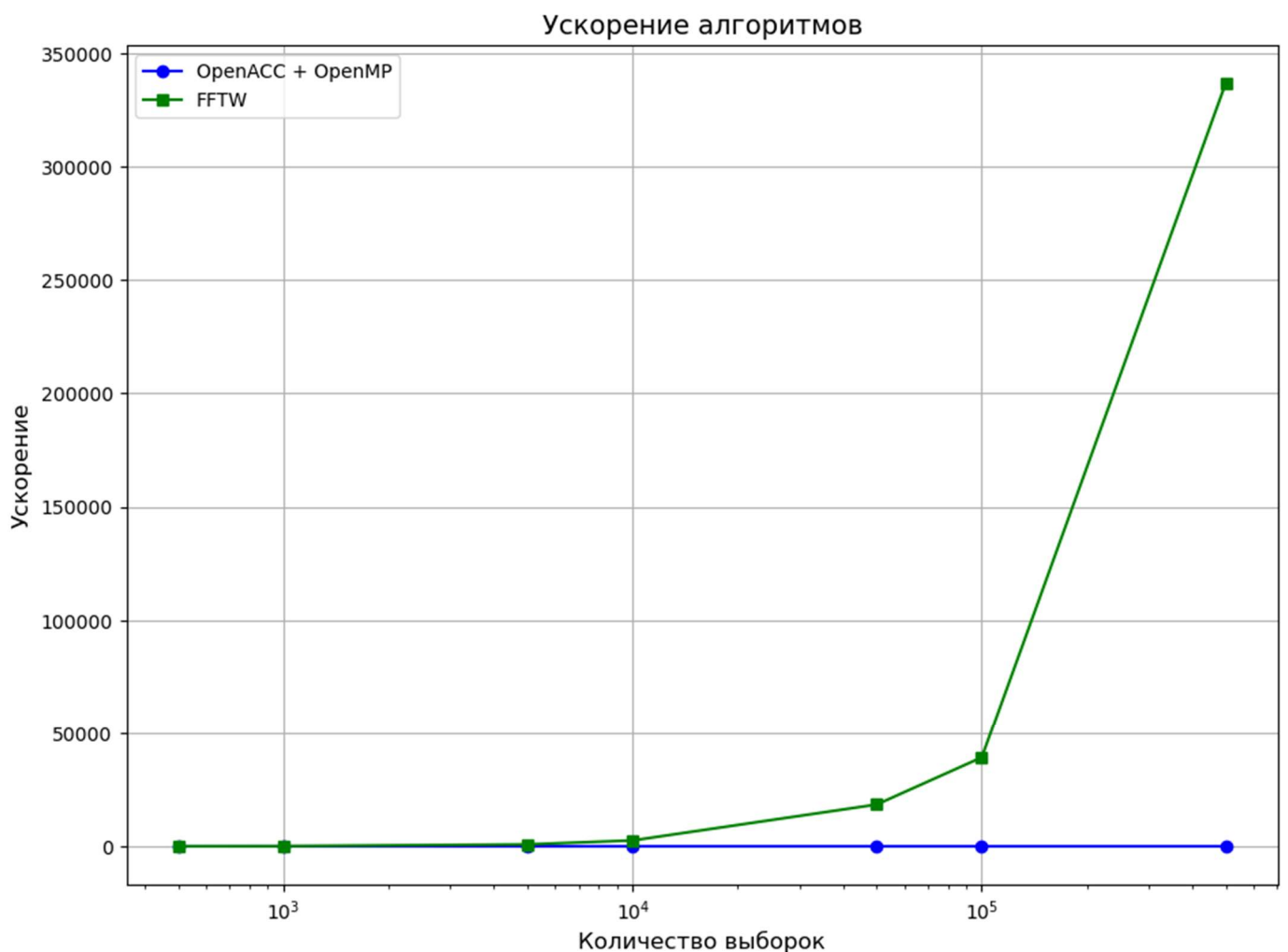


Рисунок 14. График сравнения ускорения алгоритмов для временного ряда различной длины

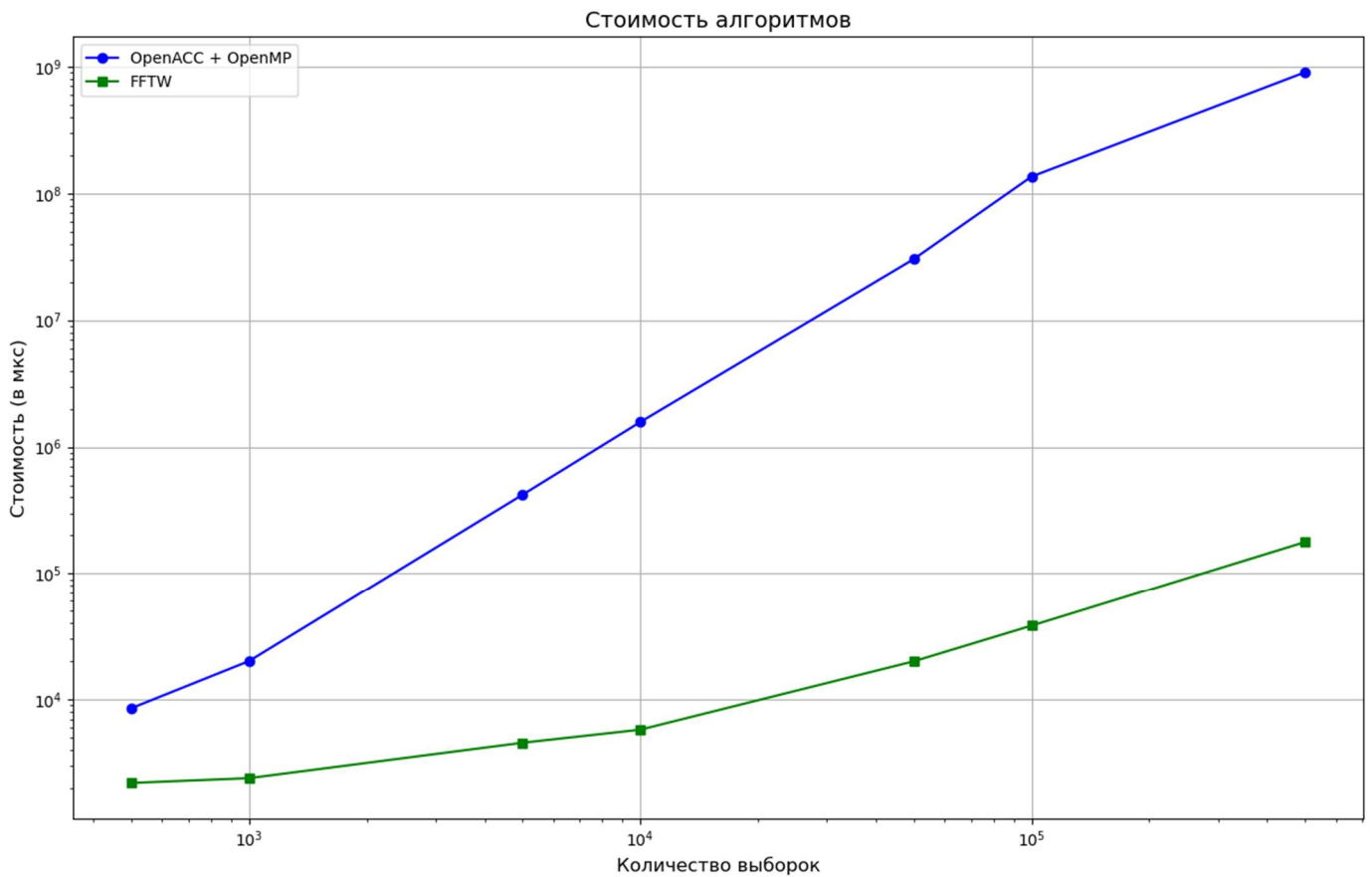


Рисунок 15. График сравнения стоимости алгоритмов для временного ряда различной длины

## Выводы

Расчет эффективности показал, что БПФ, реализованный в FFTW значительно превышает ускорение OpenACC + OpenMP для ДПФ на всех объемах данных, чем больше данных – тем больше разрыв.

Использование параллельных многопоточных вычислений на CPU и GPU позволило значительно ускорить вычисления.



## Библиографический список

1. Договор о нераспространении ядерного оружия [Электронный ресурс] URL: [https://ru.wikipedia.org/wiki/Договор\\_о\\_нераспространении\\_ядерного\\_оружия](https://ru.wikipedia.org/wiki/Договор_о_нераспространении_ядерного_оружия) (дата обращения 14.12.2024 г.)
2. Cooley J.W. and Tukey J.W. An algorithm for the machine calculation of the complex fourier series. Mathematics Computation, 19:297-301, 1965.
3. FFTW Home Page [Электронный ресурс] URL: <https://www.fftw.org/> (дата обращения 14.12.2024 г.)
4. Oscilloscope data for SPAD quenched by 100 kohm [Электронный ресурс] URL: [https://figshare.com/articles/dataset/Oscilloscope\\_data\\_for\\_SPAD\\_quenched\\_by\\_100\\_kohm/19092761/1](https://figshare.com/articles/dataset/Oscilloscope_data_for_SPAD_quenched_by_100_kohm/19092761/1) (дата обращения: 30.11.2024 г.).
5. Zheng J. et al. Dynamic-quenching of a single-photon avalanche photodetector using an adaptive resistive switch //Nature Communications. – 2022. – Т. 13. – №. 1. – С. 1517.