Федеральное государственное бюджетное образовательное учреждение «Сибирский государственный университет телекоммуникаций и информатики»

Кафедра ПМиК

Расчетно-графическое задание

по дисциплине

«Программирование графических процессоров»

Выполнил: студент III курса

ИВТ, гр. ИП-814

Краснов И.В.

Проверил:

Нужнов А.В.

Новосибирск 2021

**Оглавление**

[Задание 3](#_Toc72401696)

[Теория 4](#_Toc72401697)

[Сравнение производительности алгоритмов 7](#_Toc72401698)

[Вывод 8](#_Toc72401699)

[Приложения 9](#_Toc72401700)

[Код программы 10](#_Toc72401701)

# **Задание**

Сравнительный анализ производительности программ, реализующих алгоритм быстрого преобразования Фурье с использованием библиотек cuFFT, cuFFTW и FFTW3.

# **Теория**

Технология CUDA

Технология CUDA — это программно-аппаратная вычислительная архитектура Nvidia, основанная на расширении языка Си, которая даёт возможность организации доступа к набору инструкций графического ускорителя и управления его памятью при организации параллельных вычислений. CUDA помогает реализовывать алгоритмы, выполнимые на графических процессорах видеоускорителей Geforce восьмого поколения и старше (серии Geforce 8, Geforce 9, Geforce 200), а также Quadro и Tesla.

CUDA использует параллельную модель вычислений, когда каждый из SIMD процессоров выполняет ту же инструкцию над разными элементами данных параллельно. GPU является вычислительным устройством, сопроцессором (device) для центрального процессора (host), обладающим собственной памятью и обрабатывающим параллельно большое количество потоков. Ядром (kernel) называется функция для GPU, исполняемая потоками (аналогия из 3D графики — шейдер).

Модель программирования в CUDA предполагает группирование потоков. Потоки объединяются в блоки потоков (thread block) — одномерные или двумерные сетки потоков, взаимодействующих между собой при помощи разделяемой памяти и точек синхронизации. Программа (ядро, kernel) исполняется над сеткой (grid) блоков потоков (thread blocks). Одновременно исполняется одна сетка. Каждый блок может быть одно-, двух- или трехмерным по форме, и может состоять из 512 потоков на текущем аппаратном обеспечении.

Блоки потоков выполняются в виде небольших групп, называемых варп (warp), размер которых — 32 потока. Это минимальный объём данных, которые могут обрабатываться в мультипроцессорах. И так как это не всегда удобно, CUDA позволяет работать и с блоками, содержащими от 64 до 512 потоков.

Группировка блоков в сетки позволяет уйти от ограничений и применить ядро к большему числу потоков за один вызов. Это помогает и при масштабировании. Если у GPU недостаточно ресурсов, он будет выполнять блоки последовательно. В обратном случае, блоки могут выполняться параллельно, что важно для оптимального распределения работы на видеочипах разного уровня, начиная от мобильных и интегрированных.

Модель памяти в CUDA отличается возможностью побайтной адресации, поддержкой как gather, так и scatter. Доступно довольно большое количество регистров на каждый потоковый процессор, до 1024 штук. Доступ к ним очень быстрый, хранить в них можно 32-битные целые или числа с плавающей точкой.

СUDA имеет несколько типов памяти. Эта технология предполагает специальный подход к разработке, не совсем такой, как принят в программах для CPU. Нужно помнить о разных типах памяти, о том, что локальная и глобальная память не кэшируется и задержки при доступе к ней гораздо выше, чем у регистровой памяти, так как она физически находится в отдельных микросхемах.

Библиотеки cuFFT, cuFFTW и FFTW3

Библиотека cuFFT предоставляет реализации быстрого преобразования Фурье для GPU, которые работают в 10 раз быстрее, чем альтернативы только для процессора. cuFFT используется для создания коммерческих и исследовательских приложений в таких дисциплинах, как deep learning, computer vision, вычислительная физика, молекулярная динамика, квантовая химия, сейсмическая и медицинская визуализация. Используя cuFFT, приложения автоматически извлекают выгоду из регулярных улучшений производительности и новых архитектур GPU. Библиотека cuFFT включена как в NVIDIA HPC SDK, так и в инструментарий CUDA.

Библиотека *cuFTTW* является реализацией библиотеки *FFTW* для программ, исполняемых на видеокартах *NVidia*, поддерживающих технологию *CUDA*. Библиотека призвана стать самой быстрой библиотекой для преобразований Фурье. FFTW3 является одной из версий данной библиотеки. На 12.05.2021 последней является версия 3.3.9.

# **Сравнение производительности алгоритмов**

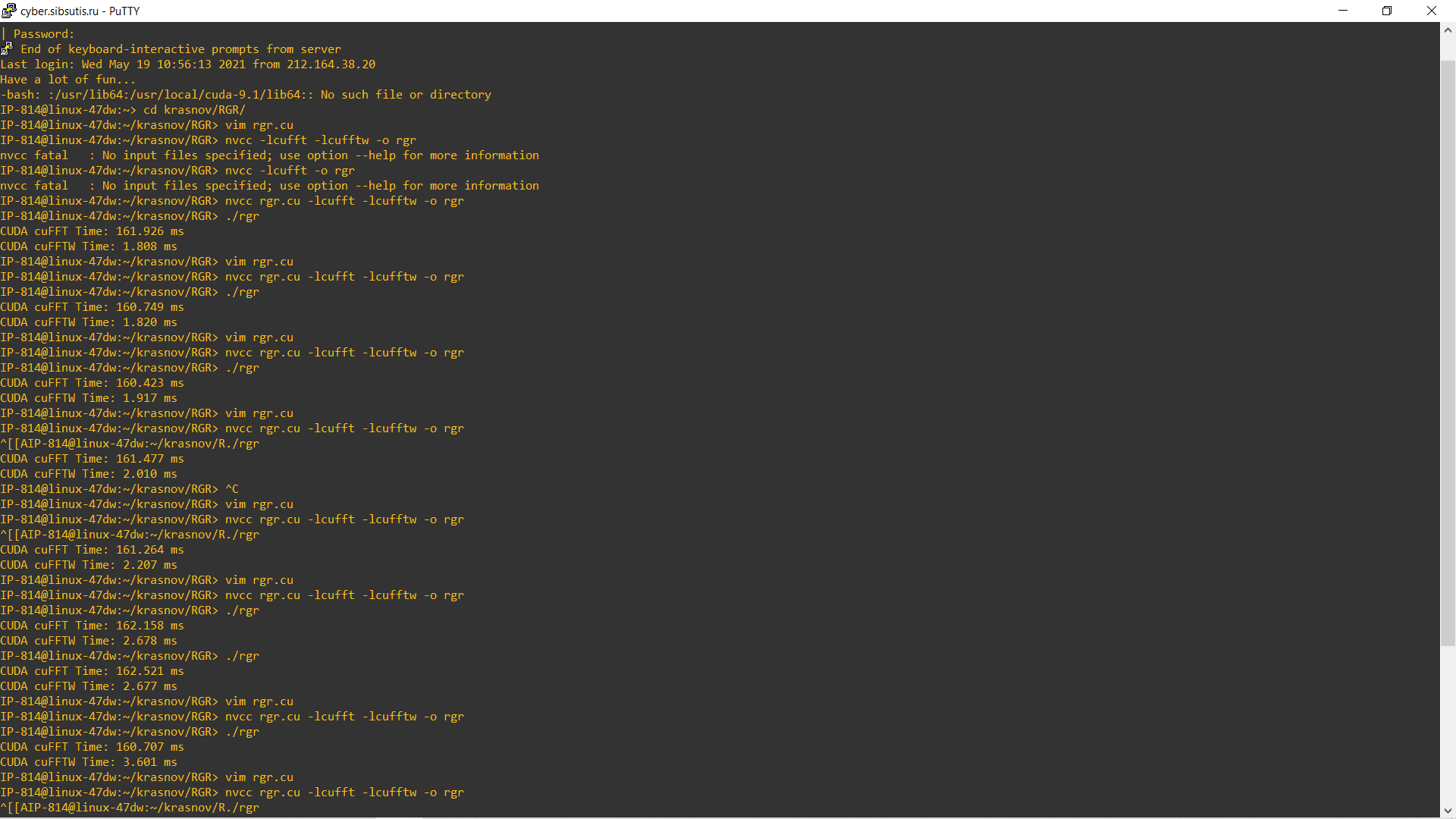
Замеряется не только время выполнения самого алгоритма быстрого преобразования Фурье, но и инициализация памяти на хосте и девайсе, копирование исходных данных с хоста на девайс и копирование результатов с девайса на хост.

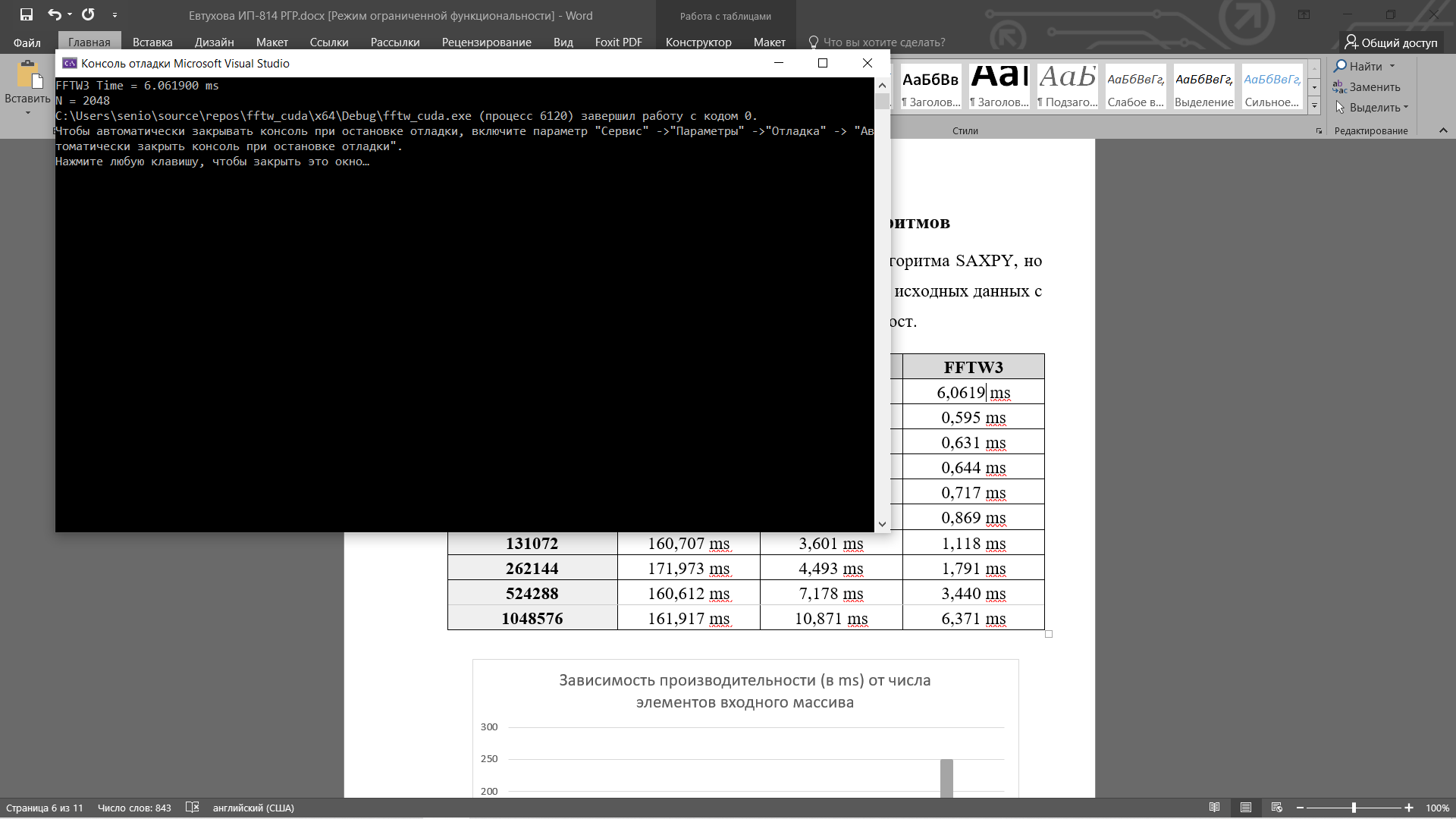
|  |  |  |  |
| --- | --- | --- | --- |
| **Размерность N** | **cuFFT** | **cuFFTW** | **FFTW3** |
| **2048** | 161,926 ms | 1,808 ms | 6,0619 ms |
| **4096** | 160,749 ms | 1,820 ms | 2,289 ms |
| **8192** | 160,423 ms | 1,917 ms | 6,6516 ms |
| **16384** | 161,477 ms | 2,010 ms | 3,3513 ms |
| **32768** | 161,264 ms | 2,207 ms | 22,6635 ms |
| **65536** | 162,158 ms | 2,678 ms | 8,6921 ms |
| **131072** | 160,707 ms | 3,601 ms | 22,0208 ms |
| **262144** | 171,973 ms | 4,493 ms | 37,167 ms |
| **524288** | 160,612 ms | 7,178 ms | 97,2654 ms |
| **1048576** | 161,917 ms | 10,871 ms | 234,5179 ms |

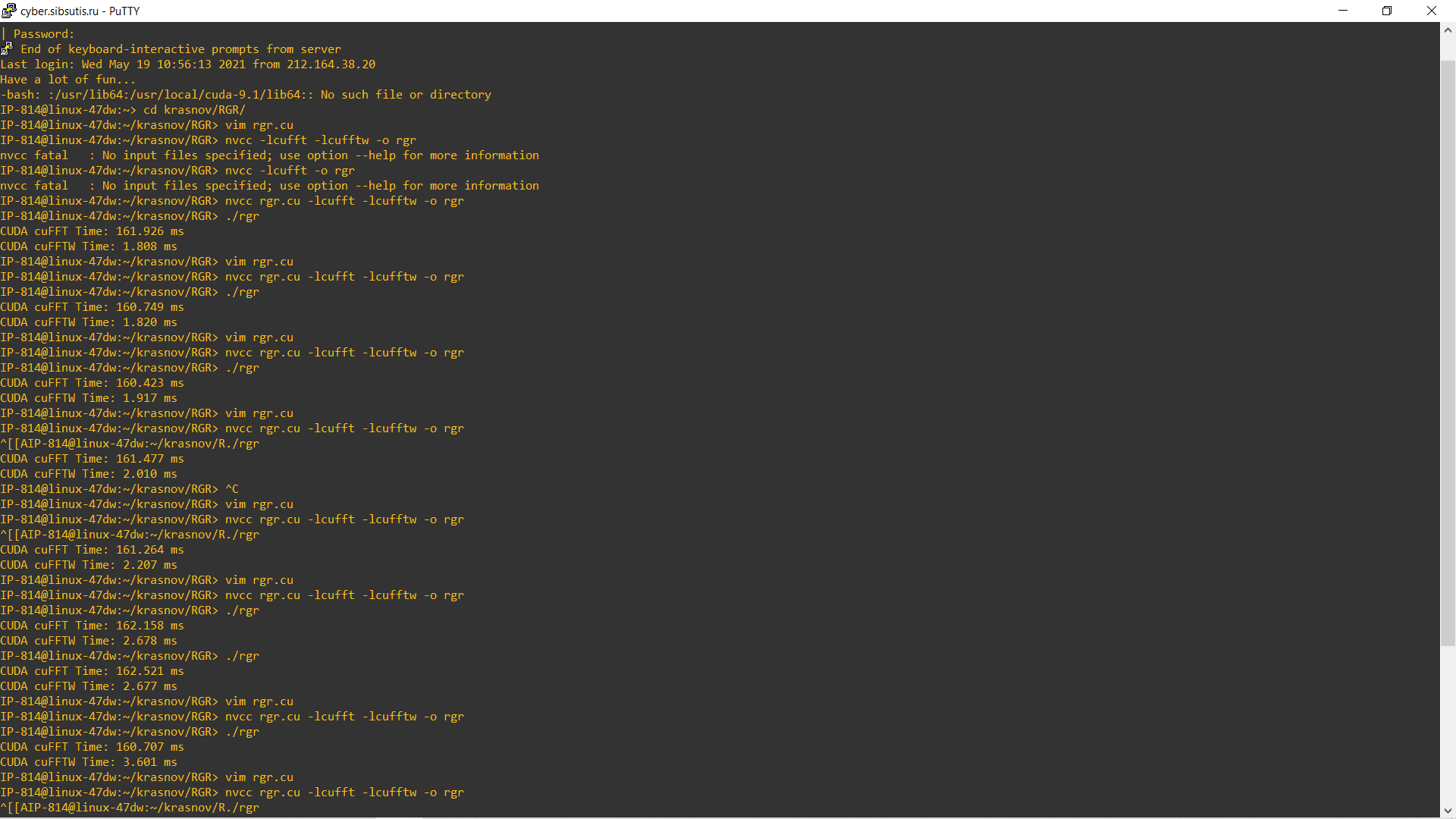
# **Вывод**

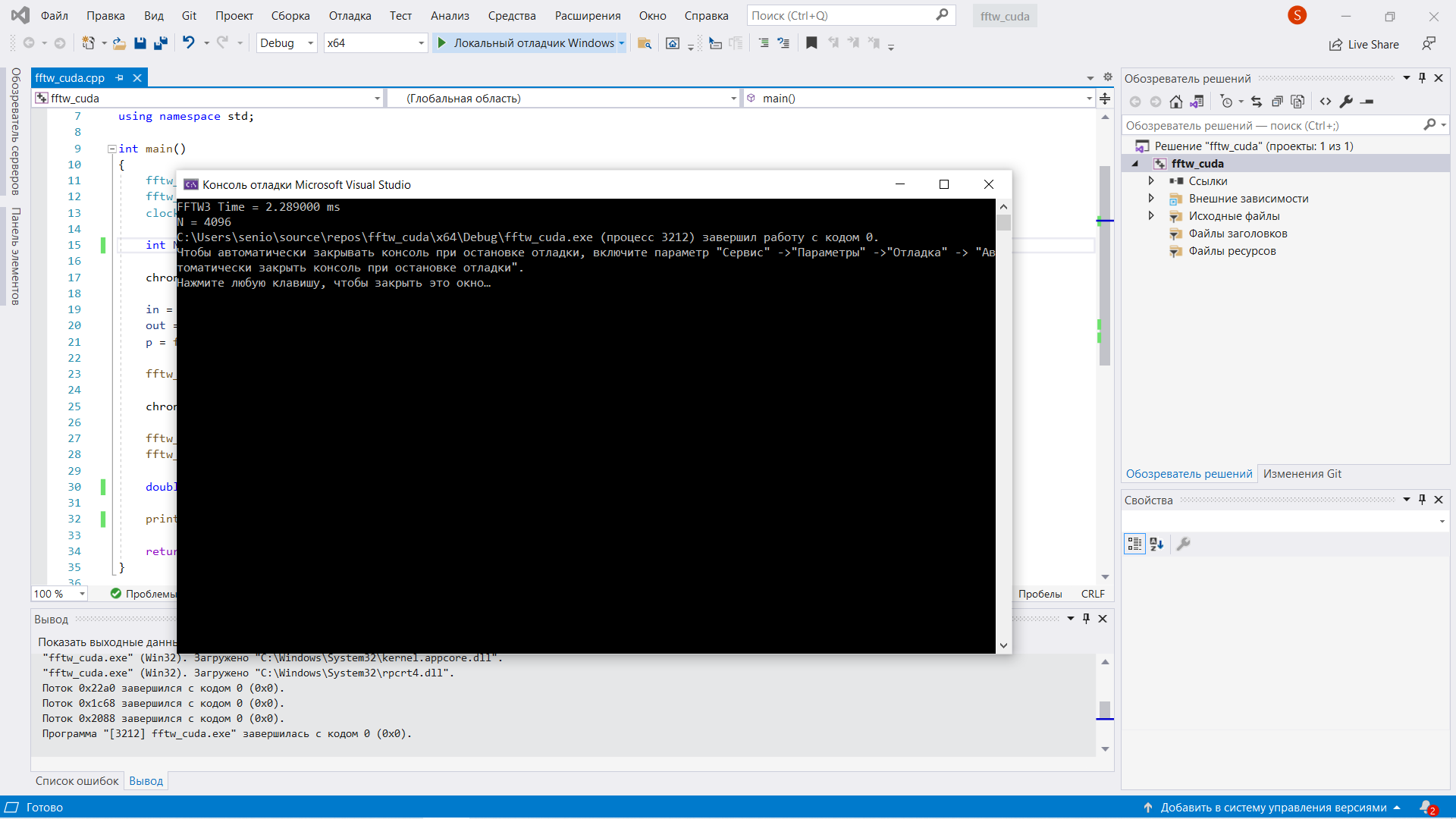
В данном задании мной был реализован алгоритм быстрого преобразования Фурье, с использованием библиотек cuFFT, cuFFTW и FFTW3. Также была определена зависимость их производительности от числа элементов входного массива.

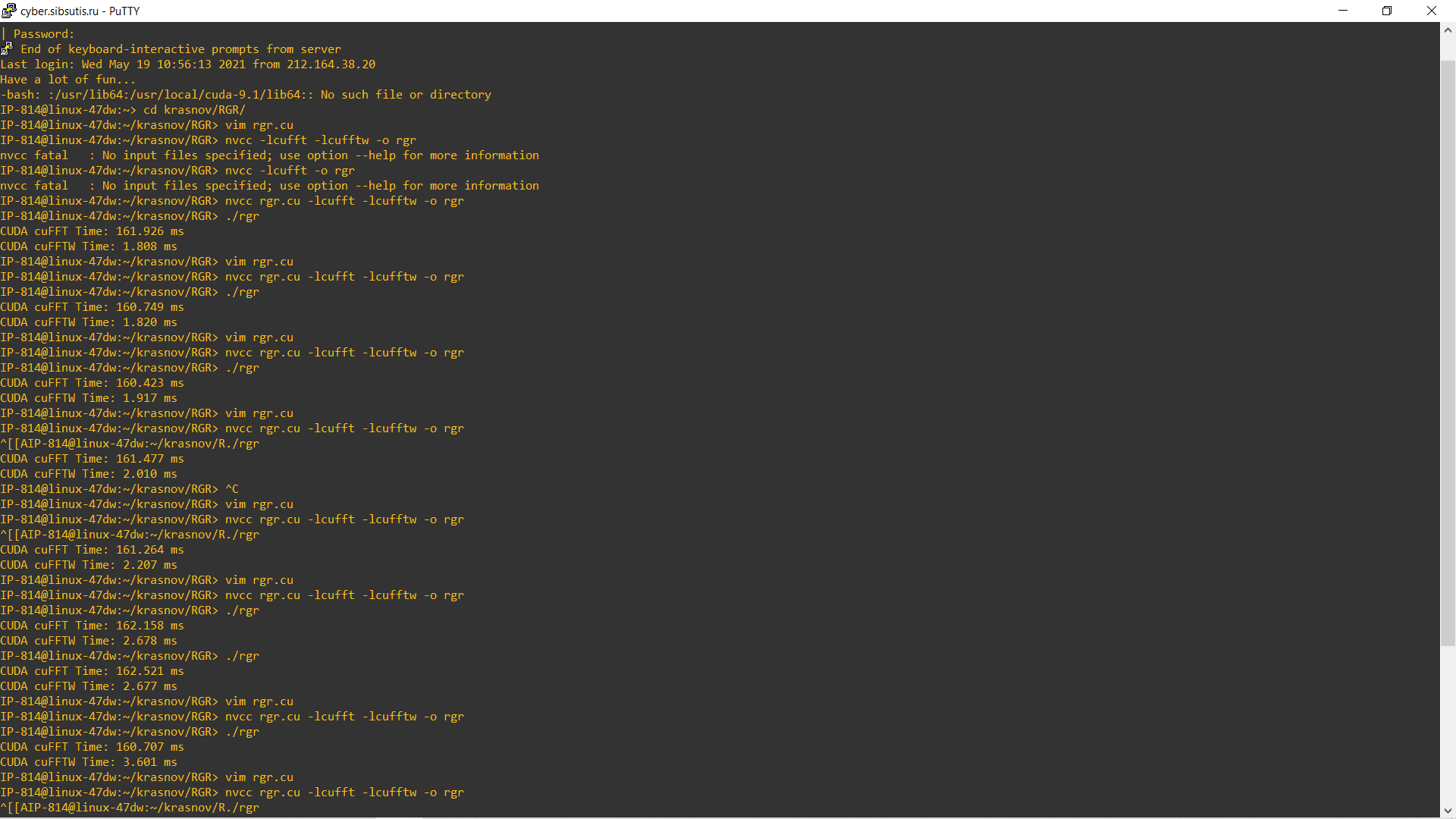
# **Приложения**

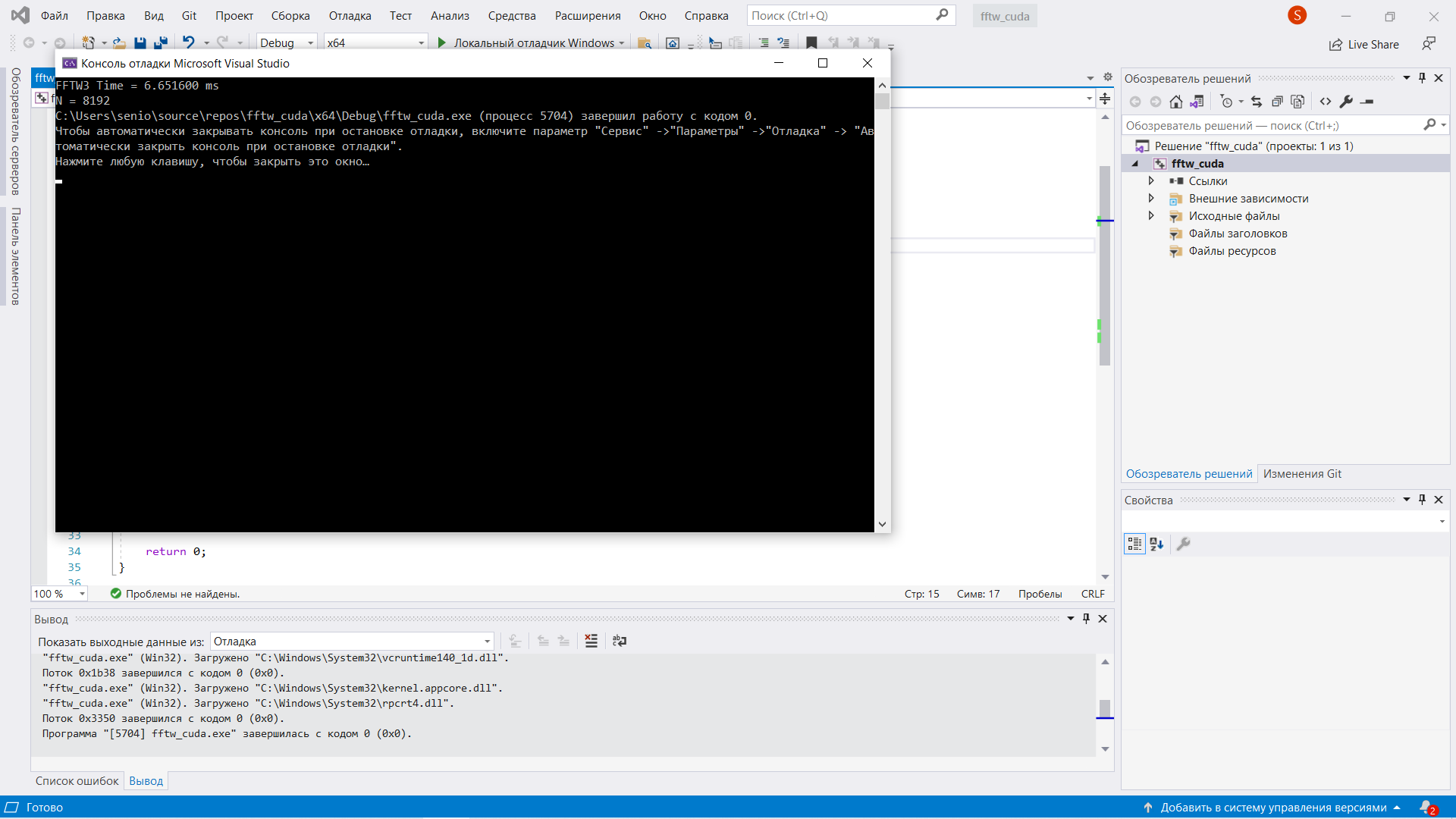


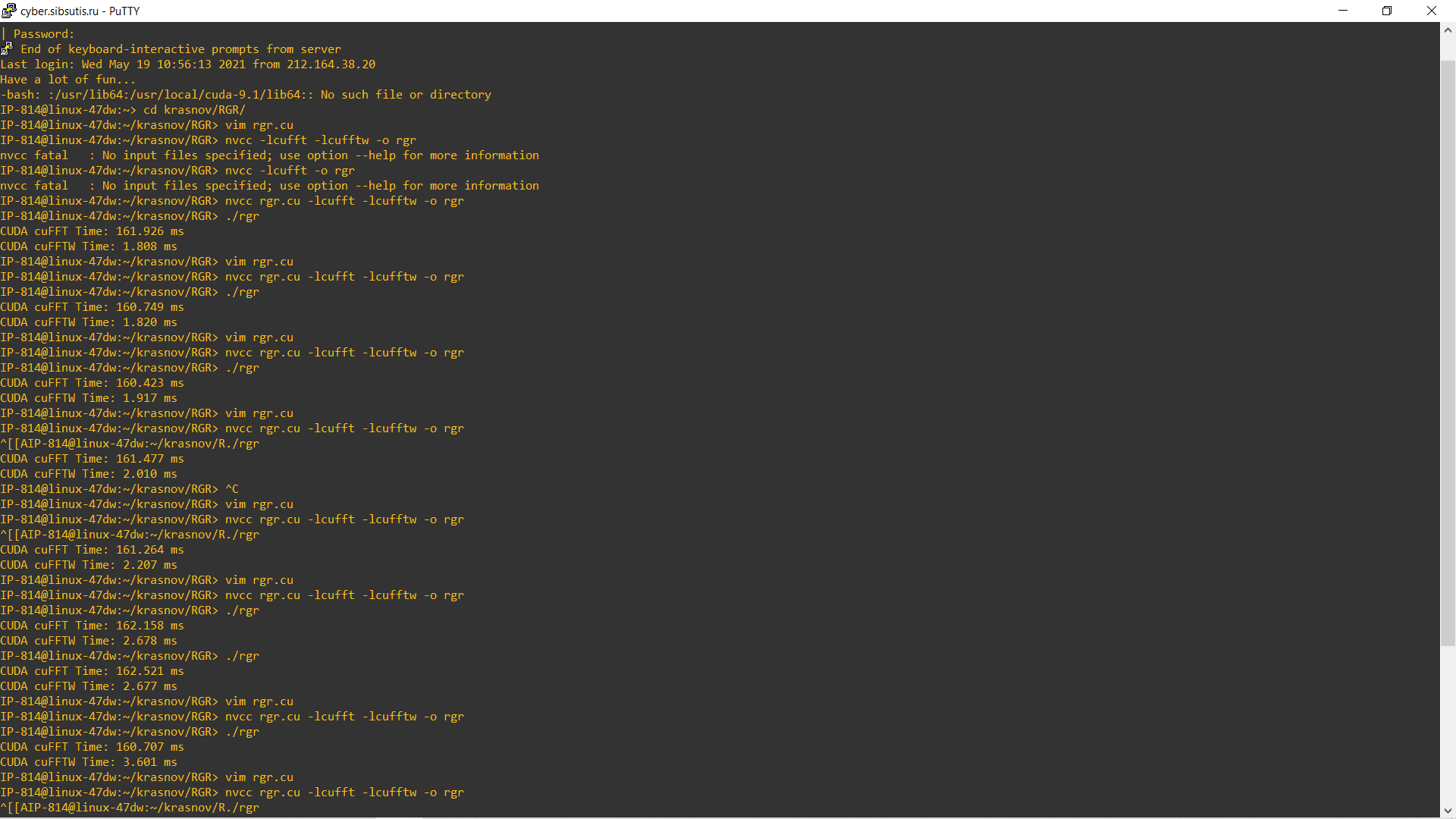


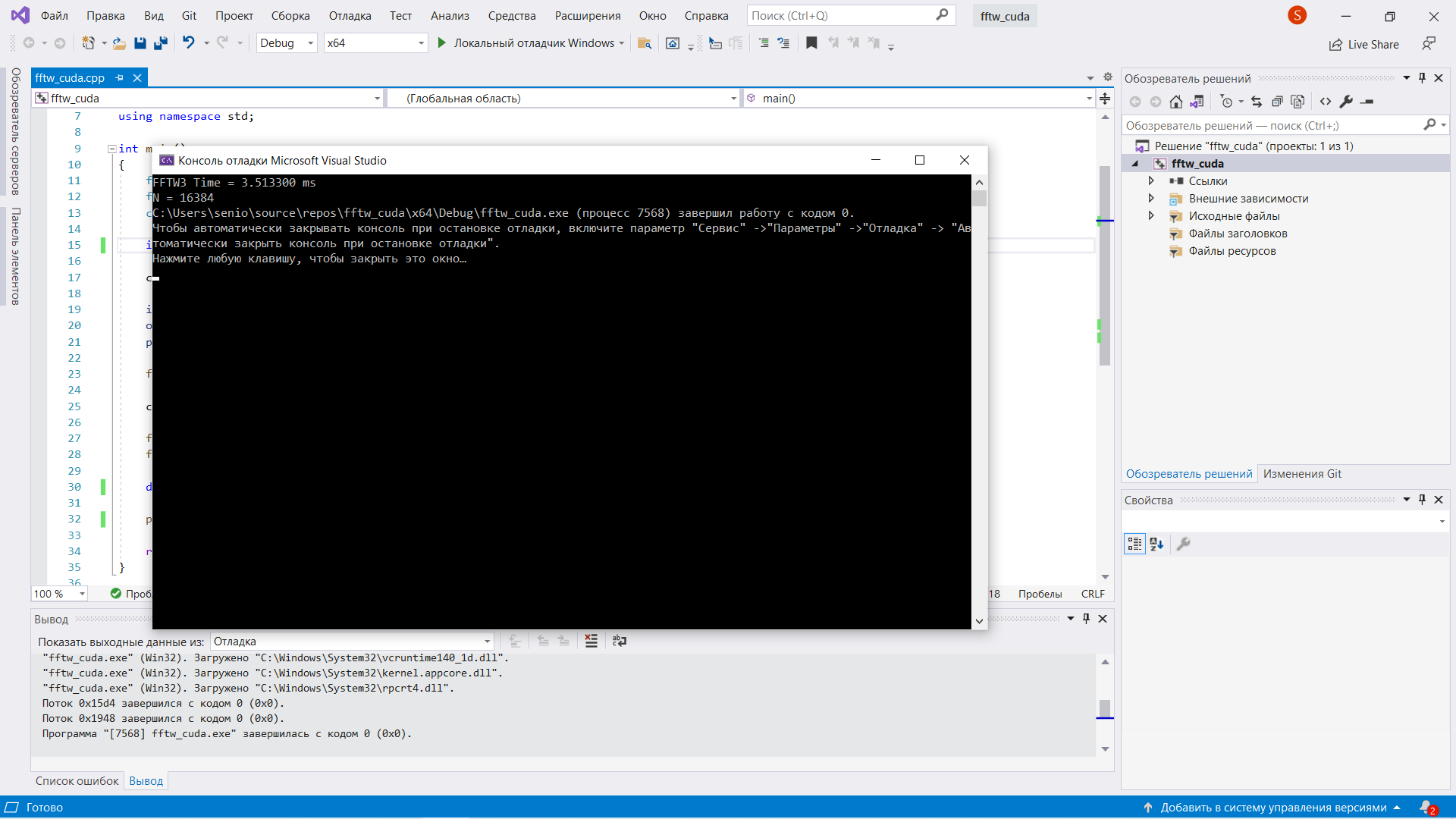


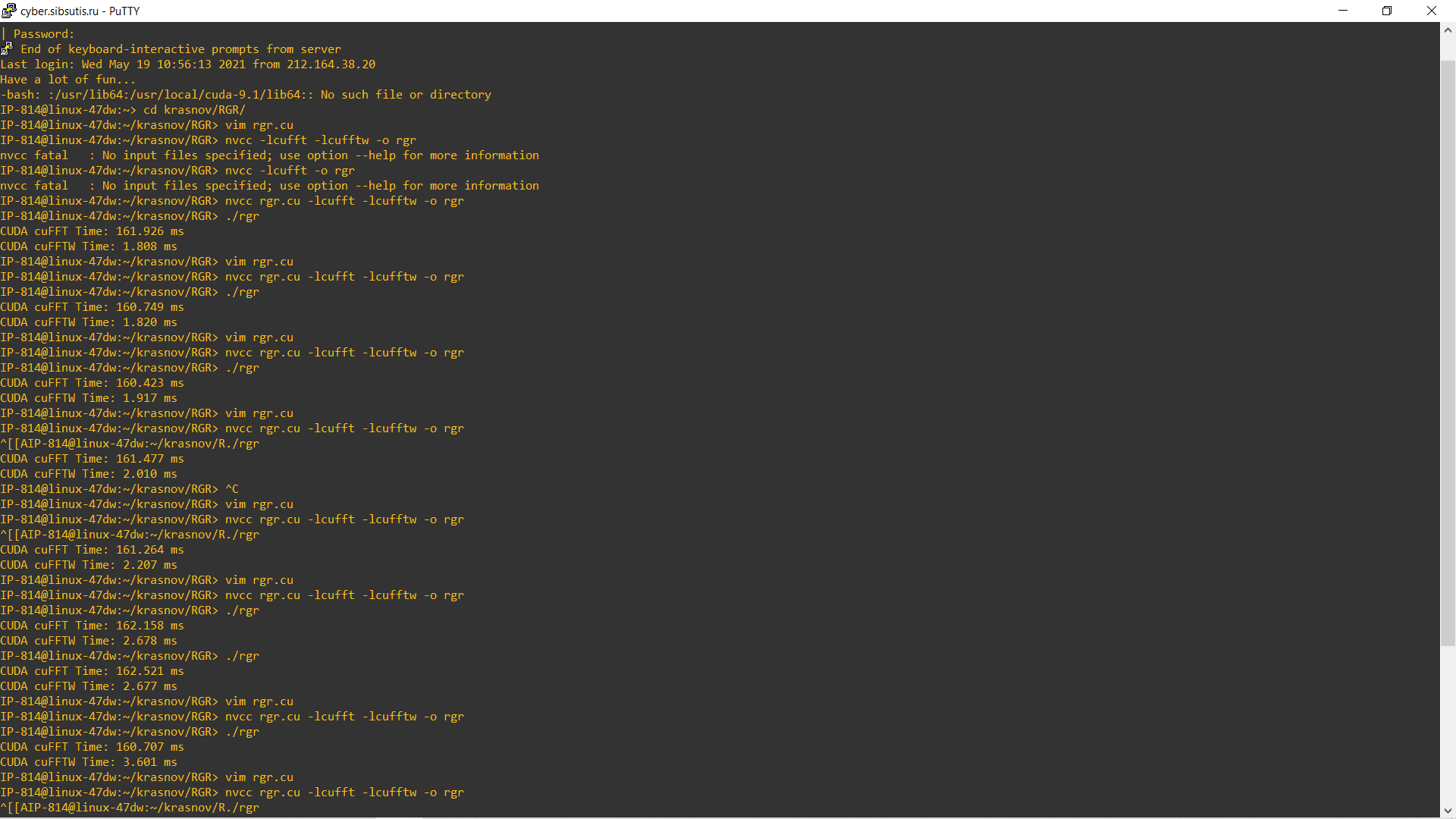


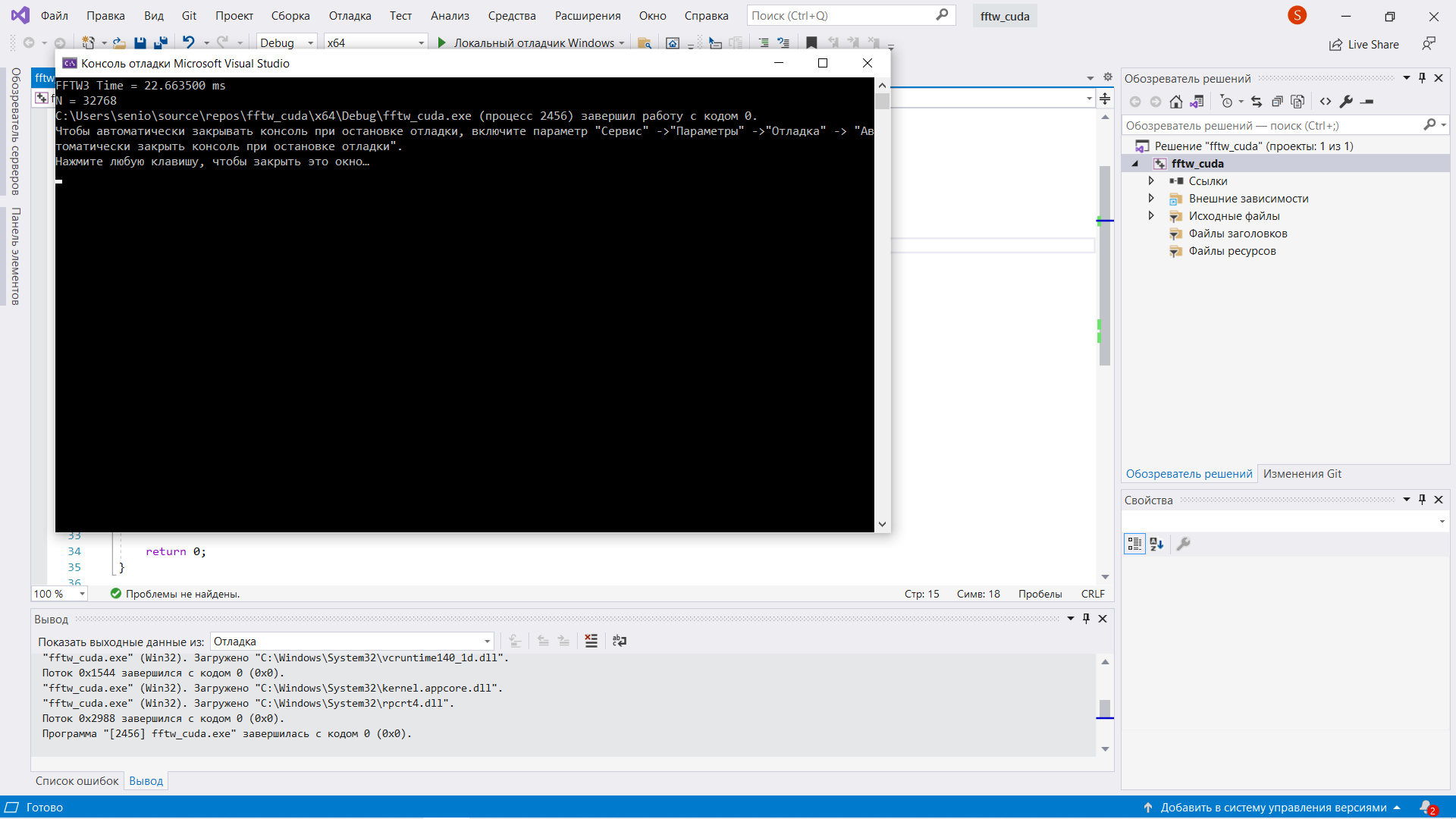


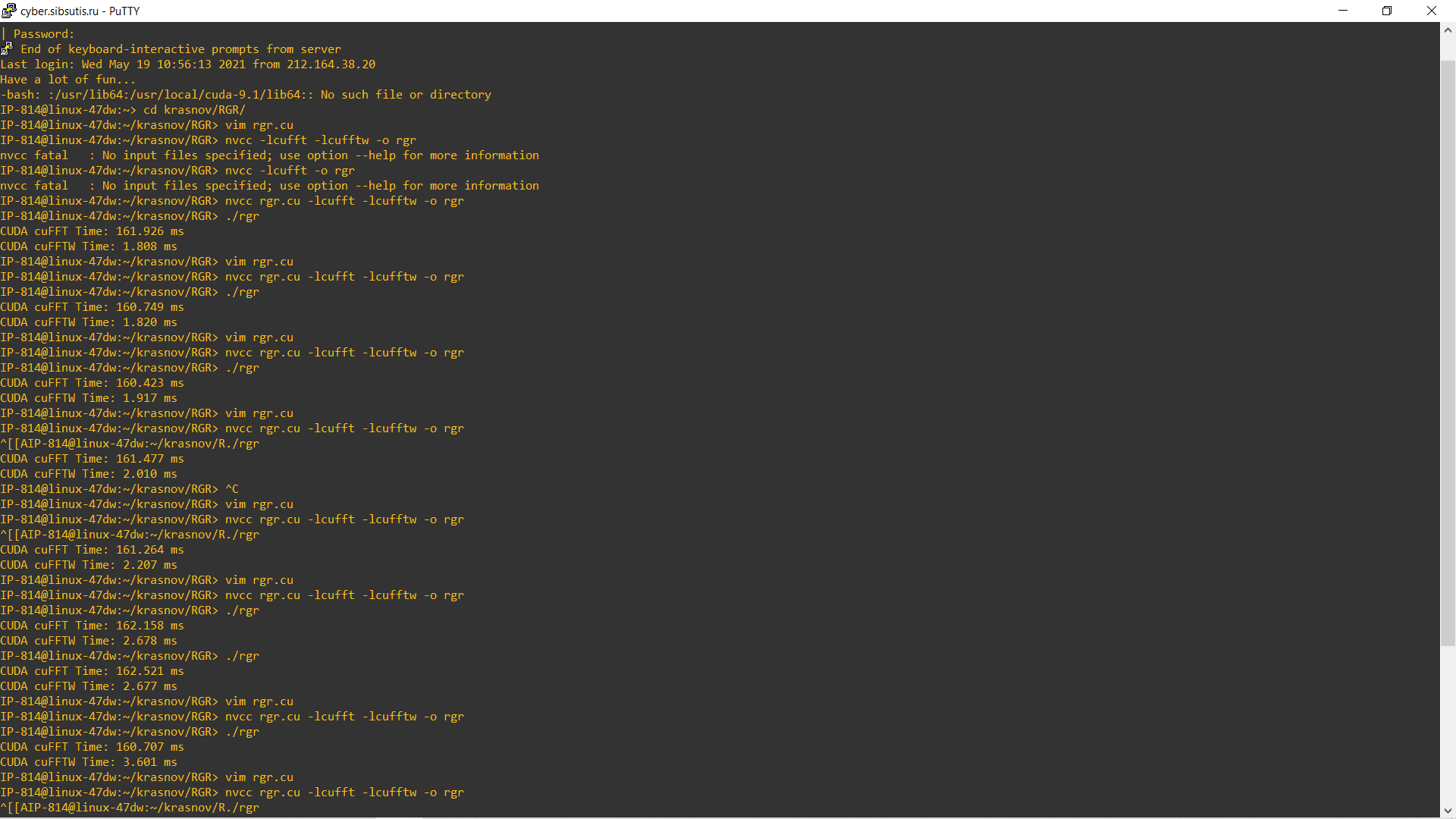


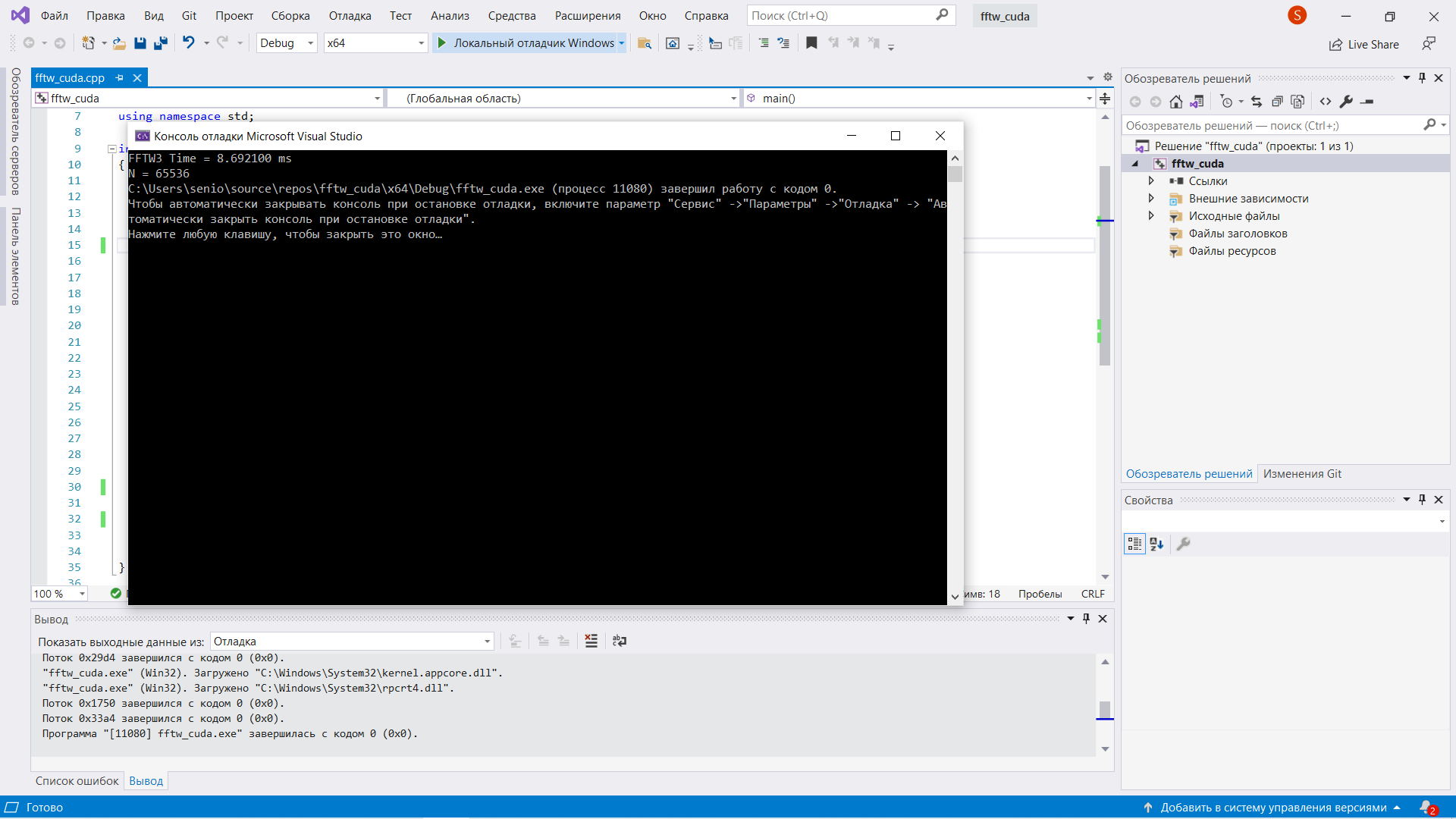


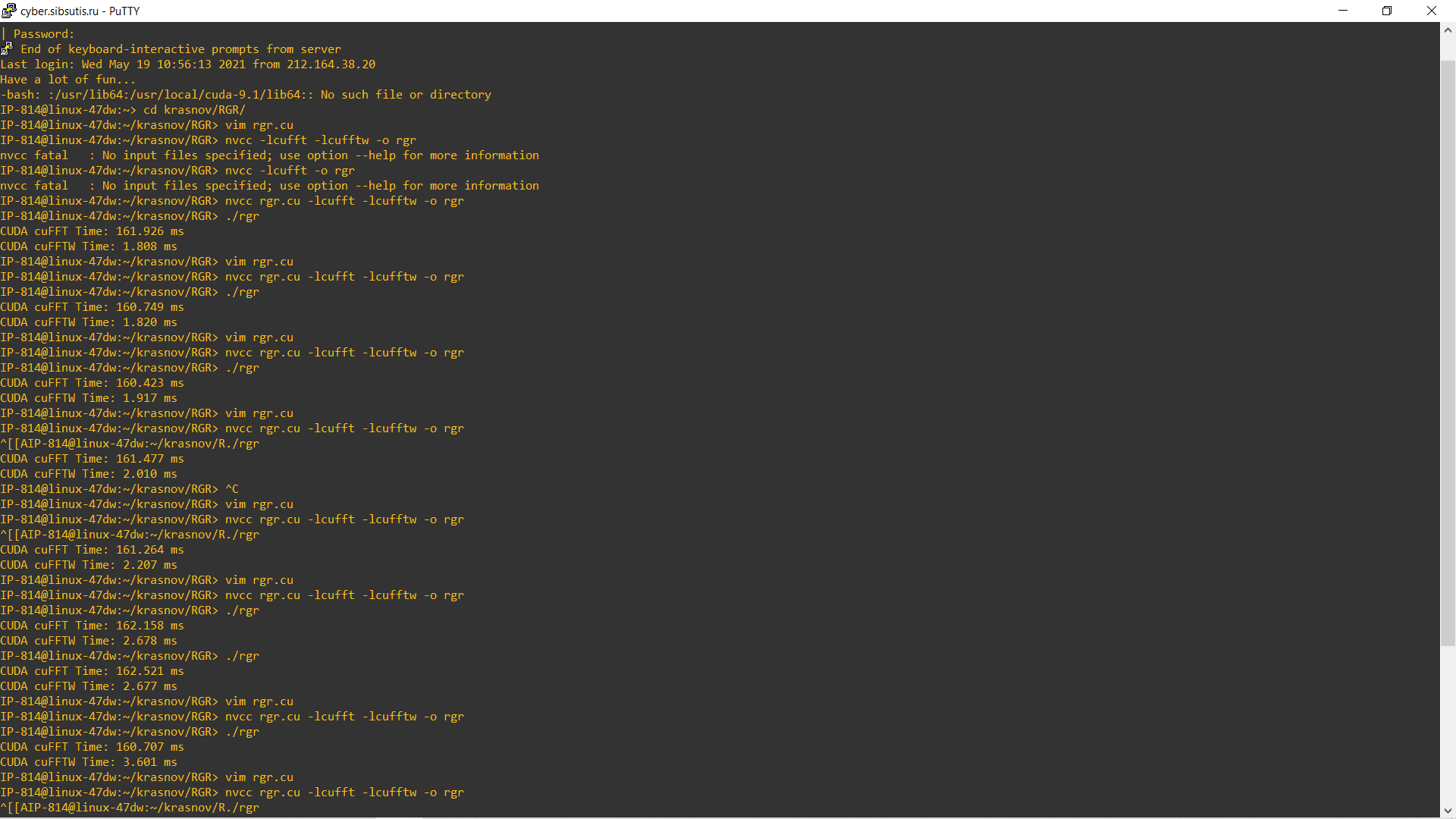


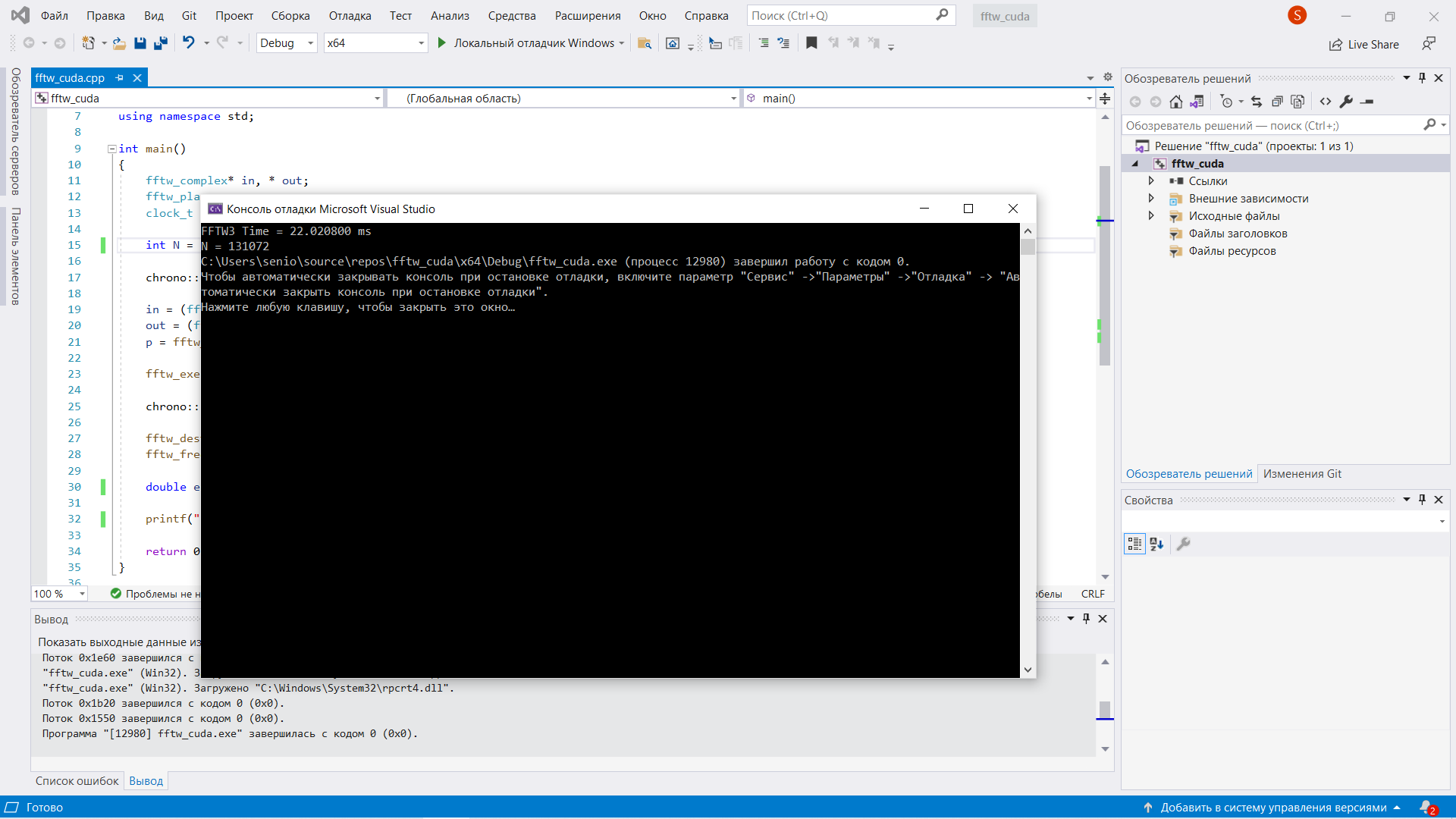


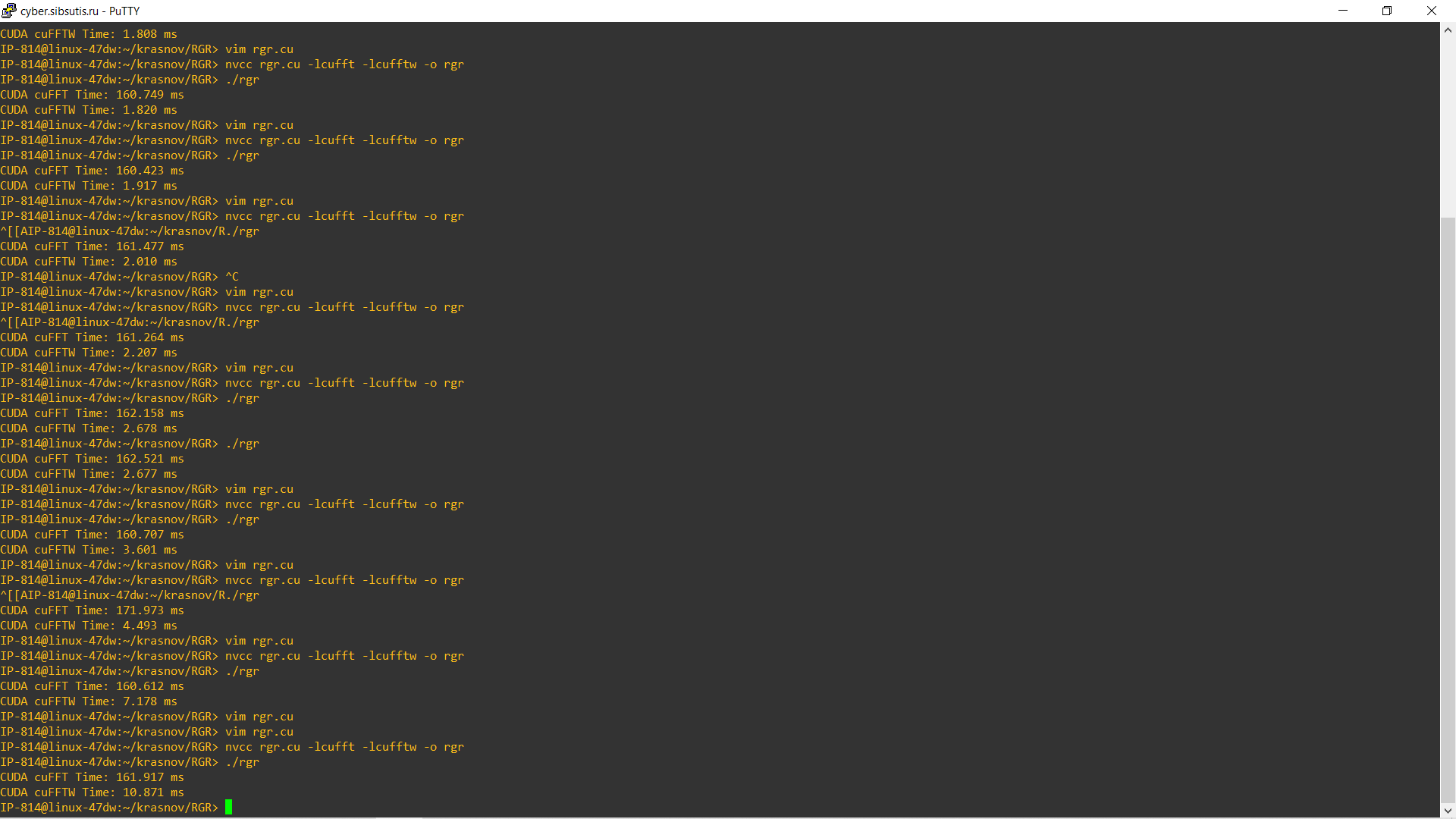


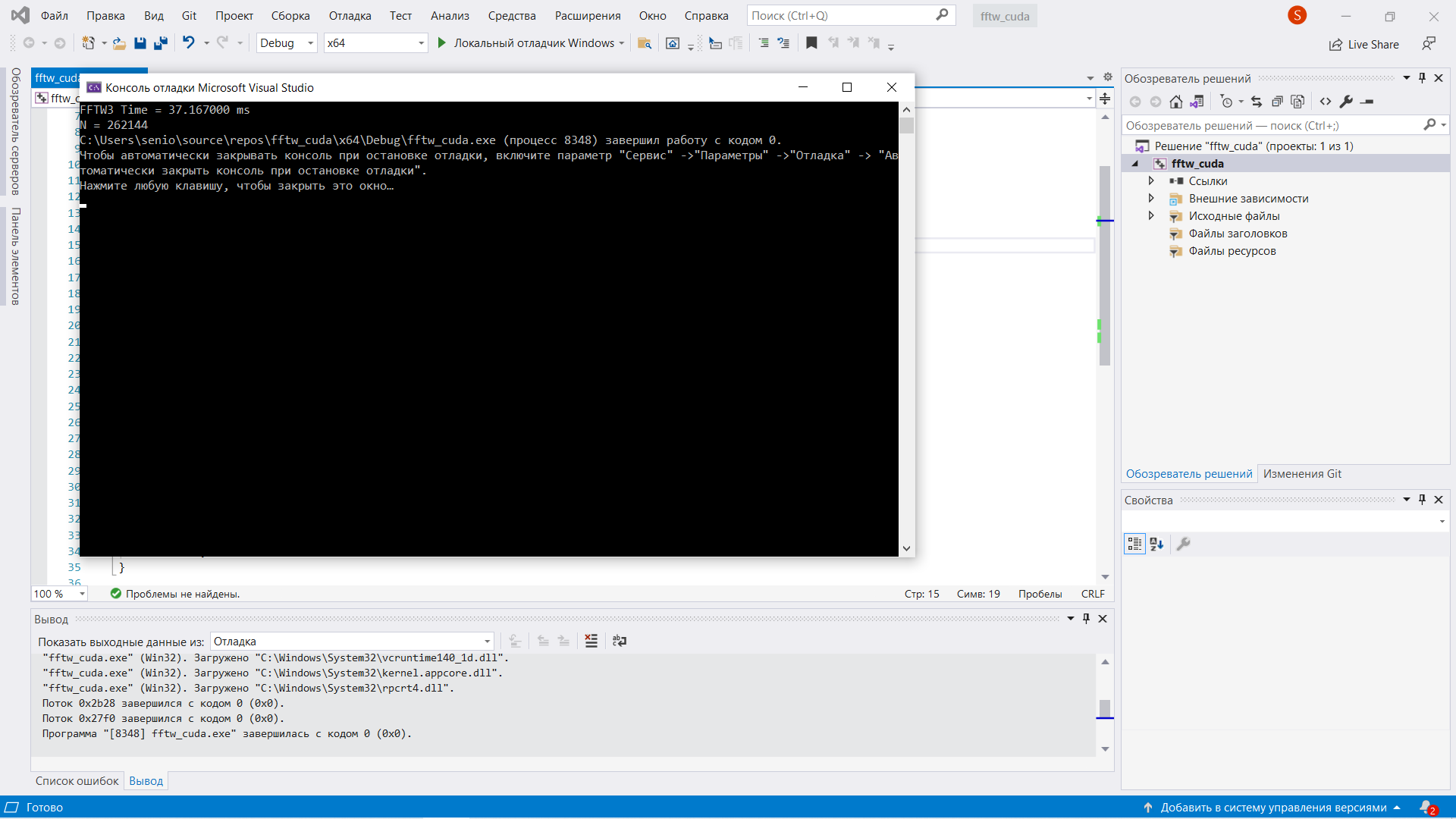


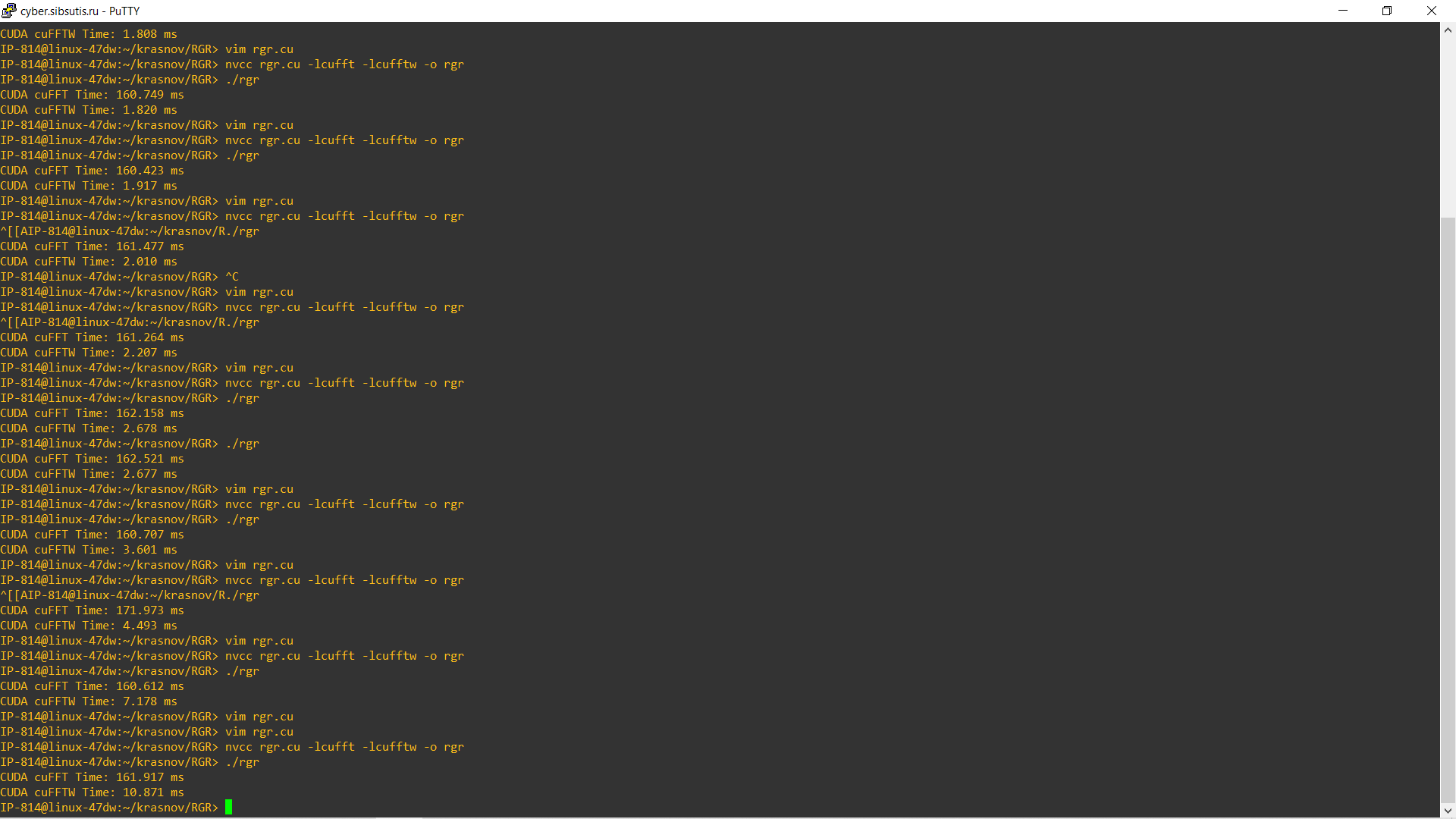


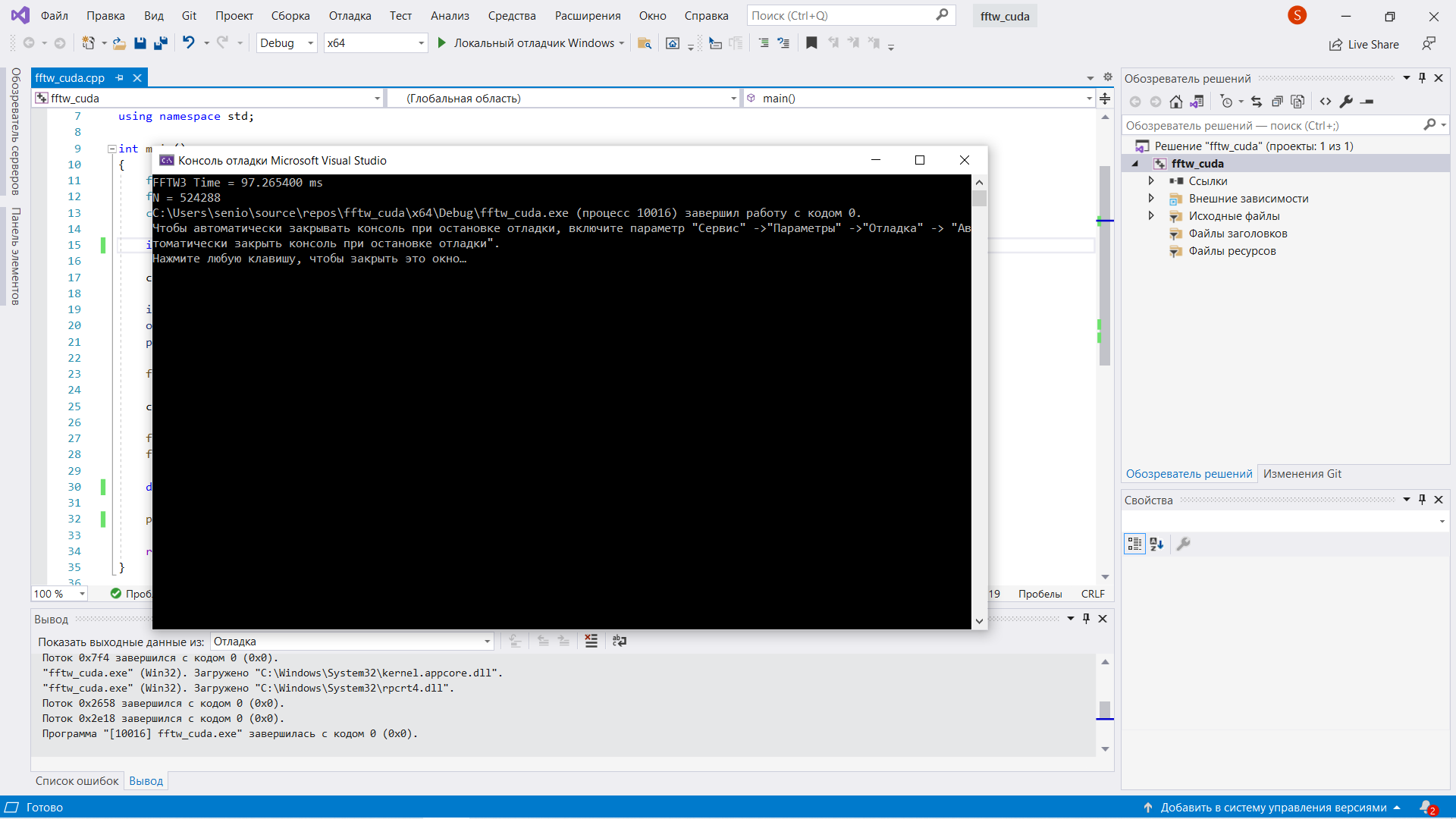


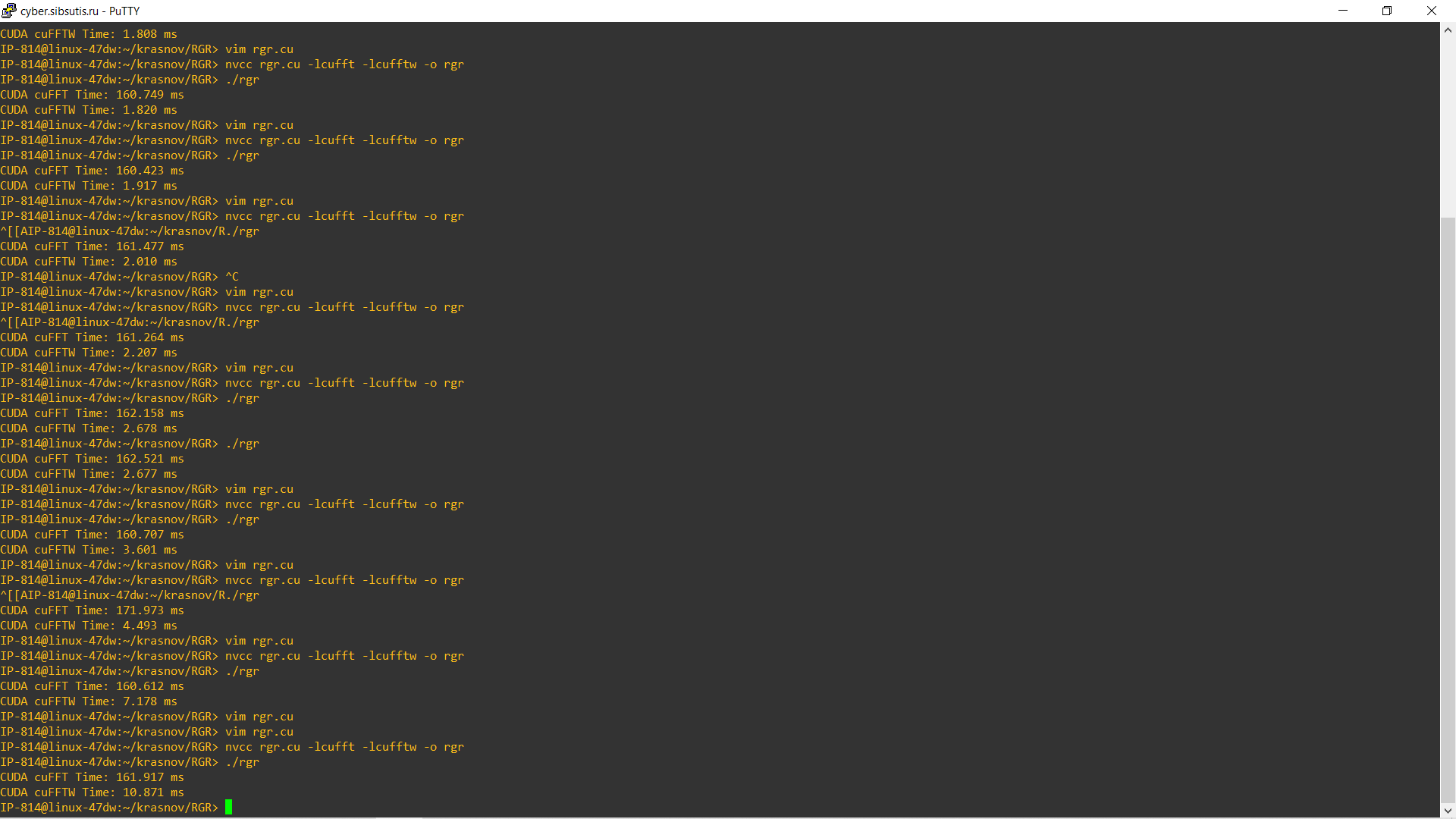


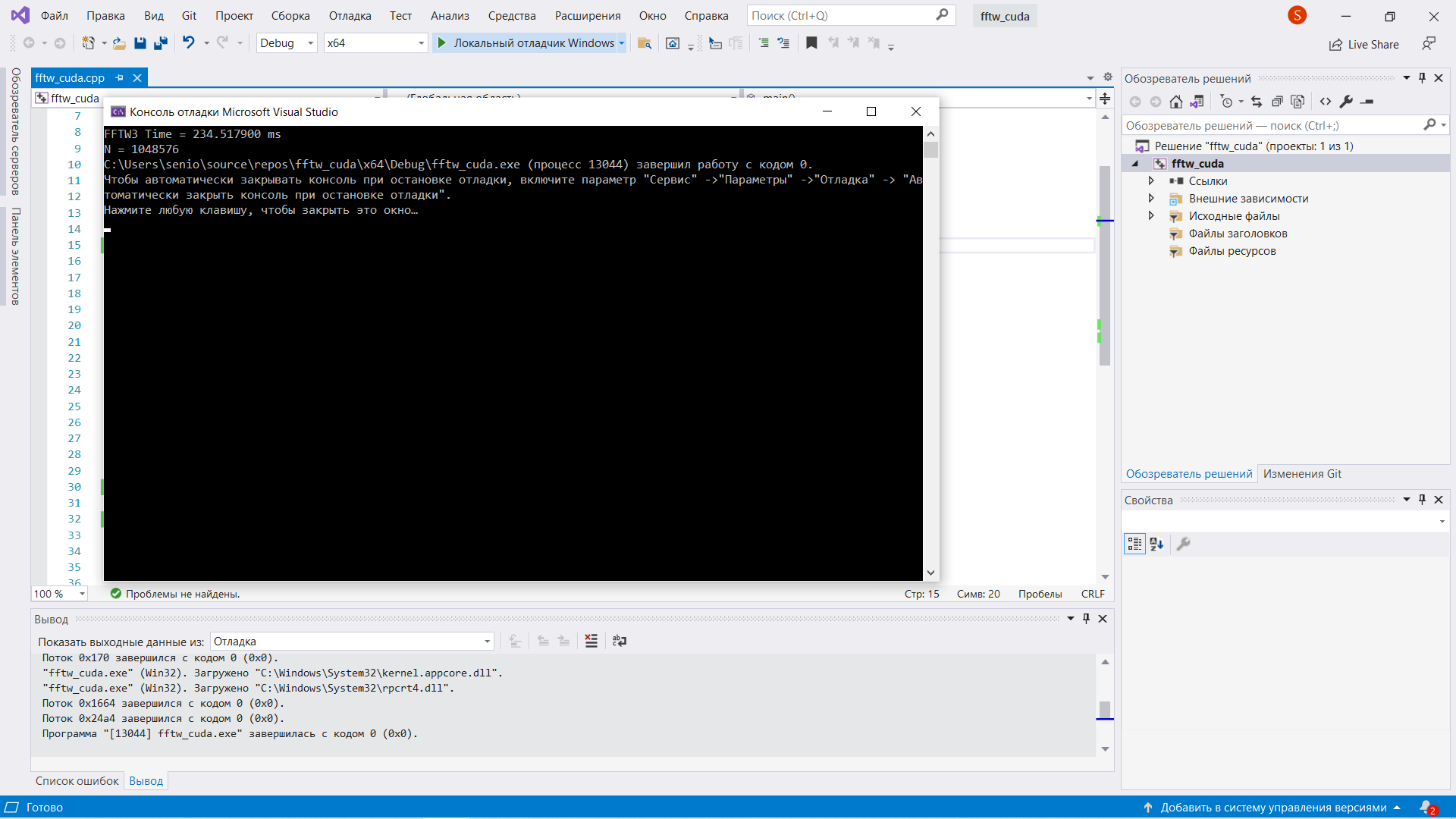












*Результаты работы программы*

# **Код программы**

rgr.cu

#include <cuda.h>

#include <stdio.h>

#include <malloc.h>

#include <math.h>

#include <thrust/device\_vector.h>

#include <thrust/fill.h>

#include <thrust/host\_vector.h>

#include <thrust/sequence.h>

#include <thrust/transform.h>

#include <cufft.h>

#include <cufftw.h>

#define CUDA\_CHECK\_RETURN(value) {\

cudaError\_t \_m\_cudaStat = value;\

if (\_m\_cudaStat != cudaSuccess) {\

fprintf(stderr, "Error %s at line %d in file %s\n",\

cudaGetErrorString(\_m\_cudaStat), \_\_LINE\_\_, \_\_FILE\_\_);\

exit(1);\

}}

float cuFFT\_trans(int N) {

cufftHandle plan;

cufftComplex \*data;

int batch = 1;

cudaMalloc((void\*\*)&data, sizeof(cufftComplex)\*(N/2+1)\*batch);

cudaEvent\_t start, stop;

float gpuTime = 0;

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

cufftPlan1d(&plan, N, CUFFT\_R2C, batch);

cufftExecR2C(plan, (cufftReal\*)data, data);

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&gpuTime, start, stop));

cufftDestroy(plan);

cudaFree(data);

return gpuTime;

}

float cuFFTW\_trans(int N) {

fftw\_complex \*in, \*out;

fftw\_plan p;

cudaEvent\_t start, stop;

float gpuTime = 0;

CUDA\_CHECK\_RETURN(cudaEventCreate(&start));

CUDA\_CHECK\_RETURN(cudaEventCreate(&stop));

CUDA\_CHECK\_RETURN(cudaEventRecord(start, 0));

in = (fftw\_complex\*) fftw\_malloc(sizeof(fftw\_complex) \* N);

out = (fftw\_complex\*) fftw\_malloc(sizeof(fftw\_complex) \* N);

p = fftw\_plan\_dft\_1d(N, in, out, FFTW\_FORWARD, FFTW\_ESTIMATE);

fftw\_execute(p);

CUDA\_CHECK\_RETURN(cudaEventRecord(stop, 0));

CUDA\_CHECK\_RETURN(cudaEventSynchronize(stop));

CUDA\_CHECK\_RETURN(cudaEventElapsedTime(&gpuTime, start, stop));

fftw\_destroy\_plan(p);

fftw\_free(in);

fftw\_free(out);

return gpuTime;

}

int main(){

int N = 1048576;

float cuFFTTime = cuFFT\_trans(N);

printf("CUDA cuFFT Time: %.3f ms\n", cuFFTTime);

float cuFFTWTime = cuFFTW\_trans(N);

printf("CUDA cuFFTW Time: %.3f ms\n", cuFFTWTime);

return 0;

}

fftw3.c

#include <fftw3.h>

#include <iostream>

#include <conio.h>

#include <ctime>

#include <chrono>

using namespace std;

int main()

{

fftw\_complex\* in, \* out;

fftw\_plan p;

clock\_t start = clock();

int N = 1048576;

chrono::system\_clock::time\_point start\_time = chrono::system\_clock::now();

in = (fftw\_complex\*)fftw\_malloc(sizeof(fftw\_complex) \* N);

out = (fftw\_complex\*)fftw\_malloc(sizeof(fftw\_complex) \* N);

p = fftw\_plan\_dft\_1d(N, in, out, FFTW\_FORWARD, FFTW\_ESTIMATE);

fftw\_execute(p); /\* repeat as needed \*/

chrono::system\_clock::time\_point end\_time = chrono::system\_clock::now();

fftw\_destroy\_plan(p);

fftw\_free(in); fftw\_free(out);

double elaps = chrono::duration\_cast<std::chrono::nanoseconds>(end\_time - start\_time).count();

printf("FFTW3 Time = %f ms\nN = %d", elaps / 1000000, N);

return 0;

}