Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Лабораторная работа № 5
«Профилирование на языке высокого уровня (C++)»

Выполнила:
Емельянова М.А.
Группа: Q4110
Проверила:
Иванова Т.В.

Санкт-Петербург
2024

Цель работы:

Исследовать программный код собственного разработанного приложения и применить к нему профилирование.

Ход работы:

Профилирование в C++ – это процесс анализа и измерения производительности программы с целью выявления узких мест, оптимизации кода и улучшения общей эффективности работы программы.

1) Разработанное мною приложение подразумевало использование математических функций и операторов. Первым делом был осуществлен поиск во всех документах проекта на наличие метода ***pow()*** – в проекте он не использовался – на этом уже изначально было сэкономлено время выполнения.

2) В C++ ключевое слово ***«inline»*** используется для указания компилятору, что функция должна быть встроена (inline) в место ее вызова. Обычно функции, которые могут быть встроены, это короткие функции, которые вызываются часто. Встраивание таких функций может улучшить производительность программы за счет уменьшения накладных расходов на вызов функции.

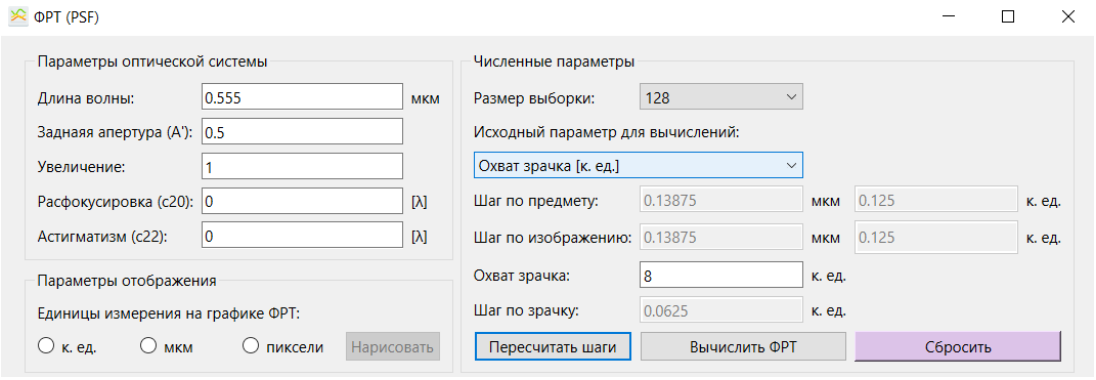
Ниже представлена основная функция расчета ФРТ:

```
6   void CalcPSF::Calc(const OpticalParameters& opt_params, Sample<double>& PSF) {
7
8       PSF.Resize(opt_params.m_int_sample_sizes);
9       PSF.SetValue(0.);
10
11      int count = 100;
12      for (int i = 0; i < count; i++) {
13          // Зрачковая функция
14          SampleComplex Pupil(opt_params.m_int_sample_sizes);
15          CalcPupilFunction(opt_params, Pupil);
16
17          // Обратное Фурье
18          CalcFFT(Pupil);
19
20          // Нормировка
21          Pupil *= std::complex<double>(opt_params.m_double_step_pupil / opt_params.m_double_step_obj_can, 0.);
22
23          // Преобразование комплексных чисел в вещественные
24          Sample<double> pupil_double = Pupil.GetIntensity().ComplexToDouble();
25
26          PSF = pupil_double;
27          PSF *= (1 / (PI * PI));
28      }
29      PSF /= count;
30  }
```

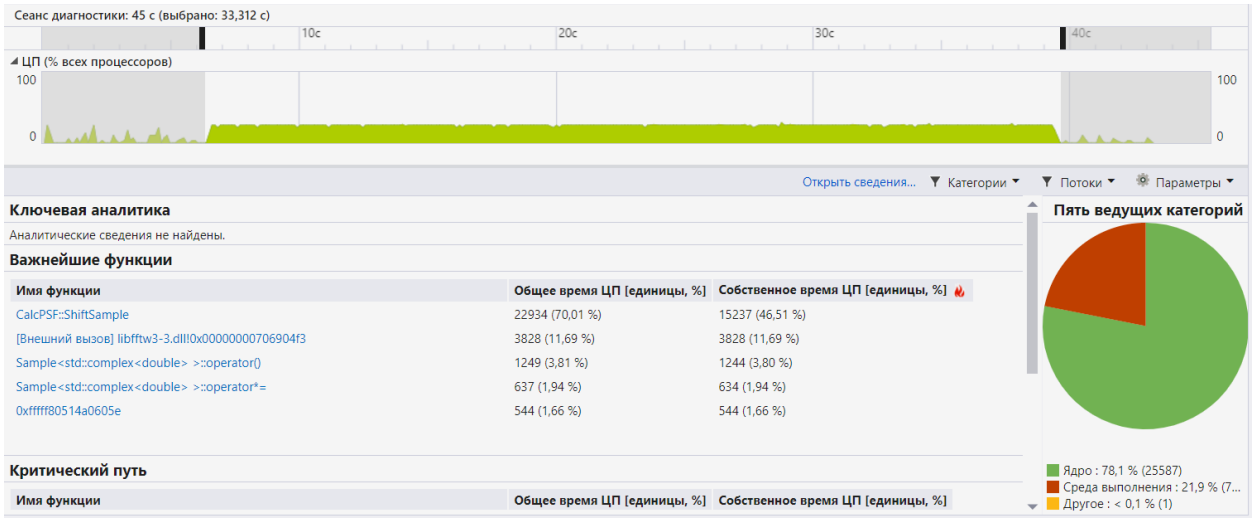
Для отладки оптимизации обычно используется цикл с некоторым количеством повторений, на котором можно ощутить и увидеть проблемные места реализованного кода. Вызов функции расчета ФРТ был модифицирован следующим образом:

```
6   void CalcPSF::Calc(const OpticalParameters& opt_params, Sample<double>& PSF) {
7
8       PSF.Resize(opt_params.m_int_sample_sizes);
9       PSF.SetValue(0.);
10
11      int count = 100;
12      for (int i = 0; i < count; i++) {
13          // Зрачковая функция
14          SampleComplex Pupil(opt_params.m_int_sample_sizes);
15          CalcPupilFunction(opt_params, Pupil);
16
17          // Обратное Фурье
18          CalcFFT(Pupil);
19
20          // Нормировка
21          Pupil *= std::complex<double>(opt_params.m_double_step_pupil / opt_params.m_double_step_obj_can, 0.);
22
23          // Преобразование комплексных чисел в вещественные
24          Sample<double> pupil_double = Pupil.GetIntensity().ComplexToDouble();
25
26          PSF = pupil_double;
27          PSF *= (1 / (PI * PI));
28      }
29      PSF *= 1 / count;
30  }
```

Оптические параметры во всех измерениях задавались одни и те же. Их значения представлены ниже:



Время выполнения программного кода без внесенных изменений составляет 33.312 с (стартовое время):



Попробуем применить inline к базовому классу Sample. Ниже представлен исходный класс без изменений:

```
 // Базовый класс для выборки (квадратной или прямоугольной)
 template <class PAR>
class Sample
{
protected:
    // Число элементов выборки по x и y
    int m_size_x = 0, m_size_y = 0;
    // Массив данных
    std::vector<PAR> m_data;

public:
    // Конструктор
    Sample() = default;
    Sample(int m_size_x) : m_size_x(m_size_x), m_size_y(m_size_x), m_data(m_size_x* m_size_x) {}
    Sample(int m_size_x, int m_size_y) : m_size_x(m_size_x), m_size_y(m_size_y), m_data(m_size_x* m_size_y) {}
    // Деструктор
    ~Sample() = default;

    // Изменяет размер для квадратной выборки
    void Resize(int size);
    // Измененяет размер для прямоугольной выборки
    void Resize(int size_x, int size_y);
    // Печатает элементы выборки
    void PrintMatrix() const;

    // Возвращает размер выборки (для квадратной)
    int GetSize() const;
    // Возвращает размер выборки по X
    int GetSizeX() const;
```

Ниже можно наблюдать внесенные изменения:

```
    // Возвращает размер выборки (для квадратной)
    inline int GetSize() const;
    // Возвращает размер выборки по X
    inline int GetSizeX() const;
    // Возвращает размер выборки по Y
    inline int GetSizeY() const;

    // Оператор получения значения элемента с номером i,j
    PAR& operator()(int i, int j);
    // оператор получения const значения элемента с номером i,j
    const PAR& operator()(int i, int j) const;

    // Осуществляет проверку: квадратная ли выборка
    inline bool IsSquare() const;
    // Осуществляет проверку: размер != 0
    inline bool IsZeroSize() const;
    // Осуществляет проверку равенства размеров с другой выборкой
    bool IsEqualSize(const Sample<PAR>& temp) const;
```

Получилось уменьшить время выполнения. Оно теперь составляет 31.515 секунд:

3) При проходе по внешнему циклу, данные, к которым обращается вложенный цикл, могут быть загружены в кэш памяти и использоваться повторно для каждой итерации внутреннего цикла. Это уменьшает количество обращений к памяти и может улучшить производительность. Кроме того, в некоторых случаях проход по внешнему циклу может обеспечить лучшую

возможность для параллелизма, так как итерации внешнего цикла могут быть независимыми и выполняться параллельно на многопроцессорной системе.

Проверим, действительно ли это работает. Ниже представлен код до изменений:

```
46  Sample<double> SampleComplex::ComplexToDouble() {
47      Sample<double> return_data(this->GetSizeX(), this->GetSizeY());
48      for (int i = 0; i < this->GetSizeX(); i++) {
49          for (int j = 0; j < this->GetSizeY(); j++) {
50              return_data(i, j) = std::real((*this)(i, j)); // сохраняем вещественную часть комплексного числа
51          }
52      }
53      return return_data;
54  }
```

Ниже – после внесенных правок:

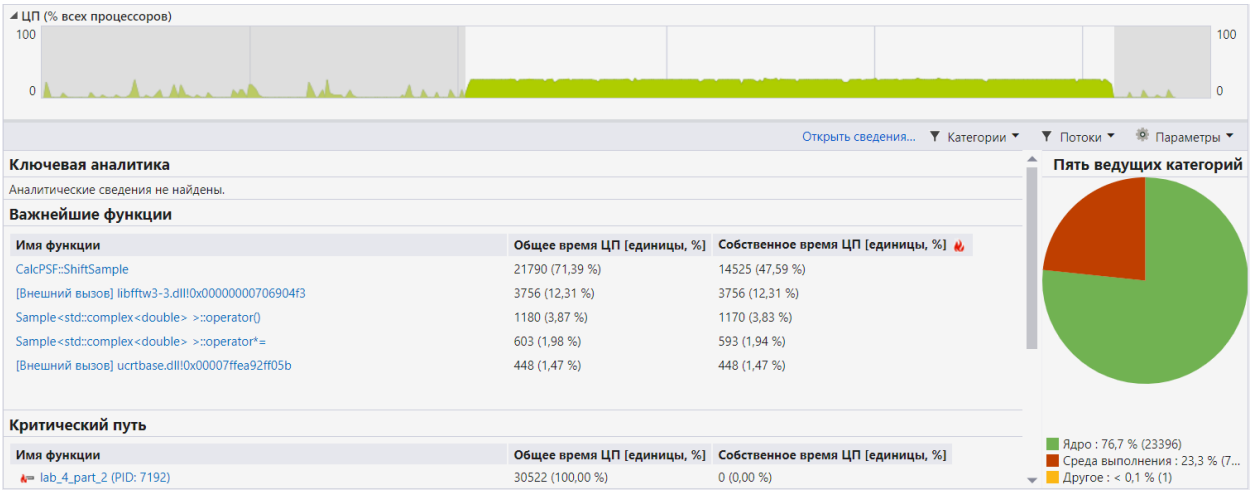
```
46  Sample<double> SampleComplex::ComplexToDouble() {
47      Sample<double> return_data(this->GetSizeX(), this->GetSizeY());
48      for (int j = 0; j < this->GetSizeY(); j++) {
49          for (int i = 0; i < this->GetSizeX(); i++) {
50              return_data(i, j) = std::real((*this)(i, j)); // сохраняем вещественную часть комплексного числа
51          }
52      }
53      return return_data;
54  }
```

В моем случае – это не помогло. Поэтому откатываемся назад.

Реализация операторов +=, *=, /= изначально была написано корректно (использовался наилучший вариант реализации – использовался одномерный массив):

```
184  template <class PAR>
185  Sample<PAR>& Sample<PAR>::operator+=(PAR value) {
186      for (PAR& i : m_data) {
187          i += value;
188      }
189      return *this;
190  }
191
192  template <class PAR>
193  Sample<PAR>& Sample<PAR>::operator*=(PAR value) {
194      for (PAR& i : m_data) {
195          i *= value;
196      }
197      return *this;
198  }
199
200  template <class PAR>
201  Sample<PAR>& Sample<PAR>::operator/=(PAR value) {
202      for (PAR& i : m_data) {
203          if (std::abs(i) < std::abs(value)) {
204              throw std::invalid_argument("Division by zero");
205          }
206          i /= value;
207      }
208      return *this;
209  }
```

4) Продолжаем с точки, равной 31 с:

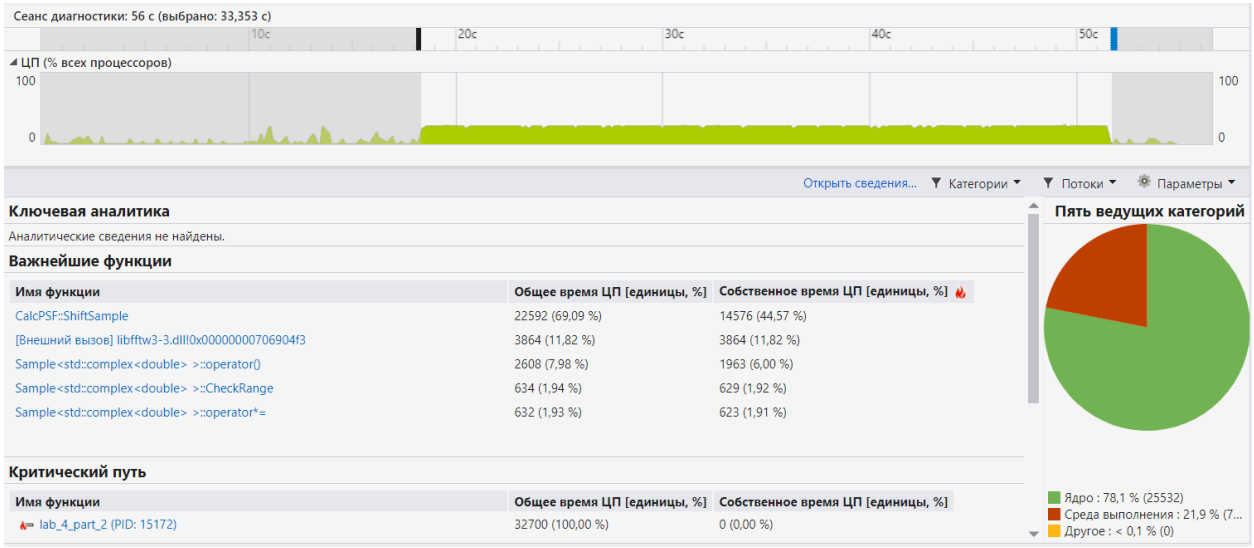


Видим, что в режиме сборке RelWithDebInfo отметилось неоптимальным: использование библиотеки fftw, на которую, к сожалению, мы никак не можем повлиять; а также операторы () и *=. Наиболее проблемные места отмечаются в красном диапазоне и процентными соотношениями:

```
                133  template <class PAR>
 96 (0,31%)     134  PAR& Sample<PAR>::operator()(int i, int j) {
552 (1,81%)     135      if (i < 0 || i >= m_size_x || j < 0 || j >= m_size_y) {
                136          throw std::out_of_range("Index out of range");
                137      }
389 (1,27%)     138      return m_data[i * m_size_y + j];
143 (0,47%)     139  }
```

Также отмечу, что проверка на выход за пределы размера выборки повторяется, поэтому попробую вынести ее в отдельный метод класса:

```
135  template <class PAR>
136  bool Sample<PAR>::CheckRange(int i, int j) const {
137      return (i >= 0 && i < m_size_x && j >= 0 && j < m_size_y);
138  }
139
140  template <class PAR>
141  PAR& Sample<PAR>::operator()(int i, int j) {
142      if (!CheckRange(i, j)) {
143          throw std::out_of_range("Index out of range");
144      }
145      return m_data[i * m_size_y + j];
146  }
147
148  template <class PAR>
149  const PAR& Sample<PAR>::operator()(int i, int j) const {
150      if (!CheckRange(i, j)) {
151          throw std::out_of_range("Index out of range");
152      }
153      return m_data[i * m_size_y + j];
154  }
```



Получили увеличение времени выполнения. И, как видим, только что написанная функция вносит вклад в потери скорости выполнения. Поэтому снова вернём всё назад, как было.

5) Рассмотрим функции через поле «Критический путь». Видим, что приличное количество времени тратится на вычисление ФРТ:

Критический путь

Имя функции	Общее время ЦП [единицы, %]	Собственное время ЦП [единицы, %]
lab_4_part_2 (PID: 15172)	32700 (100,00 %)	0 (0,00 %)
[Системный код] ntdll.dll!0x00007ffeabc22651	32700 (100,00 %)	50 (0,15 %)
__scrt_common_main_seh	32650 (99,85 %)	0 (0,00 %)
qtEntryPoint	32650 (99,85 %)	0 (0,00 %)
main	32650 (99,85 %)	0 (0,00 %)
[Системный код] qt6core.dll!0x00007ffe3d0bb52d	32650 (99,85 %)	14 (0,04 %)
DQtPSF::OnGetCalc	32636 (99,80 %)	0 (0,00 %)
CalcPSF::Calc	32634 (99,80 %)	235 (0,72 %)
CalcPSF::CalcFFT	15488 (47,36 %)	0 (0,00 %)
CalcPSF::ShiftSample	11315 (34,60 %)	7288 (22,29 %)

Имя функции	Общее время Ц...	Собственное вре...	Модуль	Категория
lab_4_part_2 (идентификатор процесса: 151...	32700 (100,00 %)	0 (0,00 %)	Несколько модул...	
[Внешний вызов] ntdll.dll!0x00007ffeabc2...	32700 (100,00 %)	50 (0,15 %)	ntdll	Ядро | Среда вып...
__scrt_common_main_seh	32650 (99,85 %)	0 (0,00 %)	lab_4_part_2	Среда выполнения
qtEntryPoint	32650 (99,85 %)	0 (0,00 %)	lab_4_part_2	Среда выполнения
main	32650 (99,85 %)	0 (0,00 %)	lab_4_part_2	Среда выполнения
[Внешний вызов] qt6core.dll!0x0...	32650 (99,85 %)	14 (0,04 %)	qt6core	Среда выполнения
DQtPSF::OnGetCalc	32636 (99,80 %)	0 (0,00 %)	lab_4_part_2	Среда выполнения
CalcPSF::Calc	32634 (99,80 %)	235 (0,72 %)	lab_4_part_2	Среда выполнения
CalcPSF::CalcFFT	15488 (47,36 %)	0 (0,00 %)	lab_4_part_2	Среда выполнения
CalcPSF::ShiftSample	11315 (34,60 %)	7288 (22,29 %)	lab_4_part_2	Среда выполнения
SampleComplex::GetIntensity	1705 (5,21 %)	330 (1,01 %)	lab_4_part_2	Среда выполнения
SampleComplex::ComplexToD...	1292 (3,95 %)	252 (0,77 %)	lab_4_part_2	Среда выполнения
CalcPSF::CalcPupilFunction	1208 (3,69 %)	335 (1,02 %)	lab_4_part_2	Среда выполнения
[Внешний вызов] ucrtbase.dll...	460 (1,41 %)	460 (1,41 %)	ucrtbase	Среда выполнения
Sample<std::complex<double...	304 (0,93 %)	299 (0,91 %)	lab_4_part_2	
[Системный код] 0xfffff80514...	293 (0,90 %)	293 (0,90 %)	[Неизвестный код]	
Sample<double>::operator=	160 (0,49 %)	0 (0,00 %)	lab_4_part_2	
Sample<double>::operator*=	155 (0,47 %)	153 (0,47 %)	lab_4_part_2	
[Системный код] 0xfffff80514...	14 (0,04 %)	14 (0,04 %)	[Неизвестный код]	
[Системный код] 0xfffff80514...	2 (0,01 %)	2 (0,01 %)	[Неизвестный код]	

Как я говорила и как видно на рисунке ниже, все трудозатраты функции – функции библиотеки *fftw*:

CalcPSF::CalcFFT | 15169 (48,58 %) | 1 (0,00 %) | lab_4_part_2 | Среда выполнения

D:\MAGISTRATURA\second_sem\c++\lab_4\part_2\calc_psf.cpp:32

```
                    32  void CalcPSF::CalcFFT(SampleComplex& Sample_psf) {
                    33      // Смещение выборки
                    34      ShiftSample(Sample_psf);
                    35
                    36      // Создаем план для fftw
10986 (35,18%)      37      fftw_plan oPlan = fftw_plan_dft_2d(Sample_psf.GetSize(), Sample_psf.GetSize(),
                    38          (fftw_complex*)Sample_psf.GetPointer(),
                    39          (fftw_complex*)Sample_psf.GetPointer(),
                    40          FFTW_BACKWARD, FFTW_ESTIMATE);
                    41
                    42      // Преобразование Фурье
   20 (0,06%)       43      fftw_execute(oPlan);
                    44
                    45      // Удаляем план
 3840 (12,30%)      46      fftw_destroy_plan(oPlan);
                    47
                    48      // Создание комплексного числа с вещественной 1/sqrt(n*n) и мнимой 0
    4 (0,01%)       49      std::complex<double> coeff(1. / sqrt(double(Sample_psf.GetSize() * Sample_psf.GetSize())), 0.);
                    50
                    51      // Домножение полученного спектра на коэффициент
                    52      Sample_psf *= coeff;
                    53
                    54      // Смещение спектра
  319 (1,02%)       55      ShiftSample(Sample_psf);
                    56  }
```

Попробуем оптимизировать метод, который делает сдвиг при преобразовании Фурье:

CalcPSF::ShiftSample | 11090 (35,52 %) | 7363 (23,58 %) | lab_4_part_2 | Среда выполнения

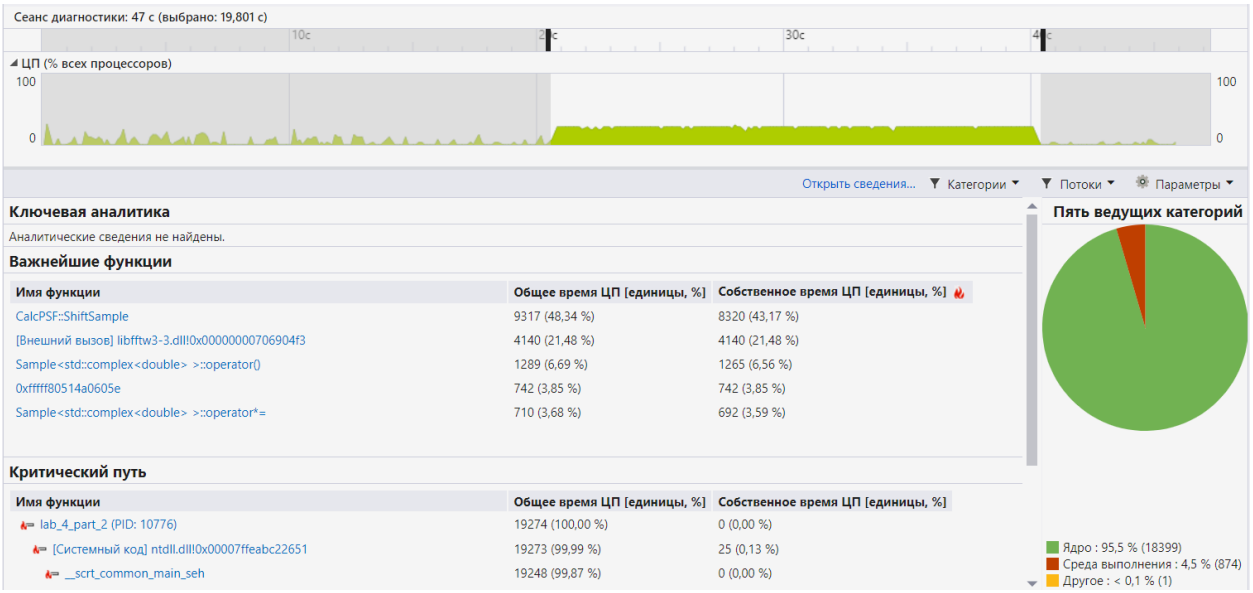
D:\MAGISTRATURA\second_sem\c++\lab_4\part_2\calc_psf.cpp:96

```
                    94  }
                    95
                    96  void CalcPSF::ShiftSample(SampleComplex& Sample_psf) {
                    97      double pi = 2 * acos(0.);
                    98
                    99      for (int j = 0; j < Sample_psf.GetSize(); j++) {
 303 (0,97%)       100          for (int i = 0; i < Sample_psf.GetSize(); i++) {
                   101              // Вычисление сдвигового коэффициента по x
1659 (5,31%)       102              std::complex<double> coeff = std::complex<double>(cos(-2. * pi / 2 * i), sin(-2. * pi / 2 * i));
                   103
                   104              // Домножение на сдвиговый коэффициент по x
8599 (27,54%)      105              Sample_psf(i, j) *= coeff;
                   106
                   107              // Вычисление сдвигового коэффициента по y
   5 (0,02%)       108              coeff = std::complex<double>(cos(-2. * pi / 2 * j), sin(-2. * pi / 2 * j));
                   109
                   110              // Домножение на сдвиговой коэффициент по y
 524 (1,68%)       111              Sample_psf(i, j) *= coeff;
                   112          }
                   113      }
                   114  }
```

Сверху представлен метод до правок, снизу – после:

```
 96  void CalcPSF::ShiftSample(SampleComplex& Sample_psf) {
 97      double pi = 2 * acos(0.);
 98
 99      // Предвычисление сдвиговых коэффициентов
100      std::vector<std::complex<double>> coeff_x(Sample_psf.GetSize());
101      std::vector<std::complex<double>> coeff_y(Sample_psf.GetSize());
102      for (int i = 0; i < Sample_psf.GetSize(); i++) {
103          coeff_x[i] = std::complex<double>(cos(-2. * pi / 2 * i), sin(-2. * pi / 2 * i));
104          coeff_y[i] = std::complex<double>(cos(-2. * pi / 2 * i), sin(-2. * pi / 2 * i));
105      }
106
107      for (int j = 0; j < Sample_psf.GetSize(); j++) {
108          for (int i = 0; i < Sample_psf.GetSize(); i++) {
109              // Домножение на сдвиговой коэффициент по x
110              Sample_psf(i, j) *= coeff_x[i];
111
112              // Домножение на сдвиговой коэффициент по y
113              Sample_psf(i, j) *= coeff_y[j];
114          }
115      }
116  }
```

Получили супер-результат! Время выполнения составляет 19.801 с.



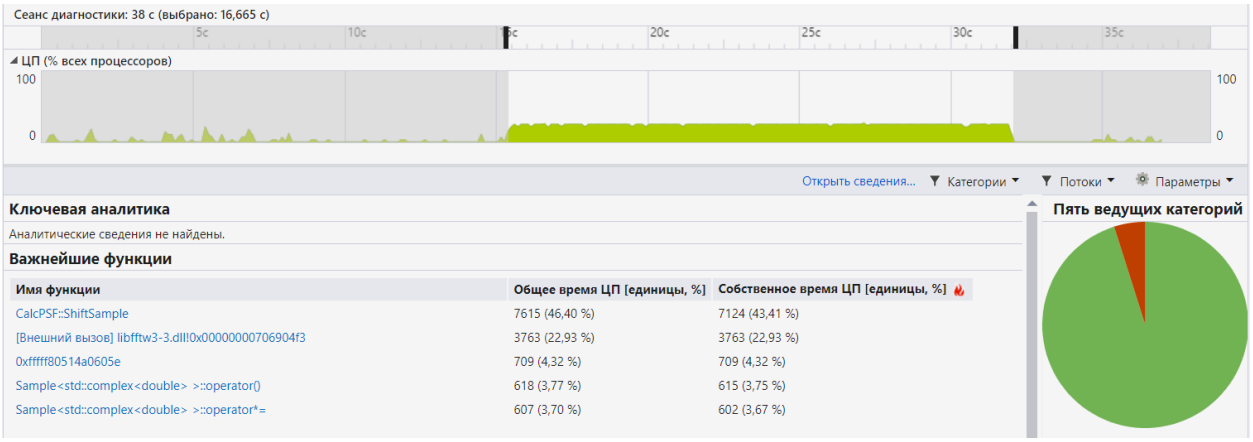
Теперь проблемные места выглядят «чуть приятнее»:

CalcPSF::ShiftSample | 3833 (23,35 %) | 3572 (21,76 %) | lab_4_part_2 | Среда в

MAGISTRATURA\second_sem\c++\lab_4\part_2\calc_psf.cpp:96

```
                95
                96  void CalcPSF::ShiftSample(SampleComplex& Sample_psf) {
                97      double pi = 2 * acos(0.);
                98
                99      // Предвычисление сдвиговых коэффициентов
               100      std::vector<std::complex<double>> coeff_x(Sample_psf.GetSize());
     1 (0,01%) 101      std::vector<std::complex<double>> coeff_y(Sample_psf.GetSize());
               102      for (int i = 0; i < Sample_psf.GetSize(); i++) {
     2 (0,01%) 103          coeff_x[i] = std::complex<double>(cos(-2. * pi / 2 * i), sin(-2. * pi / 2 * i));
     6 (0,04%) 104          coeff_y[i] = std::complex<double>(cos(-2. * pi / 2 * i), sin(-2. * pi / 2 * i));
               105      }
               106
     1 (0,01%) 107      for (int j = 0; j < Sample_psf.GetSize(); j++) {
   118 (0,72%) 108          for (int i = 0; i < Sample_psf.GetSize(); i++) {
               109              // Домножение на сдвиговой коэффициент по x
3155 (19,22%) 110              Sample_psf(i, j) *= coeff_x[i];
               111
               112              // Домножение на сдвиговой коэффициент по y
   550 (3,35%) 113              Sample_psf(i, j) *= coeff_y[j];
               114          }
               115      }
               116  }
```

6*. Попробуем немного изменить оператор (), возложив все надежды на то, что проверка на корректность не потребуется – удалось уменьшить время выполнения еще примерно на 1.5 секунды:



Результаты:

В ходе выполнения работы были рассмотрены базовые методы профилирования, в результате применения которых время выполнения вычислений ФРТ в размере 100 штук удалось уменьшить с 33.312 с до 19.801 (и до 16.655* (см. п. 6)).