Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Параллельные численные методы»

Лабораторная работа №5 Фильтрация изображений. Быстрое преобразование Фурье

Сиднев А.А.

При поддержке компании Intel

Нижний Новгород
2010

Содержание

ВВЕДЕНИЕ **4**

1. МЕТОДИЧЕСКИЕ УКАЗАНИЯ **4**

1.1. Цели и задачи работы 4
1.2. Структура работы 5
1.3. Тестовая инфраструктура 5
1.4. Рекомендации по проведению занятий 6

2. ПРЕОБРАЗОВАНИЕ ФУРЬЕ **7**

2.1. Общие сведения 7
2.2. Назначение и примеры использования 11
2.3. Алгоритм Кули и Таки 14

3. РЕАЛИЗАЦИЯ ПРОГРАММЫ, ВЫПОЛНЯЮЩЕЙ ФИЛЬТРАЦИЮ ВИДЕО **23**

3.1. Общая структура 23
3.2. Частотная фильтрация 28
3.3. Библиотека Standard Template Library 30
3.4. Библиотека OpenCV 30

4. ПОИСК ОШИБОК В ПРОЕКТЕ **31**

5. ОТЛАДКА ПРОГРАММЫ **33**

6. ОЦЕНКА ЭФФЕКТИВНОСТИ ПРОГРАММЫ **37**

7. ОПТИМИЗАЦИЯ. КОНТЕЙНЕРЫ STL **45**

8. ОПТИМИЗАЦИЯ. ВОЗМОЖНОСТИ КОМПИЛЯТОРА **49**

9. ОПТИМИЗАЦИЯ. ЭФФЕКТИВНОЕ ВЫЧИСЛЕНИЕ ТРИГОНОМЕТРИЧЕСКИХ ФУНКЦИЙ **53**

9.1. Вещественная арифметика 53
9.2. Эффективное вычисление тригонометрических функций 56

10. ОПТИМИЗАЦИЯ. ИЗБАВЛЕНИЕ ОТ ЛИШНИХ ВЕТВЛЕНИЙ **62**

11. ОПТИМИЗАЦИЯ. ИТЕРАТИВНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА **66**

12. ОПТИМИЗАЦИЯ. ИСПОЛЬЗОВАНИЕ ВСТРОЕННЫХ ТИПОВ ДАННЫХ **74**

13. ОПТИМИЗАЦИЯ. ПРЕДВЫЧИСЛЕНИЯ **76**

14. ОПТИМИЗАЦИЯ. УПРОЩЕНИЕ ВЫРАЖЕНИЙ 81

15. РАСПАРАЛЛЕЛИВАНИЕ. РАЗРАБОТКА ПРОСТЕЙШЕЙ OPENMP-ВЕРСИИ ПРОГРАММЫ 84

15.1. Гонки данных 93

16. РАСПАРАЛЛЕЛИВАНИЕ. ЭФФЕКТИВНАЯ OPENMP-РЕАЛИЗАЦИЯ ПРОГРАММЫ 94

17. ОПТИМИЗАЦИЯ. ЭФФЕКТИВНОЕ ИСПОЛЬЗОВАНИЕ КЕШ-ПАМЯТИ 100

18. БИБЛИОТЕКИ ДЛЯ ВЫЧИСЛЕНИЯ БПФ 110

19. ДОПОЛНИТЕЛЬНЫЕ ЗАДАНИЯ 115

20. ЛИТЕРАТУРА 116

20.1. Использованные источники информации 116
20.2. Дополнительная литература 116
20.3. Информационные ресурсы сети Интернет 117

21. ПРИЛОЖЕНИЯ 117

21.1. Исходная версия программы 118
21.2. Корректная версия программы 132
21.3. Финальная версия программы 147

Введение

Повышение качества изображения для восприятия человеком, создание спецэффектов, обработка изображений (выделение объектов, поиск лиц, улыбок) это задачи, которые решаются с помощью фильтрации изображений. Наиболее широкий спектр эффектов достигается за счёт использования частотных фильтров. В лабораторной работе подробно разбирается алгоритм быстрого преобразования Фурье, который является основой частотной фильтрации. На примере задачи частотной фильтрации видео рассматриваются наиболее типичные ошибки разработки последовательных и параллельных программ, а также подходы к их оптимизации. На протяжении всей работы используется инструмент Intel Parallel Studio XE 2011 для поиска ошибок, анализа эффективности, оптимизации и распараллеливания.

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – изучение подходов к оптимизации и принципов построения корректных последовательных и параллельных реализаций быстрого преобразования Фурье на примере задачи частотной фильтрации изображения с использованием пакета Intel Parallel Studio XE 2011.

Данная цель предполагает решение следующих основных задач:

1. Изучение общей идеи преобразования Фурье и алгоритма быстрого преобразования Фурье для решения задачи частотной фильтрации изображений.
2. Выполнение корректной программной реализации алгоритма быстрого преобразования Фурье для решения задачи частотной фильтрации изображений в последовательном и параллельном случаях.
3. Изучение и применение подходов к оптимизации на примере задачи частотной фильтрации изображений.
4. Освоение способов использования математической библиотеки Intel MKL для вычисления преобразования Фурье.

1.2. Структура работы

Работа построена следующим образом: дается краткая информация о дискретном и непрерывном преобразованиях Фурье, алгоритме быстрого преобразования Фурье, библиотеках STL, OpenCV и частотной фильтрации изображений. Рассматриваются типовые ошибки и способы их обнаружения, которые можно допустить при разработке последовательной программы. Выполняется последовательное применение и анализ эффективности различных подходов к оптимизации программного кода. Рассматриваются проблемы организации параллельных вычислений, иллюстрируются характерные ошибки, приводится корректная параллельная реализация. Проводится обзор использования библиотеки MKL для вычисления быстрого преобразования Фурье.

Лабораторная работа основана на задаче частотной фильтрации изображений [3]. С помощью фильтрации достигается улучшение качества изображения как для восприятия человеком, так и для упрощения обработки этого изображения компьютером. Частотная фильтрация изображения является очень мощным механизмом фильтрации, позволяющим реализовать большое количество эффектов: выделение границ объектов, размытие изображения и др. Частотная фильтрация применяется к спектру изображения, поэтому для её выполнения необходимо использовать преобразование Фурье. В лабораторной работе подробно разбирается алгоритм быстрого преобразования Фурье, программный код которого вынесен в отдельный модуль.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (Таблица 1).

Таблица 1. Тестовая инфраструктура

Процессор	2 четырехъядерных процессора Intel Xeon E5520 (2.27 GHz)
Память	16 Gb
Операционная система	Microsoft Windows 7
Среда разработки	Microsoft Visual Studio 2008: • Version 9.0.21022.8 • Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Компилятор, профилиров-	Intel Parallel Studio XE 2011:

щик, отладчик, математическая библиотека	• Intel Composer XE 2011 (package 104): • Intel(R) C++ Compiler XE for applications running on IA-32, Version 12.0.0.104 • Intel MKL v. 10.3.0.104 • Intel Inspector XE 2011 (build 119192) • Intel VTune Amplifier XE 2011 (build 119041)

1.4. Рекомендации по проведению занятий

Для выполнения лабораторной работы рекомендуется следующая последовательность действий.

1. Кратко рассказать студентам о применении преобразования Фурье на практике в задачах цифровой обработки сигналов.
2. Рассмотреть общую идею алгоритма быстрого преобразования Фурье для решения задачи частотной фильтрации изображений.
3. Рассмотреть общую идею частотной фильтрации изображений с использованием быстрого преобразования Фурье.
4. Изучить реализацию стартового проекта, выполняющего фильтрацию видео. Дать краткую информацию о библиотеках STL, OpenCV и потоках Windows.
5. Разобрать и исправить типовые ошибки, присутствующие в стартовом проекте. Провести вычислительные эксперименты.
6. Выполнить рассмотренные оптимизации последовательной версии приложения с использованием Intel Parallel Studio XE 2011. Провести вычислительные эксперименты.
7. Выполнить этапы разработки параллельной версии приложения. Продемонстрировать возможные ошибки и их влияние на результат работы.
8. Использовать в проекте преобразование Фурье из библиотеки Intel MKL.

2. Преобразование Фурье

2.1. Общие сведения

Наиболее распространённым и удобным способом задания функции является явное задание вида $y = f(x)$. Такие функции можно наглядно представлять в виде графика, оси абсцисс которого соответствуют значения аргумента, а оси ординат – значения функции. В цифровой обработке сигналов такое представление часто называют временны́м ($time\ domain$), когда по оси абсцисс откладывается время (пример такой функции показан на Рис. 1 слева), а по оси ординат – амплитуда сигнала (например, звуковой сигнал).

Однако есть альтернативный способ представления функции, предложенный Жаном Фурье, который называется частотным. В частотном пространстве ($frequency\ domain$) функция времени отображается несколько иначе за счет того, что по оси абсцисс откладывается частота, а по оси ординат – амплитуда гармоник, составляющих функцию. Таким образом, получается разложение исходной функции времени на гармонические составляющие (синусоиды с различными амплитудами и частотами). На Рис. 1 представлена функция (слева), которая является суммой трех функций (справа).

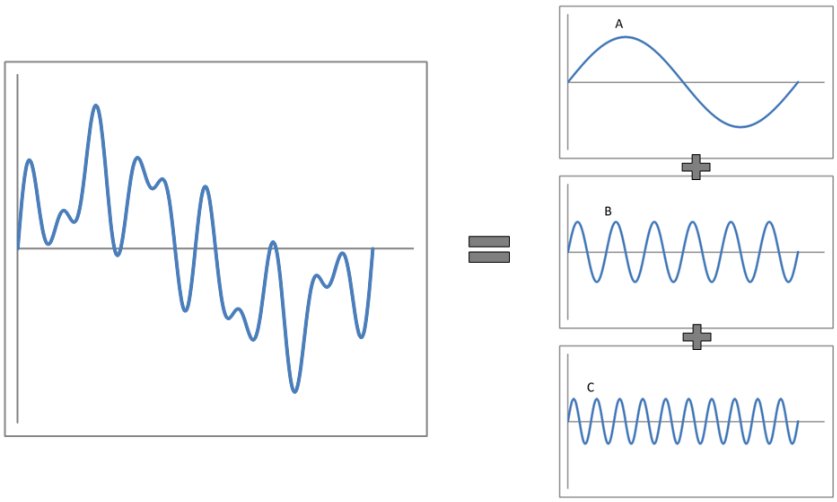


Рис. 1. Разложение функции на гармоники

Представление функции в частотной области называют спектром функции. Примеры спектров функций показаны на Рис. 2: для каждой синусоиды (слева) представлен соответствующий спектр (справа), который во всех трех случаях является одной точкой с координатами, соответствующими амплитуде и частоте синусоиды. Для перевода сигнала из временно́го пространства в частотное используется преобразование Фурье (ПФ). Для вос-

становления функции из частотного пространства используется обратное преобразование Фурье.

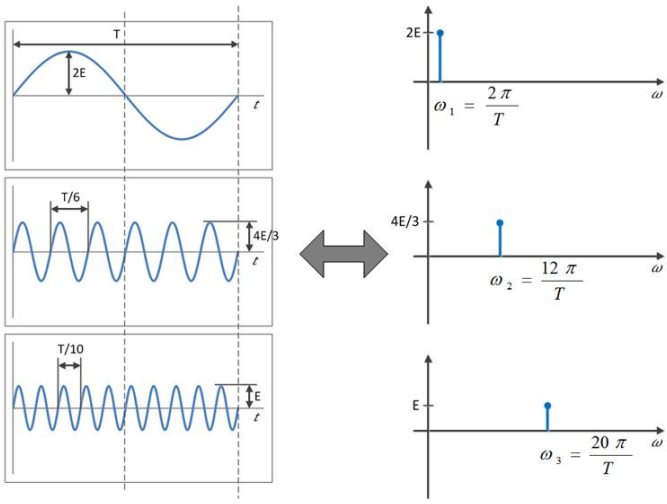


Рис. 2. Спектр гармоник

Одна из важнейших особенностей преобразования Фурье заключается в том, что спектр суммарной функции времени равен сумме спектров ее гармонических составляющих. Спектр функции, представленной на Рис. 1, – это три точки, соответствующие амплитуде и частоте этих гармоник (Рис. 3).

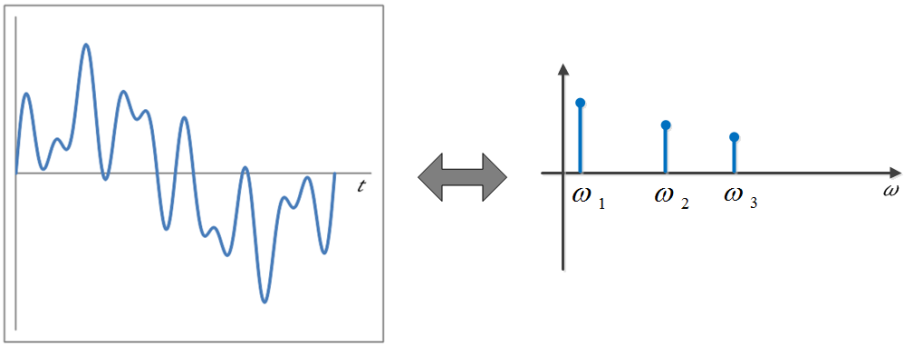


Рис. 3. Спектр функции

Непрерывное преобразование Фурье применяется к функции $x(t)$, заданной на всей числовой оси, т.е. на интервале (-∞, +∞). В результате преобразования получается функция-спектр $y(\omega)$:

$$y(\omega) = \int_{-\infty}^{+\infty} x(t) \cdot e^{-\omega \cdot t \cdot i} dt \qquad (1)$$

Обратное преобразование предназначено для восстановления функции по её спектру. Таким образом, если к фурье-образу $y(\omega)$ применить обратное преобразование Фурье, то получится исходная функция $x(t)$:

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} y(\omega) \cdot e^{\omega \cdot t \cdot i} d\omega \tag{2}$$

Функция $x(t)$ и фурье-образ $y(\omega)$ могут принимать комплексные значения[1], т.е. являются комплексными функциями действительного аргумента.

В цифровой технике работать с непрерывными функциями неудобно, поэтому функции часто задаются в виде набора значений на фиксированной сетке. Преобразование Фурье может использоваться как для непрерывных, так и для дискретных функций. В последнем случае оно называется дискретным преобразованием Фурье – ДПФ (в этом случае предполагается, что за пределами сетки функция периодически повторяется).

Рассмотрим функцию $x_\sigma(t)$, которая дискретизирует исходную функцию $x(t)$ с помощью дельта-функции:

$$x_\sigma(t) = x(t) \cdot \sum_{n=0}^{N-1} \sigma(t - n \cdot \Delta\, t) = \sum_{n=0}^{N-1} x(t) \cdot \sigma(t - n \cdot \Delta\, t) \tag{3}$$

Напомним, что дельта-функция $\sigma(t)$ – функция, которая во всех точках равна нулю, кроме одной, в которой она принимает значение бесконечности:

$$\begin{gathered} \sigma(t) = \infty, \text{если } t = 0 \\ \sigma(t) = 0, \text{если } t \neq 0 \end{gathered} \tag{4}$$

Применим преобразование Фурье к функции $x_\sigma(t)$:

$$y(\omega) = \int_{-\infty}^{\infty} x_\sigma(t) \cdot e^{-\omega \cdot t \cdot i}\, dt = \int_{-\infty}^{\infty} \sum_{n=0}^{N-1} x(t) \cdot \sigma(t - n \cdot \Delta\, t) \cdot e^{-\omega \cdot t \cdot i}\, dt \tag{5}$$

Далее вспомним фильтрующее свойство дельта-функции [1]:

$$\int_{-\infty}^{\infty} x(t) \cdot \sigma(t - \tau)\, dt = x(\tau) \tag{6}$$

---

[1] Комплексное число представляет собой упорядоченную пару действительных чисел: (x, y), где $x,\ y$ – вещественные числа. Для представления комплексных чисел часто используют аналитическую ($z = x + i \cdot y$, где i – мнимая единица ($i^2 = -1$)), тригонометрическую ($z = r \cdot (cos\varphi + i \cdot sin\varphi)$, где $r = |z|$, $cos\varphi = \frac{x}{|z|}$, $sin\varphi = \frac{y}{|z|}$) или показательную форму записи ($z = r \cdot e^{i \cdot \varphi} = r \cdot (cos\varphi + i \cdot sin\varphi)$).

Используя формулу (6), вычислим спектр дискретизированного сигнала:

$$y(\omega) = \sum_{n=0}^{N-1} \int_{-\infty}^{\infty} x(t) \cdot \sigma(t - n \cdot \Delta\, t) \cdot e^{-\omega \cdot t \cdot i}\, dt = \sum_{n=0}^{N-1} x(n \cdot \Delta\, t) \cdot e^{-\omega \cdot n \cdot \Delta t \cdot i} \tag{7}$$

Выражение (7) задаёт формулу вычисления спектра дискретной функции с шагом дискретизации $\Delta\, t$. При этом спектр $y(\omega)$ получается непрерывной, а не дискретной функцией, что очень неудобно в цифровой обработке. Первое, что нужно отметить, – это то, что можно рассматривать только один период повторения спектра $y(\omega)$ при $\omega \in [0, \frac{2\pi}{\Delta t}]$, т.к. экспонента комплексного аргумента является периодической функцией с периодом $\frac{2\pi n}{\Delta t}$, а величина $\frac{2\pi}{\Delta t}$ является максимальным периодом повторения [9].

ДПФ ставит в соответствие N отсчётам дискретного сигнала, N отсчётов дискретного спектра, при этом предполагается, что и сигнал, и спектр являются периодическими и анализируются на одном периоде.

Напомним, что $x(t)$ и $y(\omega)$ – комплексные функции действительного аргумента. Зададим следующие обозначения:

$$x_n = x(n \cdot \Delta\, t),\ n = \overline{0, N-1}$$
$$y_n = y(n \cdot \Delta\, \omega),\ n = \overline{0, N-1} \tag{8}$$

Используя формулу (7) и применяя обозначение (8) получаем выражение для прямого преобразования Фурье:

$$y_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{k \cdot n}{N} \cdot 2 \cdot \pi \cdot i}, \text{где } k = \overline{0, N-1} \tag{9}$$

Обратное преобразование в таком случае будет иметь вид:

$$x_k = \frac{1}{N} \cdot \sum_{n=0}^{N-1} y_n \cdot e^{\frac{k \cdot n}{N} \cdot 2 \cdot \pi \cdot i}, \text{где } k = \overline{0, N-1} \tag{10}$$

ДПФ можно рассматривать как аппроксимацию непрерывного ПФ (формулы (1), (9) и (2), (10)), по аналогии с тем, как интеграл аппроксимируется интегральной суммой. Определенное таким образом дискретное преобразование Фурье сохраняет практически все свойства непрерывного (разумеется, с учетом перехода к дискретному множеству).

Популярность ДПФ в задачах цифровой обработки сигналов очень высока, поэтому его вычисление реализуют в виде специализированных микропроцессоров (Рис. 4). Специализированные процессоры, позволяющие решать задачи цифровой обработки сигналов обычно называют DSP процессорами ($Digital\ Signal\ Processor$). Производительность таких процессоров значительно выше, чем процессоров общего назначения.

Рис. 4. DSP FM680[2] на ПЛИС Xilinx Virtex-6

Представленные формулы (9) и (10) требуют для вычисления дискретного преобразования Фурье значительных затрат. Трудоемкость таких алгоритмов имеет порядок $O(n^2)$. На практике вычисление ДПФ чаще всего происходит в виде быстрого преобразования Фурье (БПФ). В таком виде оно реализовано во многих математических библиотеках, например, Intel MKL, Intel IPP, FFTW. БПФ – это простой алгоритм для вычисления ДПФ с трудоемкостью $O(nlogn)$. В настоящий момент существует несколько алгоритмов быстрого преобразования Фурье. В данной работе рассматривается одна из наиболее популярных реализаций алгоритма, предложенная Кули и Таки (Cooley-Tukey).

2.2. Назначение и примеры использования

Дискретное преобразование Фурье является одним из основных алгоритмов в цифровой обработке сигналов. Далее рассмотрим два примера использования ПФ: обработка звука и обработка изображений.

2.2.1. Обработка звука[3]

Одним из первых форматов файла-контейнера для хранения записи оцифрованного аудиопотока является формат WAV. Этот формат чаще всего

[2] Изображение взято с сайта [10].

[3] По материалам [2].

используется в качестве оболочки для несжатого звука (PCM, Pulse Code Modulation). При таком кодировании в WAV-файле хранится полная информация об исходном звуке, оцифрованном и проквантованном с некоторой частотой, например 44 кГц. Этой информации абсолютно достаточно для воспроизведения всех частот исходного сигнала, меньших половины частоты квантования (по т. Котельникова). При этом одна минута записи занимает около 15 мегабайт, поэтому в настоящее время очень популярным стал формат кодирования MPEG Layer-3 со сжатием.

При сжатии кодеком MPEG Layer-3 исходный звуковой сигнал разбивается на фрагменты, длительностью по 50 миллисекунд, каждый из которых анализируется отдельно. При анализе фрагмент раскладывается на гармоники по методу Фурье, из которых в соответствии с теорией восприятия звука человеческим ухом выбрасываются те гармоники, которые человек хуже воспринимает на фоне остальных:

- более тихие гармоники на фоне более громких;
- звуки, замаскированные вследствие инертности слуха (так, например, если за очень громким хлопком, с задержкой в долю секунды, пойдет какой-то иной кратковременный сигнал, то его слышно не будет).

При воспроизведении файла закодированного MPEG Layer-3 производится обратное преобразование, при котором оставшиеся гармоники вновь преобразуются в звуковую волну. Поскольку часть информации об исходном сигнале была удалена, звук получается не совпадающий с исходным. Одной из основных характеристик качества аудио-потока является *битрейт*.

Битрейт – это объем информации на каждую секунду записи:

- чем он меньше, тем меньший размер имеют файлы с одинаковой по времени длине;
- чем он меньше, тем большее количество «лишних» гармоник приходится удалять.

2.2.2. Обработка изображений[4]

Все фильтры делят на две группы: пространственные и частотные. Типов пространственных фильтров достаточно много: линейные, медианные, изотропные и др. Процесс фильтрации в этих фильтрах основан на простом применении маски фильтра ко всем точкам изображения. В каждой точке отклик фильтра вычисляется с использованием предварительно заданных связей. Все пространственные фильтры имеют линейную трудоёмкость.

[4] По материалам [3].

Частотная фильтрация является более общим механизмом фильтрации, чем пространственная. Все пространственные фильтры можно реализовать с помощью частотных. Частотные фильтры выполняют фильтрацию изображений в частотной области. Для этого с помощью преобразования Фурье происходит получение спектра изображения. После применения фильтра к спектру изображения выполняется обратное преобразование Фурье и получается отфильтрованное изображение.

Простота фильтрации в частотной области заключается в простой интерпретации частот:

- низкие частоты – плавное изменение яркости изображения;
- высокие частоты – быстрое изменение яркости изображения (границы объектов).

За счёт изменения значений определённых частот можно добиться размытия изображения, повышение резкости и др.

На Рис. 5 представлено изображение кластерной стойки (слева) и Фурье-спектра этого изображения (справа).

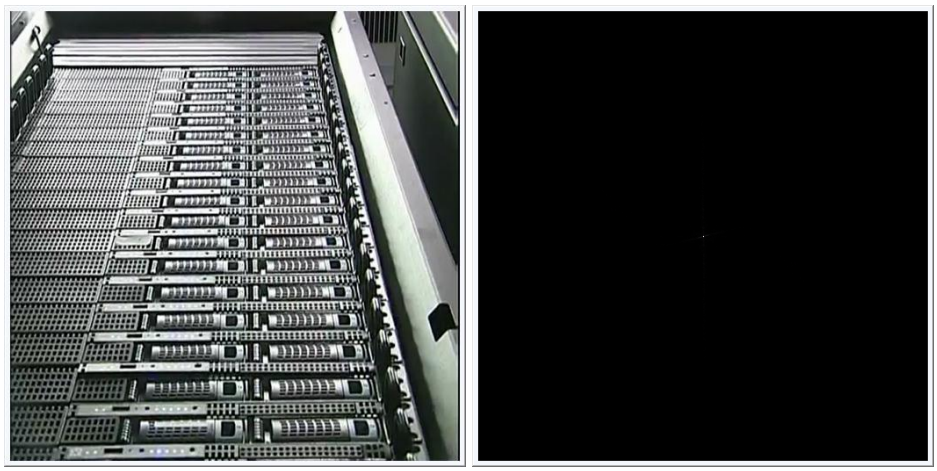


Рис. 5. Изображение и его спектр

Спектр изображения содержит большую часть информации в центре картинки (белая точка в центре), поэтому для более наглядной визуализации спектра будем выводить логарифмированный спектр (Рис. 6). Логарифмический спектр получается за счёт применения функции логарифма к каждой точке спектра и последующей нормировки полученных значений (если какая-либо точка спектра имела нулевое значение, то на логарифмическом спектре соответствующая точка будет принимать значение 0).

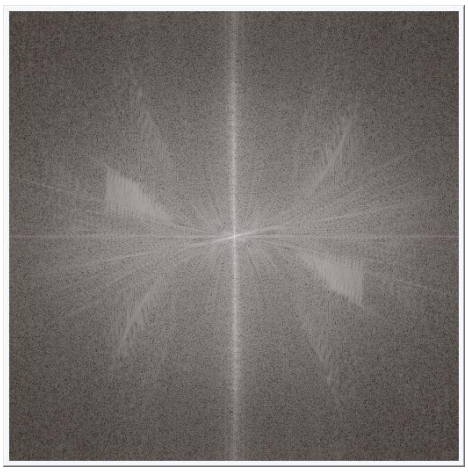


Рис. 6. Логарифмический спектр

2.3. Алгоритм Кули и Таки

Идея быстрого преобразования Фурье, предложенного Кули и Таки, состоит в сокращении числа умножений и сложений, выполняемых в исходном ДПФ. При вычислении ДПФ приходится многократно считать множители $e^{-\frac{k\cdot n}{N}\cdot 2\cdot\pi\cdot i}$. Нетрудно заметить, что эти множители при всех k и n $(k,\ n=\overline{0,N-1})$, удовлетворяющих выражению $(k\cdot n)\ mod\ N = const$ равны. Понятно, что таких k и n достаточно много. Например, выражение $(k\cdot n)\ mod\ N = 0$ верно при $k=0$ и любом $n=0,\dots,N-1$ или при $n=0$ и любом $k=0,\dots,N-1$. За счет использования этого факта и происходит экономия числа арифметических операций.

2.3.1. Общая идея

Идея алгоритма Кули и Таки состоит в построение рекуррентного соотношения для вычисления ДПФ. Исходное множество точек разбивается на два равных подмножества (будем считать, что размер входных данных является степенью двойки). Над каждым из полученных множеств выполняется ДПФ, после чего множества объединяются за линейное время. При таком разбиении трудоёмкость вычисления ДПФ для половины точек составит $\frac{N^2}{4}$ операций, а трудоёмкость всего алгоритма – $\frac{N^2}{2}=\frac{N^2}{4}+\frac{N^2}{4}$ вместо N^2. Для вычисления ДПФ над каждым из подмножеств снова выполняется деление на 2 равные части. Такое разбиение продолжается до тех пор, пока количество элементов, над которыми необходимо выполнить ДПФ, не будет равно 1. После этого выполняется объединение всех промежуточных результатов (Рис. 7). Обработку подмножеств, получаемых после разбие-

ния можно выполнять независимо, этот факт будет использован в дальнейшем при реализации параллельной версии.

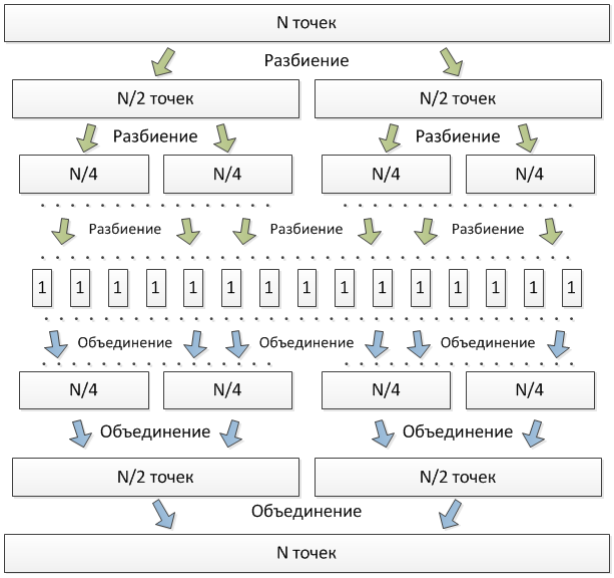


Рис. 7. Идея разбиения и объединения в алгоритме Кули и Таки

Рассмотрим более строгое обоснование алгоритма и получим формулу, позволяющую выполнять объединение результатов двух ДПФ за линейное время.

Для простоты, будем предполагать, что количество точек дискретизации N является степенью двойки. Алгоритм, рассчитанный на общий случай, можно найти в [4]. По предположению, N делится пополам нацело, поэтому можно разбить сумму, которая вычисляет k-ое слагаемое ДПФ, на две суммы, содержащие четные и нечётные слагаемые:

$$y_k = \sum_{n=0}^{N/2-1} x_{2\cdot n} \cdot e^{-\frac{2\cdot n\cdot k}{N}\cdot 2\cdot \pi\cdot i} + \sum_{n=0}^{N/2-1} x_{2\cdot n+1} \cdot e^{-\frac{(2\cdot n+1)\cdot k}{N}\cdot 2\cdot \pi\cdot i} \qquad (11)$$

Количество слагаемых в обеих суммах одинаково. Вынесем из второй суммы множитель $e^{-\frac{k}{N}\cdot 2\cdot \pi\cdot i}$ и приведём степени экспонент к более удобному виду:

$$y_k = \sum_{n=0}^{N/2-1} x_{2\cdot n} \cdot e^{-\frac{n\cdot k}{N/2}\cdot 2\cdot \pi\cdot i} + \left(\sum_{n=0}^{N/2-1} x_{2\cdot n+1} \cdot e^{-\frac{n\cdot k}{N/2}\cdot 2\cdot \pi\cdot i}\right) \cdot e^{-\frac{k}{N}\cdot 2\cdot \pi\cdot i} \qquad (12)$$

Первая сумма в формуле (12) представляет собой k-ый спектральный отсчёт ДПФ для чётных слагаемых, вторая сумма – k-ый спектральный отсчёт ДПФ для нечётных слагаемых. Множитель $e^{-\frac{k}{N}\cdot 2\cdot \pi\cdot i}$ обычно называют

коэффициентом поворота. Запишем выражение для $k+\frac{N}{2}$ спектрального отсчёта исходного ДПФ:

$$y_{k+\frac{N}{2}} = \sum_{n=0}^{\frac{N}{2}-1} x_{2\cdot n}\cdot e^{-\frac{n\cdot\left(k+\frac{N}{2}\right)}{\frac{N}{2}}\cdot 2\cdot\pi\cdot i} + \left(\sum_{n=0}^{N/2-1} x_{2\cdot n+1}\cdot e^{-\frac{n\cdot(k+\frac{N}{2})}{N/2}\cdot 2\cdot\pi\cdot i}\right)\cdot e^{-\frac{(k+\frac{N}{2})}{N}\cdot 2\cdot\pi\cdot i}$$

$$= \sum_{n=0}^{N/2-1} x_{2\cdot n}\cdot e^{-\frac{n\cdot k}{N/2}\cdot 2\cdot\pi\cdot i}\cdot e^{-n\cdot 2\cdot\pi\cdot i} + \left(\sum_{n=0}^{N/2-1} x_{2\cdot n+1}\cdot e^{-\frac{n\cdot k}{N/2}\cdot 2\cdot\pi\cdot i}\cdot e^{-n\cdot 2\cdot\pi\cdot i}\right)\cdot e^{-\frac{k}{N}\cdot 2\cdot\pi\cdot i}\cdot e^{-\pi\cdot i} \quad (13)$$

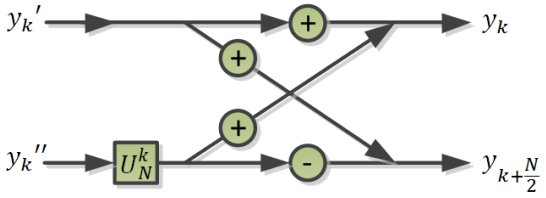
Упростим (13) учитывая, $e^{-n\cdot 2\cdot\pi\cdot i}=1$, а $e^{-\pi\cdot i}=-1$:

$$y_{k+\frac{N}{2}} = \sum_{n=0}^{N/2-1} x_{2\cdot n}\cdot e^{-\frac{n\cdot k}{N/2}\cdot 2\cdot\pi\cdot i} - \left(\sum_{n=0}^{N/2-1} x_{2\cdot n+1}\cdot e^{-\frac{n\cdot k}{N/2}\cdot 2\cdot\pi\cdot i}\right)\cdot e^{-\frac{k}{N}\cdot 2\cdot\pi\cdot i} \quad (14)$$

Формулы (12) и (14) задают соотношение для вычисления отсчётов дискретного спектра. Запишем это соотношение более компактно:

$$y_k = y_k' + y_k''\cdot e^{-\frac{2\cdot\pi\cdot k\cdot i}{N}}, k=\overline{0,\frac{N}{2}-1}$$
$$y_{k+\frac{N}{2}} = y_k' - y_k''\cdot e^{-\frac{2\cdot\pi\cdot k\cdot i}{N}}, k=\overline{0,\frac{N}{2}-1} \quad (15)$$

Величины $y_0', y_1', \dots, y_{\frac{N}{2}-1}'$ являются коэффициентами ДПФ для чётных слагаемых, а $y_0'', y_1'', \dots, y_{\frac{N}{2}-1}''$ – для нечётных. Соотношение (15) обычно визуализируют, как показано на Рис. 8, поэтому его часто называют «бабочкой». Итак, для вычисления ДПФ над N элементами по соотношению «бабочки» необходимо вычислить два ДПФ размерности $\frac{N}{2}$. Величину $\frac{N}{2}$ будем называть шагом «бабочки». Очевидно, что по соотношению (15) можно выполнить объединение результатов вычисления двух ДПФ за линейное время.



$$U_N^k = e^{-\frac{2\cdot\pi\cdot k\cdot i}{N}}$$

Рис. 8. «Бабочка»

Идея алгоритма БПФ заключалась в рекурсивном разбиении множества точек на две части до тех пор, пока количество элементов не станет равным 1. Далее выполнялось фиктивное ДПФ над 1 элементом, т.к. ДПФ над одним элементом x является величиной x (см. формулу (9)). После этого выполняется попарное объединение результатов выполнения ДПФ, начиная с пар по одному элементу. Трудоемкость такого алгоритма составляется $O(nlogn)$ за счёт того, что разбиение имеет логарифмическую глубину, а объединение результатов ДПФ линейно (Рис. 7).

Рассмотрим алгоритм вычисления БПФ на примере. Пусть на вход подается массив из четырёх элементов x_0, x_1, x_2, x_3. Вычисления проиллюстрированы на рисунке ниже, $U_N^k = e^{-\frac{2\cdot\pi\cdot k\cdot i}{N}}$.

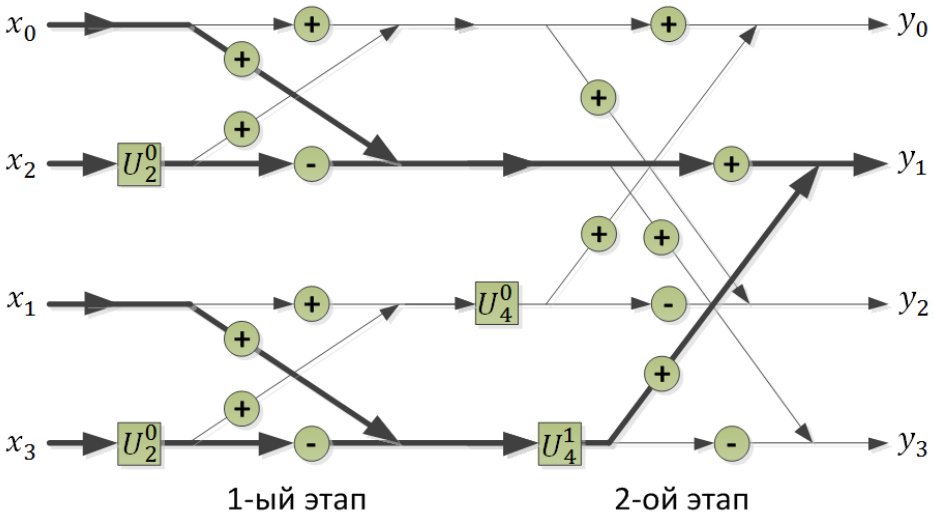


Рис. 9. Пример вычисления БПФ

Поскольку размер входного массива равен 4, вычисления состоят из двух этапов. В общем случае, когда размер входных данных равен степени двойки ($N = 2^n$), число этапов вычисления БПФ равно $n = log_2(N)$. На каждом этапе многократно выполняется одна и та же базовая операция – «бабочка» (Рис. 8).

На первом этапе вычислений происходит применение «бабочки» с шагом равным 1 ко всем элементам, т.е. ко всем парам, состоящим из соседних элементов входного массива. В нашем примере это пары (x_0, x_2) и (x_1, x_3). Коэффициент поворота «бабочки» будет равен 1 ($U_2^0 = e^0 = 1$) для всех обрабатываемых пар.

На втором этапе вычислений шаг «бабочки» увеличится в два раза, т.е. будет равен 2. Таким образом «бабочка» будет применяться к парам элементов (x_0, x_1) и (x_2, x_3). При этом коэффициенты поворота для этих пар будут различны: $U_4^0 = e^0 = 1$ и $U_4^1 = e^{-\frac{\pi}{2} \cdot i} = -i$ соответственно.

Таким образом, для вычисления БПФ над массивом из 4-х элементов потребуется 4 раза выполнить «бабочку». Как видно из Рис. 9 (жирные стрелки), на выходе y_1 будут присутствовать все элементы, подаваемые на вход, с разными коэффициентами (за счет коэффициентов поворота). Можно убедиться, что полученное значение полностью совпадает со значением, полученным при вычислении ДПФ по формуле (9) (выражение (16)). Аналогичным образом обстоит ситуация с остальными выходами.

$$y_1 = \sum_{n=0}^{3} x_n \cdot e^{-\frac{\pi}{2} \cdot n \cdot i} = x_0 - i \cdot x_1 - x_2 + i \cdot x_3 \tag{16}$$

Реализацию алгоритм Кули и Таки можно разбить на два этапа: подготовительный и вычислительный. Подготовительный этап состоит в перестановке элементов входного массива данных (с использованием *бит-реверсирования*). Так как соотношение «бабочки» делит множество точек на два подмножества, будет удобно, если все чётные элементы исходного массива будут располагаться плотно слева, а нечетные – плотно справа (Рис. 10). Перестановка элементов входного массива позволяет добиться этого на каждом этапе вычисления БПФ. Заметим, что подготовительный этап не является обязательным, он требуется лишь для упрощения реализации алгоритма.

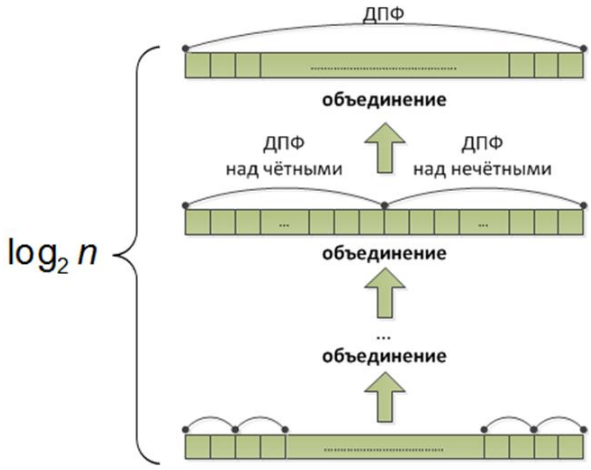


Рис. 10. Вычисление БПФ

Рассмотрим реализацию каждого этапа отдельно.

2.3.2. Реализация перестановки элементов массива с использованием бит-реверсирования

Данный этап заключается в изменении порядка следования исходных данных таким образом, чтобы с ними было удобно работать в дальнейшем. В некоторых алгоритмах БПФ этот этап пропускается и внедряется в вычислительный этап.

Бит-реверсирование – это преобразование двоичного числа путем изменение порядка следования бит в нем на противоположный. Бит-реверсирование применимо только к индексам элементов массива и предназначено для изменения порядка следования этих элементов, при этом значения самих элементов не изменяются. На Рис. 11 число 18 после применения бит-реверсирования преобразуется в 9: первый бит переходит на последнюю позицию, второй бит – на предпоследнюю и т.д. При этом реверсируются только первые 5 бит числа, т.к. исходный массив в данном примере содержит только 32 элемента (для хранения чисел от 0 до 31 необходимо использовать 5 бит).

Рис. 11. Бит-реверсирование числа 18

Рассмотрим алгоритм работы на примере. Пусть исходный массив содержит 16 элементов, тогда преобразование индексов будет происходить, как показано на Рис. 12: слева – исходные числа, справа от стрелки – реверсивные аналоги.

0	0	0	0	= 0	→	0	0	0	0	= 0
0	0	0	1	= 1	→	1	0	0	0	= 8
0	0	1	0	= 2	→	0	1	0	0	= 4
0	0	1	1	= 3	→	1	1	0	0	= 12
0	1	0	0	= 4	→	0	0	1	0	= 2
0	1	0	1	= 5	→	1	0	1	0	= 10
0	1	1	0	= 6	→	0	1	1	0	= 6
0	1	1	1	= 7	→	1	1	1	0	= 14
1	0	0	0	= 8	→	0	0	0	1	= 1
1	0	0	1	= 9	→	1	0	0	1	= 9
1	0	1	0	= 10	→	0	1	0	1	= 5
1	0	1	1	= 11	→	1	1	0	1	= 13
1	1	0	0	= 12	→	0	0	1	1	= 3
1	1	0	1	= 13	→	1	0	1	1	= 11
1	1	1	0	= 14	→	0	1	1	1	= 7
1	1	1	1	= 15	→	1	1	1	1	= 15

Рис. 12. Пример выполнения бит-реверсирования

Таким образом, исходный массив из 16 элементов ($a[0], a[1], \dots, a[15]$) после бит-реверсирования индексов элементов будет содержать элементы в следующем порядке: $a[0], a[8], a[4], a[12], a[2], a[10], a[6], a[14], a[1], a[9], a[5], a[13], a[3], a[11], a[7], a[15]$.

Далее будет рассмотрен алгоритм перестановки элементов массива с использованием бит-реверсирования, сохраняющий результат в новый массив (реализация out-of-place[5]). Первый этап алгоритма заключается в вычислении количества бит необходимого для хранения индексов элементов массивов, т.е. вычислении величины $log_2 size$ (напомним, что $size$ является степенью двойки, поэтому логарифм будет иметь целое значение). Вычисление количества бит можно сделать следующим простым циклом (**`size`** – количество элементов массива):

[5] Вычислительные алгоритмы разделяют на две группы: out-of-place и in-place. Для in-place алгоритмов преобразование исходных данных происходит с использованием малого, обычно константного, количества дополнительной памяти. В этом случае часто входные данные замещаются выходными. Алгоритмы out-of-place требуют дополнительной памяти, пропорциональной количеству исходных данных и не изменяют исходные данные. Алгоритмы in-place обычно сложнее, чем их out-of-place аналоги.

```
bitsCount = 0;
for (int tmp_size = size; tmp_size > 1; tmp_size/=2,
  bitsCount++ );
```

Следующий этап алгоритма состоит в выполнении бит-реверсирования для каждого индекса элемента входного массива и копировании обрабатываемого элемента на новую позицию в выходном массиве. Программный код, выполняющий перестановку элементов массива с использованием бит-реверсирования, представлен ниже.

```
void BitReversing(vector<complex<double>> inputSignal,
  vector<complex<double>> &outputSignal, int size)
{
  int bitsCount = 0;

  //Определение количества бит для бит-реверсирования
  //bitsCount = log2(size)
  for( int tmp_size = size; tmp_size > 1; tmp_size/=2,
       bitsCount++ );

  //Выполнение бит-реверсирования
  //ind - индекс элемента в  массиве input
  //revInd - соответствующие индексу ind индекс
  //(бит-реверсивный) в массиве output
  for(int ind = 0; ind < size; ind++)
  {
    int mask = 1 << (bitsCount - 1);
    int revInd = 0;

    for(int i=0; i<bitsCount; i++)//Бит-реверсирование
    {
      bool val = ind & mask;
      revInd |= val << i;
      mask = mask >> 1;
    }

    outputSignal[revInd] = inputSignal[ind];
  }
}
```

Рассмотрим выполнение бит-реверсирования на примере сигнала, содержащего 32 элемента. Количество бит необходимое для хранения индексов элементов массивов будет равно 5 ($log_2 32$). На Рис. 13 приведены значения основных переменных в зависимости от итерации цикла, выполняющего бит-реверсирование на примере числа 18.

Переменные	1 итерация	2 итерация	3 итерация	4 итерация	5 итерация
ind	1 0 0 1 0	1 0 0 1 0	1 0 0 1 0	1 0 0 1 0	1 0 0 1 0
mask	**1** 0 0 0 0	0 **1** 0 0 0	0 0 **1** 0 0	0 0 0 **1** 0	0 0 0 0 **1**
val	0 0 0 0 **1**	0 0 0 0 **0**	0 0 0 0 **0**	0 0 0 0 **1**	0 0 0 0 **0**
revInd	0 0 0 0 **1**	0 0 0 **0** 1	0 0 **0** 0 1	0 **1** 0 0 1	**0** 1 0 0 1

Рис. 13. Значения основных переменных при выполнении бит-реверсирования

2.3.3. Рекурсивный алгоритм

Будем считать, что предварительный этап выполнен, поэтому вычисления можно выполнять в соответствии со схемой, изображенной на Рис. 10. Алгоритм будет заключаться в выполнении операции объединения с использованием «бабочки» ко всем парам элементов массива.

С помощью «бабочки» выполняется вычисление дискретных отсчётов спектра, содержащего N точек, на основании дискретных отсчётов спектра для четных и нечетных отсчётов исходного сигнала (каждый такой спектр содержит $\frac{N}{2}$ точек). «Бабочка» применяется к паре спектральных отсчётов исходных ДПФ и получает два значения спектральных отсчётов результирующего ДПФ. Программный код, реализующий операцию «бабочки», представлен ниже.

```
void Butterfly(vector<complex<double>> &signal,
  complex<double> &u,//Коэффициент поворота
  int offset,        //Смещение обрабатываемых
                     //элементов в массиве
  int butterflySize) //Шаг «бабочки»
{
  complex<double> tem =
                     signal[offset +  butterflySize] * u;

  signal[offset + butterflySize] = signal[offset] - tem;
  signal[offset] += tem;
}
```

Рекурсивный алгоритм вычисления БПФ будет состоять из следующих шагов:

1. Вычисление БПФ над первой половиной массива.
2. Вычисление БПФ над второй половиной массива.
3. Применение «бабочки» ко всем парам элементов.

Программный код, реализующий вычисление БПФ, представлен ниже.

```
void SerialFFTCalculation(vector<complex<double>> &signal,
```

```
  int first, int size)
{
  if(size==1)
    return;

  double const coeff=2.0*PI/size;

  SerialFFTCalculation( signal, first,          size/2);
  SerialFFTCalculation( signal, first + size/2, size/2);

  for (int  j=first; j<first+size/2; j++ )
    Butterfly(signal,
              complex<double>(cos(-j*coeff),sin(-j*coeff)),
              j, size/2 );
}
```

2.3.4. Итеративный алгоритм

Рассмотрим псевдокод итеративного алгоритма вычисления БПФ. Пусть N – размер входных данных, **mas** – входной массив комплексных чисел. Алгоритм будет иметь in-place реализацию, поэтому посчитанные значения будут сохранены в исходный массив.

1. $s = 1$ (шаг «бабочки»), $b = log_2(N)$.
2. $i = 0, j = 0$.
3. Применить «бабочку» к элементам **mas**$[i \cdot 2 \cdot s + j]$ и **mas**$[i \cdot 2 \cdot s + j + s]$ с коэффициентом поворота $e^{-\frac{j}{s} \cdot \pi \cdot i}$.
4. Если $j < s - 1$, то $j = j + 1$, переход на шаг 3.
5. Если $\mathrm{i} < \frac{\mathrm{N}}{2 \cdot \mathrm{s}} - 1$, то $\mathrm{i} = \mathrm{i} + 1, \mathrm{j} = 0$, переход на шаг 3.
6. Если $\mathrm{s} < \frac{\mathrm{b}}{2}$, то $\mathrm{s} = \mathrm{s} \cdot 2$, переход на шаг 2, иначе завершение алгоритма.

3. Реализация программы, выполняющей фильтрацию видео

В данном разделе будет дано общее представление о структуре программы, библиотеках STL и OpenCV и частотной фильтрации.

3.1. Общая структура

3.1.1. Руководство пользователя

В данной работе рассматривается приложение, которое выполняет частотную фильтрацию кадров из видеофайла. В программе реализованы высо-

кочастотные и низкочастотные фильтры Гаусса и Баттерворта. Работающая программа выглядит так, как показано на Рис. 14.

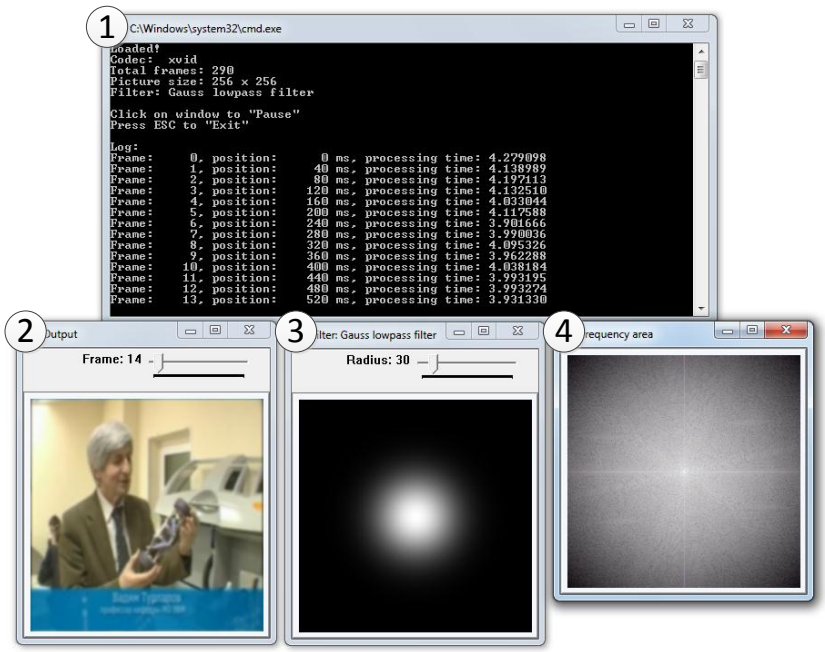


Рис. 14. Работа программы (фильтр низких частот Гаусса)

Приложение содержит 4 окна:

1. Основное консольное окно.
2. Окно, содержащее отфильтрованное изображение.
3. Окно фильтра.
4. Окно отфильтрованного логарифмического спектра изображения.

Запуск программы осуществляется из командной строки. Первым параметром является относительный путь к видеофайлу, далее указывается режим работы программы (ключ “**-s**”) и тип фильтра:

- “**-gl**” – фильтр низких частот Гаусса;
- “**-gh**” – фильтр высоких частот Гаусса;
- “**-bl**” – фильтр низких частот Баттерворта;
- “**-bh**” – фильтр высоких частот Баттерворта.

Если ключ “**-s**” указан, то отображается только одно консольное окно программы. Этот режим необходим для проведения вычислительных экс-

периментов. По умолчанию используется фильтр низких частот Гаусса, ключ “**`-s`**” не установлен и загружается видеофайл по следующему пути: “**`Videos\TestVideo1.avi`**”.

Пример запуска программы: “**`filter.exe video.avi -s -gh`**”.

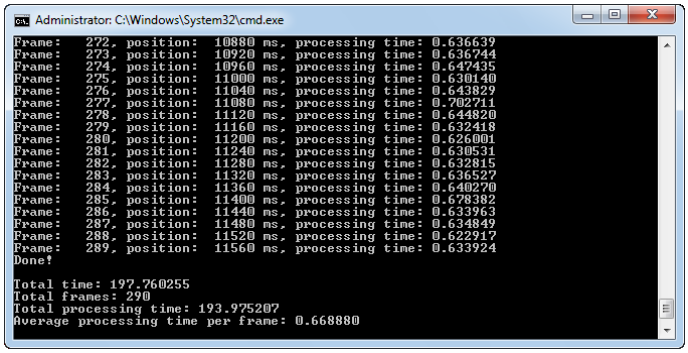


Рис. 15. Результат обработки видеофрагмента

После запуска программы в консольном окне выводится информация об обрабатываемом видеофрагменте (путь к видеофайлу, кодек, количество кадров, размер изображения) и используемый фильтр. По мере обработки видеофайла выводится лог, содержащий информацию об обрабатываемых кадрах: номер обрабатываемого кадра, положение кадра (в мс), время обработки кадра. После обработки всего видеофрагмента программа завершается и выводит суммарную статистику (Рис. 15): общее время обработки видеофрагмента, количество обработанных кадров, общее время фильтрации кадров, среднее время фильтрации кадра. Для принудительного завершения программы необходимо нажать на клавишу “ESC”. В случае возникновения ошибки в консоли выводится соответствующее сообщение, и программа автоматически завершается.

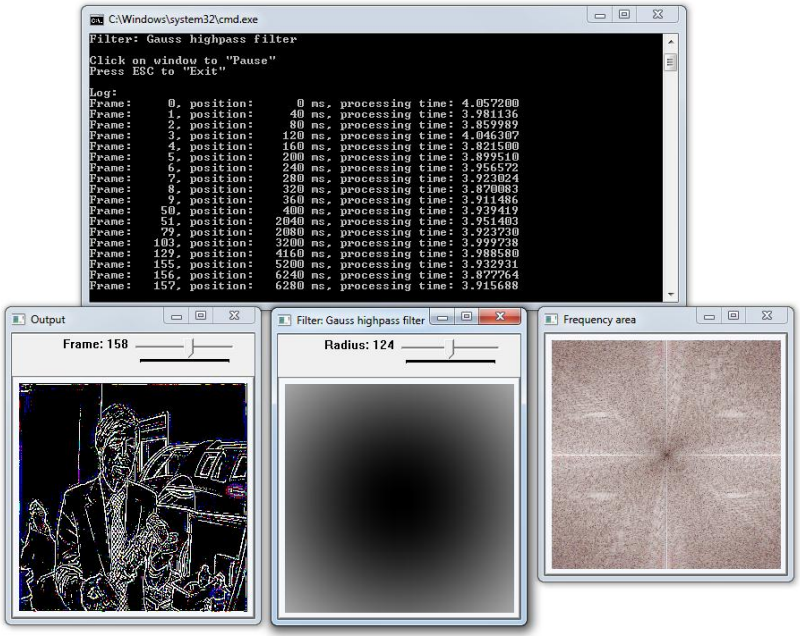


Рис. 16. Работа программы (фильтр высоких частот Гаусса)

Окно с отфильтрованным изображением содержит ползунок (TrackBar), отображающий текущую позицию обрабатываемого кадра. С помощью этого ползунка можно перейти к любому кадру обрабатываемого видеофрагмента.

Окно фильтра содержит ползунок, который задаёт параметр фильтра. Чем больше значение этого параметра, тем сильнее эффект фильтрации.

При щелчке мышью по области, которая содержит фильтр, отфильтрованное изображение или логарифмический спектр, программа переводится в режим паузы. В этом режиме смена обрабатываемого кадра не происходит. Для отмены режима паузы необходимо снова щелкнуть по области, содержащей фильтр, отфильтрованное изображение или логарифмический спектр.

3.1.2. Руководство программиста

Откройте базовую версию проекта:

“**`2. filter (simple)\filter.sln`**”.

Проект содержит один заголовочный файл и два cpp-файла (Рис. 17).

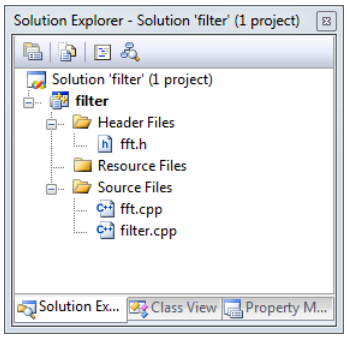


Рис. 17. Состав проекта

Алгоритм работы программы состоит из следующих шагов:

1. Разбор командной строки.

Выполняется проверка корректности аргументов командной строки, вывод сообщения об ошибке в случае обнаружения неизвестных символов в командной строке и заполнение переменных, задающих режим работы приложения: тип используемого фильтра **`fType`** и булева переменная **`silent`**, запрещающая визуализацию, если значение переменной равно истине.

2. Загрузка видеофайла.

На данном этапе выполняется проверка, разрешено ли чтение видеофайла. Если файл отсутствует или доступ к нему запрещён (например, в связи с тем, что файл используется другим приложением), то печатается соответствующее предупреждение и программа завершается. Далее выполняется захват видеофайла средствами OpenCV. Если кодек открываемого файла не поддерживается, то выводится предупреждение и программа завершается.

3. Получение информации о видеофайле.

Выполняется считывание информации о загруженном видеофайле: количество кадров, видеокодек, ширина и высота кадра.

4. Проверка корректности размеров кадра.

Проверяется ширина и высота кадра на равенство степени двойки. Если это не так, то выводится сообщение об ошибке и программа завершается.

5. Создание семафора, отвечающего за потоковую функции визуализации.

Далее с помощью семафора выполняется

6. Печать информации о загруженном видеофайле.
7. Создание изображений средствами OpenCV для визуализации.
8. Создание потока для визуализации.

9. Цикл обработки очередного кадра изображения.

Цикл выполняется до тех пор, пока не будет достигнут конец видеофайла или пользователь на нажмёт клавишу ESC. В цикле последовательно выполняется установка номера очередного обрабатываемого кадра, загрузка кадра, обработка кадра с помощью функции **`ProcessFrame()`** и измерение времени работы этой функции, увеличение счётчика количества обработанных кадров, вывод на экран информации об обработанном кадре, проверка условий выхода из цикла (нажата ли клавиша ESC, завершён ли поток визуализации, т.е. нажатие клавиши ESC произошло в этом потоке), увеличение номера обрабатываемого кадра, если не включен режим паузы.

10. Печать суммарной статистики.

На экран выводится общее время работы цикла обработки кадров, количество обработанных кадров, суммарное время обработки кадров, среднее время обработки кадра.

11. Освобождение ресурсов.

Полный программный код простейшей версии программы представлен в приложении 21.2.

3.2. Частотная фильтрация

Идея частотной фильтрации заключается в преобразовании спектра изображения за счёт его умножения на функцию фильтра.

Исходное изображение переводится в частотную область с помощью применения прямого ДПФ. Предварительно исходное изображение домножается на функцию -1^{x+y} для того, чтобы центрировать спектр изображения. После этого функция-спектр домножается на функцию фильтра. Далее выполняется обратное ДПФ (предварительно отфильтрованный спектр снова домножается на -1^{x+y}) и получается отфильтрованное изображение.

Мы рассмотрим 4 фильтра:

- фильтр низких частот Баттерворта;
- фильтр высоких частот Баттерворта;
- фильтр низких частот Гаусса;
- фильтр высоких частот Гаусса.

Фильтры низких частот обрезают высокочастотные составляющие фурье-образа, находящиеся на большом расстоянии от начала координат (Рис. 18).

Рис. 18. Пример низкочастотной фильтрации

Фильтры высоких частот обрезают низкочастотные составляющие фурье-образа, находящиеся на близком расстоянии от начала координат (Рис. 19).

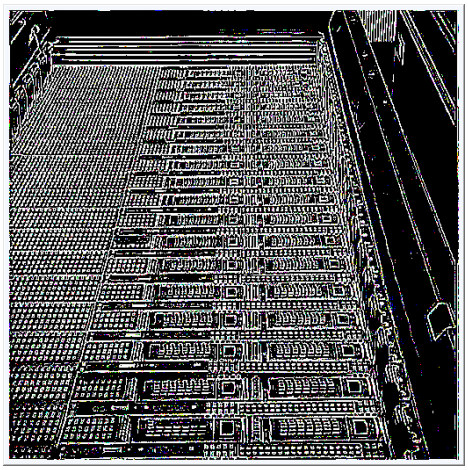


Рис. 19. Пример высокочастотной фильтрации

Пусть $D(u,v)$ задаёт евклидово расстояние точки с координатами u,v от начала координат:

$$D(u,v) = \sqrt{\left(u - \frac{M}{2}\right)^2 + \left(v - \frac{N}{2}\right)^2} \tag{17}$$

Фильтр высоких частот Баттерворта задаётся формулой (18), где n – порядок фильтра, D_0 – расстояние от начала координат, задающее частоту среза, $D(u,v)$ – расстояние точки с координатами u,v от начала координат (17).

$$H(u,v) = \frac{1}{1 + (\frac{D_0}{D(u,v)})^{2n}} \tag{18}$$

Фильтр высоких частот Гаусса задаётся формулой (19).

$$H(u,v) = 1 - e^{-\frac{D^2(u,v)}{2D_0^2}} \tag{19}$$

Фильтр низких частот Гаусса задаётся формулой (20).

$$H(u,v) = e^{-\frac{D^2(u,v)}{2D_0^2}} \tag{20}$$

Фильтр низких частот Баттерворта задаётся формулой (21).

$$H(u,v) = \frac{1}{1 + (\frac{D(u,v)}{D_0})^{2n}} \tag{21}$$

3.3. Библиотека Standard Template Library

Библиотека Standard Template Library (STL) – это стандартная библиотека языка C++, которая содержит реализации контейнеров (вектор, стек, очередь, хеш-таблица и др.) и алгоритмов, позволяющих манипулировать данными в этих контейнерах. Большинство контейнеров STL являются шаблонными классами, поэтому хранить в этих контейнерах допускается любой тип данных. Более подробная информация о библиотеке STL представлена в [6].

В рассматриваемой программе будет использоваться контейнер **`vector`** для хранения компонентов цвета обрабатываемого изображения. Этот контейнер выбран за счёт того, что он обеспечивает прямой доступ к элементам и константное время вставки элементов в конец вектора.

В программе также будет использован класс **`string`**, позволяющий упростить работу со строками.

3.4. Библиотека OpenCV

Библиотека компьютерного зрения OpenCV позволяет эффективно выполнять обработку и визуализацию изображений и видео. Помимо этого библиотека содержит большое количество алгоритмов компьютерного зрения.

В рассматриваемой программе будет использован простейший функционал библиотеки OpenCV, позволяющий работать с отдельными кадрами видеофайла и выполнять визуализацию обработанного кадра.

4. Поиск ошибок в проекте

Откройте стартовый проект **filter.sln** расположенный в директории **C:\ParallelCalculus\05_FastFourierTransform\00. filter (bugs)**, последовательно выполняя следующие шаги:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\00. filter (bugs)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Программный код содержат ряд синтаксических ошибок, которые можно обнаружить при помощи компилятора.

Выполните сборку проекта. Для этого выполните команду **Build Solution** в меню **Build**.

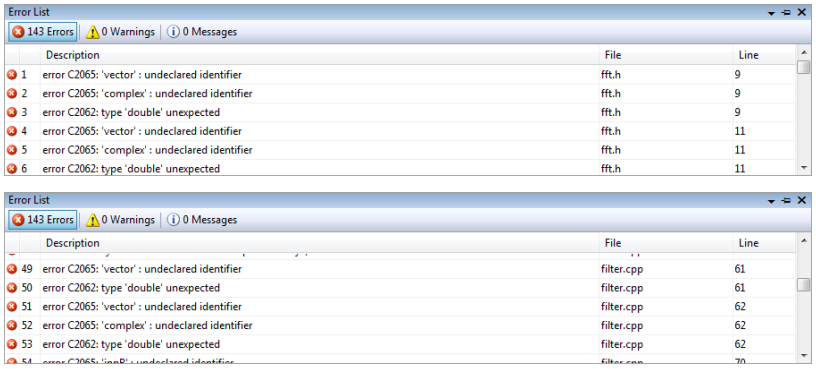


Рис. 20. Ошибки компиляции 1

Сборка включает две стадии: компиляцию и линковку. После компиляции в окне **Error List** будет отображено 143 ошибки (Рис. 20), поэтому линковка исполняемого модуля не будет выполнена. В данном случае проект достаточно мал и его компиляция занимает несколько секунд, соответственно исправлять ошибки имеет смысл последовательно, то есть сначала исправить одну найденную ошибку, потом снова запустить компиляцию. Такой подход удобен тем, что наличие одной ошибки в коде может приводить к появлению новых ошибок, которые являются всего лишь следствием первой[6].

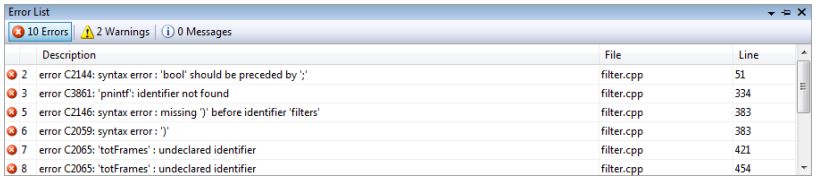
---

[6] Для «больших» проектов, где объем исходного кода измеряется мегабайтами, а время сборки десятками минут и часами, такой подход, конечно же, неприменим. Хотя необходи-

Большинство представленных ошибок на Рис. 20 вызваны тем, что компилятор обнаружил неизвестные идентификаторы (undeclared identifier), которые использованы в проекте: **`vector`**, **`complex`**. Для решения этой проблемы необходимо указать пространство имён **`std`**, в котором объявлены указанные типы. Для этого достаточно подключить пространство имен, используя конструкцию **`using namespace`**:

```
using namespace std;
```

Добавьте программный код, представленный выше, в файлы **filter.cpp** (строка 14) и **fft.h** (строка 6). Снова выполните компиляцию исходных кодов проекта.



Error List

10 Errors | 2 Warnings | 0 Messages

	Description	File	Line
2	error C2144: syntax error : 'bool' should be preceded by ';'	filter.cpp	51
3	error C3861: 'pnintf': identifier not found	filter.cpp	334
5	error C2146: syntax error : missing ')' before identifier 'filters'	filter.cpp	383
6	error C2059: syntax error : ')'	filter.cpp	383
7	error C2065: 'totFrames' : undeclared identifier	filter.cpp	421
8	error C2065: 'totFrames' : undeclared identifier	filter.cpp	454

Рис. 21. Ошибки компиляции 2

Количество ошибок значительно сократится – должно остаться 10 сообщений (Рис. 21). Ошибка с идентификатором 2 ('bool' should be preceded by ';') вызвана отсутствием разделителя “`;`”.

```
int r = 30;
```

Добавьте “`;`” в конец строки 50 файла **filter.cpp**. Ошибка с идентификатором 3 ('pnintf': identifier not found) вызвана использованием в программе неизвестного идентификатора. В данном случае ошибка связана с опечаткой в названии функции **`printf`**.

```
printf("%s\n", helpString);
```

Поправьте название функции в 334 строке файла **filter.cpp**. Ошибки с идентификаторами 7 и 8 ('totFrames' : undeclared identifier) вызваны использованием в программе необъявленной переменной, которая хранит количество обработанных кадров.

```
int totFrames = 0;
```

Объявите переменную **`totFrames`** в строке 275 файла **filter.cpp**. Выполните компиляцию исходных кодов проекта.

мо отметить, что при правильной разработке ситуации, когда результатом компиляции является сотня-другая ошибок, быть не должно.

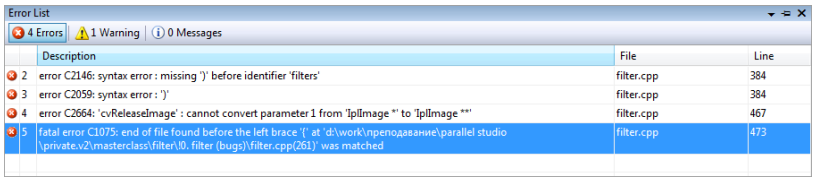


Рис. 22. Ошибки компиляции 3

Ошибка с идентификатором 4 ('cvReleaseImage': cannot convert parameter 1 from 'IplImage *' to 'IplImage **') вызвана несоответствием прототипа функции и передаваемого аргумента (Рис. 22). В данном случае аргументом функции освобождения ресурсов должен быть двойной указатель, а передаётся только одинарный указатель.

```
cvReleaseImage(&empty);
```

Допишите оператор взятия адреса **&** перед аргументом функции в строке 467 файла **filter.cpp**. Ошибка с идентификатором 2 (missing ')' before identifier 'filters') вызвана неправильной передачей параметров в функцию **printf**.

```
printf("Filter: %s\n", filters[fType]);
```

Допишите запятую после строки форматирования функции **printf** в строке 384 файла **filter.cpp**. Ошибка с идентификатором 5 (end of file found before the left brace '{') вызвана тем, что количество открывающихся фигурных скобок не соответствует количеству закрывающихся фигурных скобок. Допишите закрывающуюся фигурную скобку **}** в строке 317 файла **filter.cpp**.

Выполните сборку проекта. Убедитесь, что все ошибки компиляции устранены и в результате линковки собирается исполняемый модуль **filter.exe**.

5. Отладка программы

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\01. filter (errors)** в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\01. filter (errors)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Debug** версии проекта (Рис. 23) с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

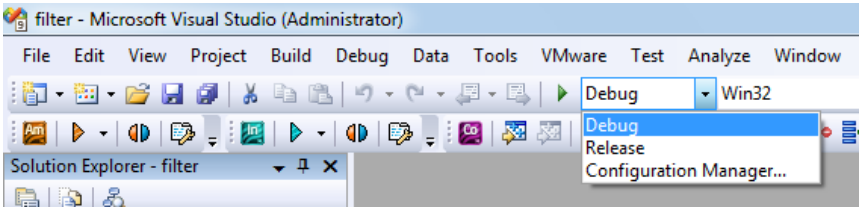


Рис. 23. Выбор **Debug** версии проекта для сборки

Программный код содержит ряд ошибок работы с памятью, которые можно обнаружить с помощью инструмента Intel Inspector XE.

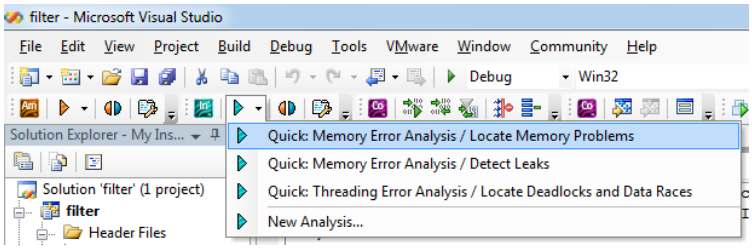


Рис. 24. Intel Inspector XE. Анализ ошибок работы с памятью

Для поиска ошибок работы с памятью воспользуйтесь инструментом Intel Inspector XE (Рис. 24):

- используйте **Debug**-версию программы;
- выберите режим работы **Quick: Memory Error Analysis / Locate Memory Problems**;
- нажмите **ESC** после начала обработки первого кадра (для анализа достаточно одного обработанного кадра).

После завершения программы и обработки собранных данных Intel Inspector XE выдаст список найденных ошибок. Среди них будет обнаружена 41 утечка памяти (Рис. 25). Согласно информации Intel Inspector XE все утечки происходят в исходных файлах библиотеки OpenCV, но интерпретация информации сильно затруднена отсутствием связи с исходными кодами нашей программы.

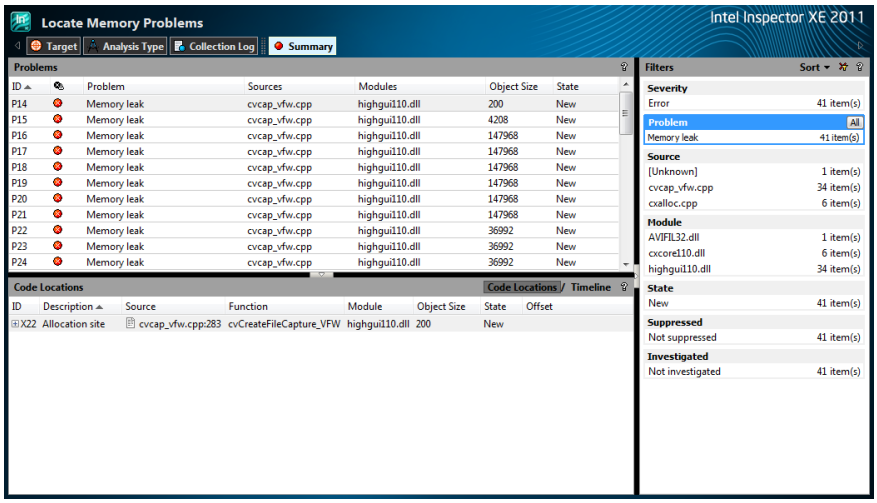


Рис. 25. Intel Inspector XE. Утечки памяти

Тем не менее, хотя мы и не можем посмотреть на конкретные операторы, с которыми Intel Inspector XE связывает найденные утечки, нетрудно предположить, что проблема почти наверняка не в библиотеке OpenCV, а в том, как мы используем ее объекты. В данном случае имеют место следующие моменты.

- Не освобождается ресурс **`film`** перед завершением работы программы. *Выполните освобождение ресурса* **`film`** *в строке 465 файла* **filter.cpp**:

```
cvReleaseCapture(&film);
```

- Не закрывается открытый файл. *Выполните закрытие файла* **`f`** *в строке 329 файла* **filter.cpp**:

```
fclose(f);
```

- Не уничтожается одно из окон перед завершением работы программы. *Выполните уничтожение окна с заголовком* **`fStr`** *в строке 256 файла* **filter.cpp**:

```
cvDestroyWindow(fStr);
```

- Не освобождаются ресурсы картинки. *Выполните освобождение ресурса* **`filter`** *в строке 467 файла* **filter.cpp**:

```
cvReleaseImage(&filter);
```

Теперь можно осуществить первый запуск приложения. Выполните команду **Rebuild Solution** в меню **Build** для сборки проекта. Нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**. После запуска исполняемого модуля появится 4 окна программы (Рис. 26).

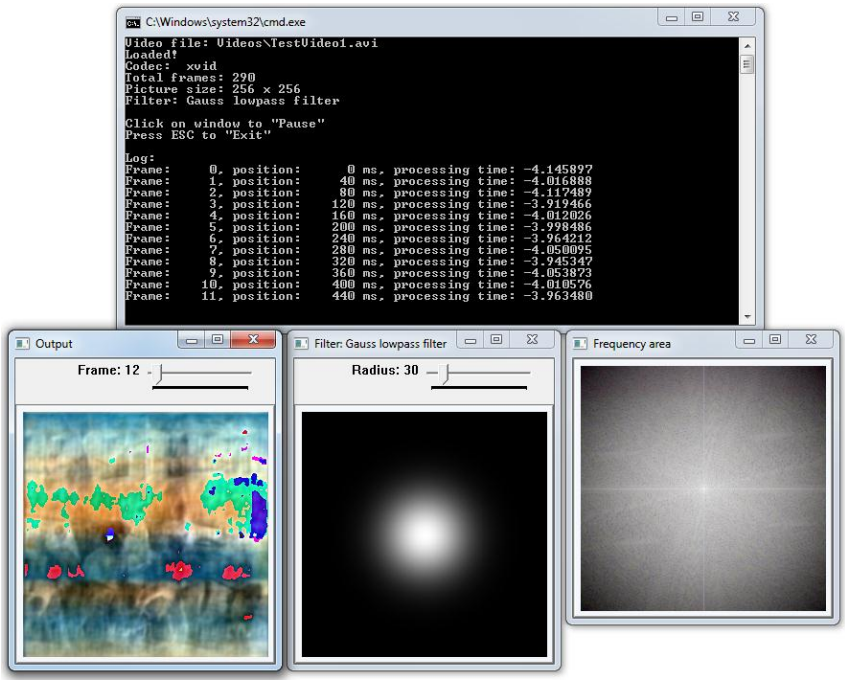


Рис. 26. Работа программы

Программа запускается, но содержит ошибки, которые легко заметить: изображение в окне **Output** некорректно, время обработки кадров отрицательное.

Типичным подходом при измерении времени является считывании текущего значения счётчика времени перед выполнением измеряемого участка кода и после него и вычитание из конечного времени начального. Отрицательное время обработки кадра является ошибкой в последовательности операндов. *Поменяйте уменьшаемое и вычитаемое местами или допишите перед разностью знак “–“ в строке 426 файла* **filter.cpp**:

```
printf("Frame: %5i, position: %6i ms, processing time:
        %f\n", frame, msec,
        -(start.QuadPart - finish.QuadPart) /
        (double)fr.QuadPart);
```

Обработка кадра происходит в большей части программы, поэтому неправильное изображение в окне **Output** может быть вызвано множеством причин, поиск такой ошибки очень трудоёмок. Тем не менее, можно заметить, что окно фильтра и окно отфильтрованного логарифмического спектра содержат корректные изображения (это можно допустить, сравнив Рис. 14 и Рис. 26). Такое допущение уменьшает круг поиска и приводит к тому, что ошибка может быть только при выполнении ПФ. При этом прямое ПФ вы-

полнилось верно, т.к. спектр изображения корректен, следовательно, ошибка возникает при выполнении обратного ПФ.

```
void SerialFFTCalculation(vector<complex<double>> &signal,
  int first, int size, int step, int offset,
  bool forward=true)
{
  if(size==1)
    return;

  double const coeff=2.0*PI/size;

  SerialFFTCalculation( signal, first, size/2,
                        step, offset);
  SerialFFTCalculation( signal, first + size/2, size/2,
                        step, offset, forward);

  for (int  j=first; j<first+size/2; j++ )
    if(forward)
      Butterfly(signal,
         complex<double>(cos(-j*coeff), sin(-j*coeff)), j ,
         size/2, step, offset );
    else
      Butterfly(signal,
         complex<double>(cos(j*coeff), sin(j*coeff)), j ,
         size/2, step, offset );
}
```

Вычисление прямого и обратного ПФ происходит в одной функции **`SerialFFTCalculation`**. Направление преобразования задаётся через параметр **`forward`** этой функции, значение по умолчанию **`true`** соответствует прямому ПФ. Функция **`SerialFFTCalculation`** рекурсивная, но в первом вызове этой функции параметр не передаётся, что является ошибкой, т.к. при дальнейших рекурсивных вычислениях будет вычисляться только прямое ПФ. *Добавьте параметр* **`forward`** *в рекурсивный вызов функции* **`SerialFFTCalculation`** *в строку 44 файла* **fft.cpp**:

```
  SerialFFTCalculation(signal, first, size/2,
    step, offset, forward);
```

Выполните сборку проекта и запуск полученного приложения. Убедитесь (визуально), что программа работает корректно (Рис. 14).

6. Оценка эффективности программы

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в ди-

ректории **C:\ParallelCalculus\05_FastFourierTransform\02. filter (simple)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\02. filter (simple)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);

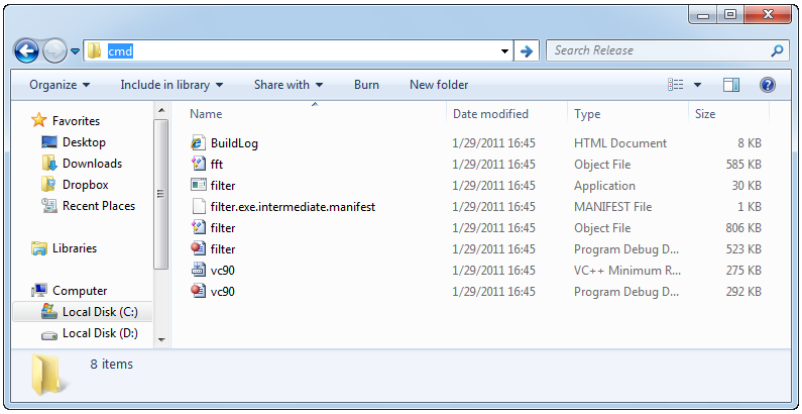


Рис. 27. Запуск интерпретатора командной строки

- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 128.5 с. Для ролика продолжительностью 11.5 с. это очень медленно, т.к. на обработку 1 секунды видео (25 кадров) затрачивается более 10 секунд. На обработку одного кадра тратится 0.44 с.

Итак, программа работает очень медленно и нам предстоит ускорить её выполнение. Для этого первым делом необходимо определить «горячие точки» (hotspots) программы – те функции, на выполнение которых тратится наибольшее время. Для поиска «горячих точек» необходимо воспользоваться профилировщиком.

Профилировщики предназначены для измерения производительности всего приложения или его участков. Они позволяют выполнять замеры времени работы и подсчёт количества вызовов, как функций, так и отдельных команд. Полученная с помощью профилировщика информация о времени выполнения каждого участка кода приложения позволяет обнаружить наиболее медленные из них и провести дальнейшую оптимизацию.

На данный момент существует множество простых профилировщиков, которые работают только на уровне функций (например, gprof в ОС Linux). Для их работы в исполняемом файле должна содержаться специальная информация для профилирования, что может исказить поведение приложения. Наиболее мощным на сегодняшний день профилировщиком является Intel VTune Amplifier XE, который позволяет собирать более детальную информацию о работе приложения, чем его аналоги. Intel VTune Amplifier XE может выполнять не только замеры времени выполнения отдельных команд, но и подсчёт числа событий процессора, которые были вызваны выполнением этих команд (промах кеш-памяти, неправильно предсказанное ветвление и др.).

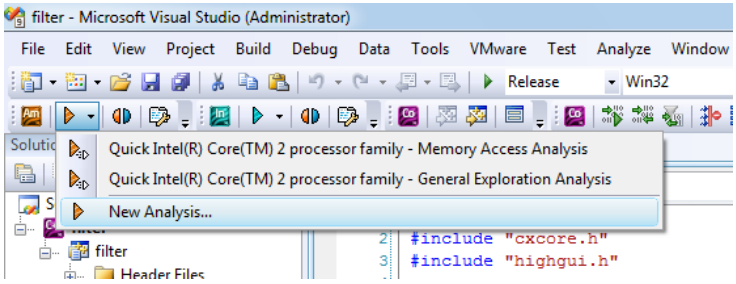


Рис. 28. Intel VTune Amplifier XE. Пользовательский анализ

Для поиска «горячих точек» воспользуйтесь инструментом Intel VTune Amplifier XE (Рис. 28):

- выберите **Release**-версию программы;
- используйте режим работы **Hotspots** (Рис. 29).

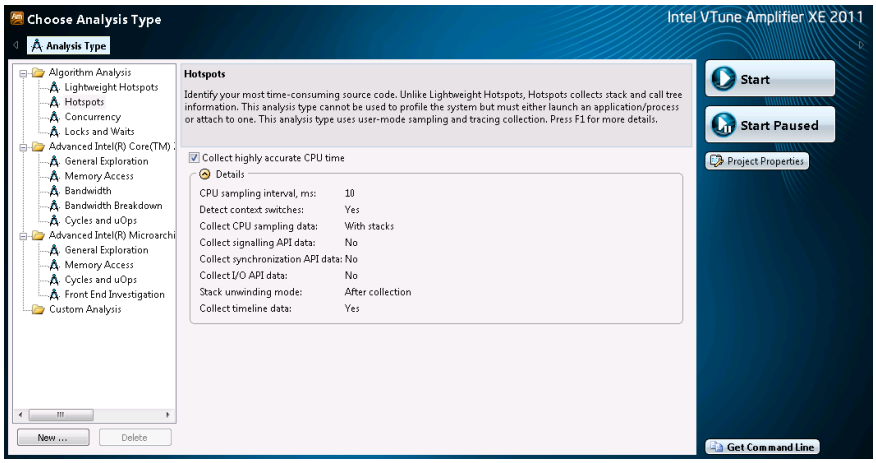


Рис. 29. Intel VTune Amplifier XE. Поиск “горячих” точек

После завершения программы и обработки собранных данных Intel VTune Amplifier XE выдаст список функций, на выполнение которых затрачивается наибольшее время (Рис. 30). Функция вычисления БПФ **`SerialFFTCalculation()`** является самой «горячей». Оптимизация этой функции потенциально может дать наибольший выигрыш.

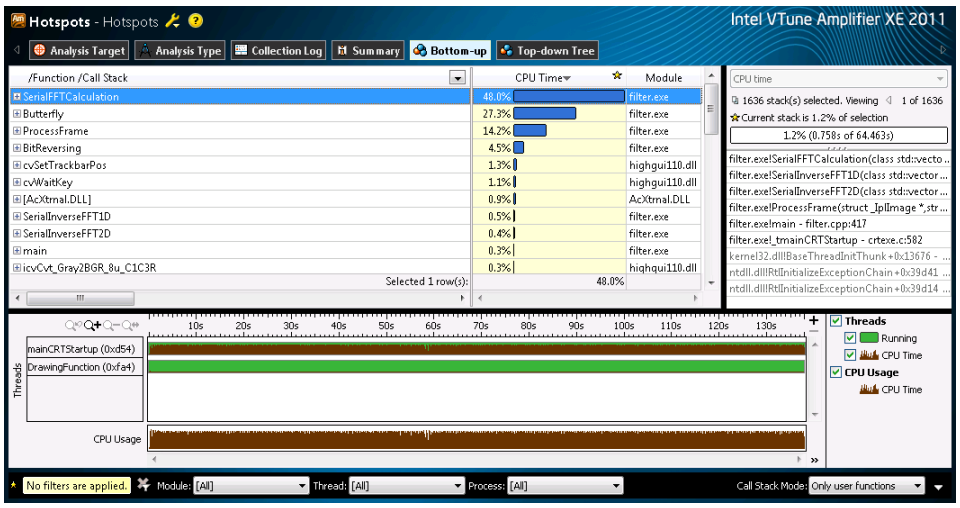


Рис. 30. Intel VTune Amplifier XE. «Горячие точки» программы

Функция **`SerialFFTCalculation()`** выполняется почти половину времени работы программы (48%). Рассмотрим эту функцию подробнее. Нажмите на функции **`SerialFFTCalculation()`** два раза левой кнопкой мыши (Рис. 31).

В открывшемся окне представлен исходной код программы с указанием процента времени, затраченного на выполнение каждой строки.

Line	Source	CPU Time
37	void SerialFFTCalculation(vector<complex<double>> &signal, int first, int size, int step, int offset, b	
38	{	0.2%
39	if(size==1)	0.3%
40	return;	
41		
42	double const coeff=2.0*PI/size;	0.2%
43		
44	SerialFFTCalculation(signal, first, size/2, step, offset, forward);	1.2%
45	SerialFFTCalculation(signal, first + size/2, size/2, step, offset, forward);	0.5%
46		
47	for (int j=first; j<first+size/2; j++)	0.1%
48	if(forward)	0.8%
49	Butterfly(signal, complex<double>(cos(-j*coeff), sin(-j*coeff)), j , size/2, step, offset);	22.0%
50	else	
51	Butterfly(signal, complex<double>(cos(j*coeff), sin(j*coeff)), j , size/2, step, offset);	22.1%
52	}	0.7%
	Selected 2 row(s):	44.1%

Рис. 31. Профиль функции **`SerialFFTCalculation`**

Из профиля функции **`SerialFFTCalculation()`** видно, что она вызывает в цикле функцию **`Butterfly()`**, которая занимает 44.1% от общего времени работы программы. Перейдём к анализу этой функции (Рис. 32).

Line	Source	CPU Time
28		
29	void Butterfly(vector<complex<double>> &signal, complex<double> u, int offset, int butterflySize, int step, int off)	
30	{	0.4%
31	complex<double> tem = signal[off + step*(offset + butterflySize)] * u;	15.4%
32		
33	signal[off + step*(offset + butterflySize)] = signal[off + step*(offset)] - tem;	7.4%
34	signal[off + step*(offset)] += tem;	3.3%
35	}	0.9%
36		
	Selected 3 row(s):	26.0%

Рис. 32. Профиль функции **`Butterfly`**

Функция **`Butterfly()`** выполняет простейшие действия, но за счёт большого количества вызовов является самым «узким местом» в программе. Простейшие действия заключаются в выполнении операций сложения, вычитания, умножения, индексирования и присваивания. Дело в том, что эти операции перегружены и выполняются для классов **`complex`** и **`vector`**, что негативно сказывается на быстродействии.

Необходимо понимать, что главное достоинство библиотеки STL в универсальности и экономии времени на разработку собственных реализаций, но не в производительности. STL-контейнеры работают достаточно медленно. Компилятор генерирует наиболее эффективный код при использовании простых (встроенных) типов данных. Таким образом, откажемся от использования контейнера **`std::vector`** – будем использовать обычные массивы.

Дополнительно отметим, что в программе происходит неэффективное использование контейнера **`std::vector`**, который передаётся в функцию по значению, что приводит к дополнительным накладным расходам (Рис. 33). Более правильно передавать этот контейнер по ссылке.

```
60 void SerialFFT2D(vector<complex<double>> inputSignal, vector<complex<double>> &outputSignal, int w, int h)
61 {
62   vector<complex<double>> tem(w*h);
63
64   for(int i=0; i<w; i++)
65     SerialFFT1D(inputSignal, tem, h, w, i);
66
67
68   for(int j=0; j<h; j++)
69     SerialFFT1D(tem, outputSignal, w, 1, h*j);
70 }
71
```

Рис. 33. Передача контейнера по значению

Контейнер **`std::vector`** обеспечивает как прямой доступ к элементам с помощью оператора индексирования, так и позволяет добавлять элементы в конец, динамически увеличивая свой размер. Таким образом, использование функции добавления элемента в конец контейнера **`push_back()`** приводит к замедлению программы (Рис. 34), использование оператора индексирования с предварительным выделением памяти выглядит предпочтительным.

```
62   vector<double> f(size);
63   vector<complex<double>> inpR, inpG, inpB,
64                           outR(size), outG(size), outB(size);
65
66   double maxR = 0.0 , maxG = 0.0, maxB = 0.0;
67   double minR = 0.0 , minG = 0.0, minB = 0.0;
68
69   //Fill inp<R|G|B> arrays
70   for (int i=0; i<size; i++)
71     inpB.push_back(pow(-1.0,i/w+i%w) * ((unsigned char*)frame->imageData)[i*3 + 0]);
72
73   for (int i=0; i<size; i++)
74     inpG.push_back(pow(-1.0,i/w+i%w) * ((unsigned char*)frame->imageData)[i*3 + 1]);
75
76   for (int i=0; i<size; i++)
77     inpR.push_back(pow(-1.0,i/w+i%w) * ((unsigned char*)frame->imageData)[i*3 + 2]);
78
```

Рис. 34. Неэффективное заполнение контейнера

Для перехода от использования контейнера **`std::vector`** к массивам необходимо выполнить следующие действия: исправить прототипы функций, изменить создание объектов, заменить функцию **`push_back()`** на оператор индексирования.

Прототипы функций в файле **fft.h** *примут следующий вид*:

```
void SerialFFT2D(complex<double> *inputSignal,
  complex<double> *outputSignal, int w, int h);

void SerialInverseFFT2D(complex<double> *inputSignal,
  complex<double> *outputSignal, int w, int h);
```

Прототипы функций в файле **fft.cpp** *примут следующий вид*:

```
…
void BitReversing(complex<double> *inputSignal,
```

```
  complex<double> *outputSignal,
  int size, int step, int offset)
…

void Butterfly(complex<double> *signal, complex<double> u,
  int offset, int butterflySize, int step, int off)
…

void SerialFFTCalculation(complex<double> *signal,
  int first, int size, int step,
  int offset, bool forward=true)
…

void SerialFFT1D(complex<double> *inputSignal,
  complex<double> *outputSignal,
  int size, int step, int offset)
…

void SerialInverseFFT1D(complex<double> *inputSignal,
  complex<double> *outputSignal,
  int size, int step, int offset)
…
```

Функции вычисления двумерного ПФ в файле **fft.cpp** *подвергнутся более значительным модификациям и примут следующий вид*:

```
void SerialFFT2D(complex<double> *inputSignal,
  complex<double> *outputSignal, int w, int h)
{
  complex<double> *tem = new complex<double>[w*h];

  for(int i=0; i<w; i++)
    SerialFFT1D(inputSignal, tem, h, w, i);


  for(int j=0; j<h; j++)
    SerialFFT1D(tem, outputSignal, w, 1, h*j);

  delete[] tem;
}

void SerialInverseFFT2D(complex<double> *inputSignal,
  complex<double> *outputSignal, int w, int h)
{
  complex<double> *tem = new complex<double>[w*h];

  for(int i=0; i<w; i++)
    SerialInverseFFT1D(inputSignal, tem, h, w, i);

  for(int j=0; j<h; j++)
    SerialInverseFFT1D(tem, outputSignal, w, 1, h*j);
```

```
  delete[] tem;
}
```

Изменения в файле **filter.cpp** *коснутся только функции обработки кадра* **ProcessFrame**:

```
void ProcessFrame(IplImage *frame, IplImage *outFrame,
  IplImage* filter, IplImage* freq, FilterType t, int *_r)
{
  int w = frame->width,
      h = frame->height,
      r = *_r,
      size = w * h;

  double *f=new double[size];
  complex<double> *inpR = new complex<double>[size],
                  *inpG = new complex<double>[size],
                  *inpB = new complex<double>[size],
                  *outR = new complex<double>[size],
                  *outG = new complex<double>[size],
                  *outB = new complex<double>[size];

  double maxR = 0.0 , maxG = 0.0, maxB = 0.0;
  double minR = 0.0 , minG = 0.0, minB = 0.0;

  //Fill inp<R|G|B> arrays
  for (int i=0; i<size; i++)
    inpB[i] = pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 0];

  for (int i=0; i<size; i++)
    inpG[i] = pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 1];

  for (int i=0; i<size; i++)
    inpR[i] = pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 2];

…
  delete[] inpB;
  delete[] inpG;
  delete[] inpR;
  delete[] outG;
  delete[] outR;

  delete f;
}
```

7. Оптимизация. Контейнеры STL

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\03. filter (down with STL)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\03. filter (down with STL)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Программный код содержит ряд ошибок работы с памятью, которые можно обнаружить с помощью инструмента Intel Inspector XE.

Выполните сборку **Debug** версии проекта (Рис. 23) с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Для поиска ошибок работы с памятью воспользуйтесь инструментом Intel Inspector XE (Рис. 24):

- используйте **Debug**-версию программы;
- выберите режим работы **Quick Memory Error Analysis / Locate Memory Problems**;
- нажмите **ESC** после начала обработки первого кадра (достаточно одного обработанного кадра).

После завершения программы и обработки собранных данных Intel Inspector XE выдаст список найденных ошибок (Рис. 35).

ID		Problem	Sources	Modules	Object Size	State
P1		Mismatched allocation/deallocation	filter.cpp	filter.exe		New
P2		Kernel resource leak	cvcap_vfw.cpp	highgui110.dll		New
P3		Kernel resource leak	grfmt_base.cpp	highgui110.dll		New
P4		Kernel resource leak	initcrit.c	MSVCR90D.dll		New
P5		Uninitialized memory access	cvcap_vfw.cpp	highgui110.dll		New
P6		Uninitialized memory access	cvcap_vfw.cpp	highgui110.dll		New
P7		Uninitialized memory access	cxalloc.cpp; cxcopy.cpp	cxcore110.dll		New
P8		Uninitialized memory access	cxalloc.cpp; cxcopy.cpp	cxcore110.dll		New
P9		Uninitialized memory access	cxalloc.cpp; cxcopy.cpp	cxcore110.dll		New
P10		Uninitialized memory access	cxalloc.cpp; cxcopy.cpp	cxcore110.dll		New
P11		Uninitialized memory access	cxalloc.cpp; cxcopy.cpp	cxcore110.dll		New
P12		Uninitialized memory access	cxalloc.cpp; utils.cpp	cxcore110.dll; highgui110.dll		New
P13		Uninitialized partial memory access	cvcap_vfw.cpp	highgui110.dll		New
P14		Memory leak	filter.cpp	filter.exe	1048576	New

Рис. 35. Intel Inspector XE. Список ошибок

Из всего списка к нашему приложению непосредственно относятся диагностические сообщения 1 и 14. Ошибка с идентификатором 14 является утечкой памяти (memory leak). Такие ошибки обычно вызваны тем, что в программе происходит выделение динамической памяти, но не выполняется её освобождение. Нажмите два раза левой кнопкой мыши на предупреждении об ошибке. Рассмотрим ее детальное описание (Рис. 36).

Focus Code Location: filter.cpp:68 - Allocation site

```
60          size = w * h;
61
62      double *f=new double[size];
63      complex<double> *inpR = new complex<double>[size],
64                      *inpG = new complex<double>[size],
65                      *inpB = new complex<double>[size],
66                      *outR = new complex<double>[size],
67                      *outG = new complex<double>[size],
68                      *outB = new complex<double>[size];
69
70      double maxR = 0.0 , maxG = 0.0, maxB = 0.0;
71      double minR = 0.0 , minG = 0.0, minB = 0.0;
72
73      //Fill inp<R|G|B> arrays
74      for (int i=0; i<size; i++)
75        inpB[i] = pow(-1.0,i/w+i%w) * ((unsigned char*)frame->imageData)[i*3 + 0];
76
77      for (int i=0; i<size; i++)
```

Рис. 36. Утечка памяти

Intel Inspector XE подсвечивает строку, которая инициировала выделение памяти. Действительно, можно заметить, что в конце функции **`ProcessFrame`** происходит удаление всех массивов, кроме **`outB`**. *Добавьте освобождение памяти для указателя* **`outB`** *в строку 213 файла* **filter.cpp**:

```
delete[] outB;
```

Ошибка с идентификатором 1 вызвана несоответствием операций выделения и освобождения памяти. Дело в том, что выделение памяти под массив или под один элемент производится разными реализациями операции **`new`**. Освобождать выделенную память необходимо соответствующим оператором. С помощью оператора **`delete`** необходимо удалять память, которая была выделена под один элемент, а с помощью оператора **`delete[]`** – под массив элементов. Нажмите два раза левой кнопкой мыши на предупреждение об ошибке с идентификатором 1. Детальное описание ошибки (Рис. 37) содержит указание строки, которая инициировала выделение памяти и строки, которая отвечала за освобождение памяти.

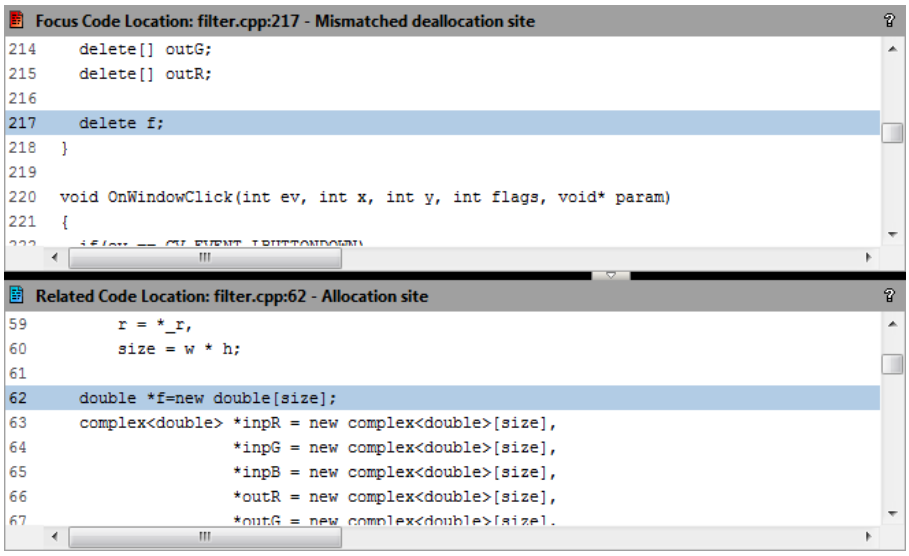


Рис. 37. Освобождение памяти не соответствует выделению

Замените оператор **`delete`** *на оператор* **`delete[]`** *в строке 217 файла* **filter.cpp**:

```
delete[] f;
```

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);

- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 120.8 с (Рис. 38). Таким образом, время выполнения программы сократилось на 6 %, что является хорошим результатом, т.к. внесённые модификации достаточно просты.

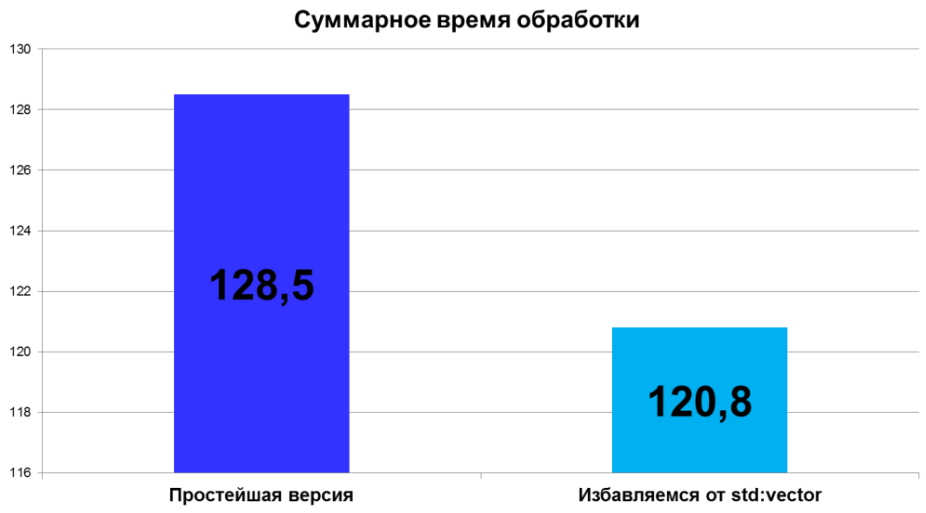


Рис. 38. Прогресс оптимизации

В комплект поставки Intel Parallel Studio XE входит оптимизирующий компилятор Intel. Выполните конвертацию проекта (Рис. 39) для того, чтобы использовать при сборке проекта компилятор Intel C++ Compiler вместо компилятора из поставки Microsoft Visual Studio.

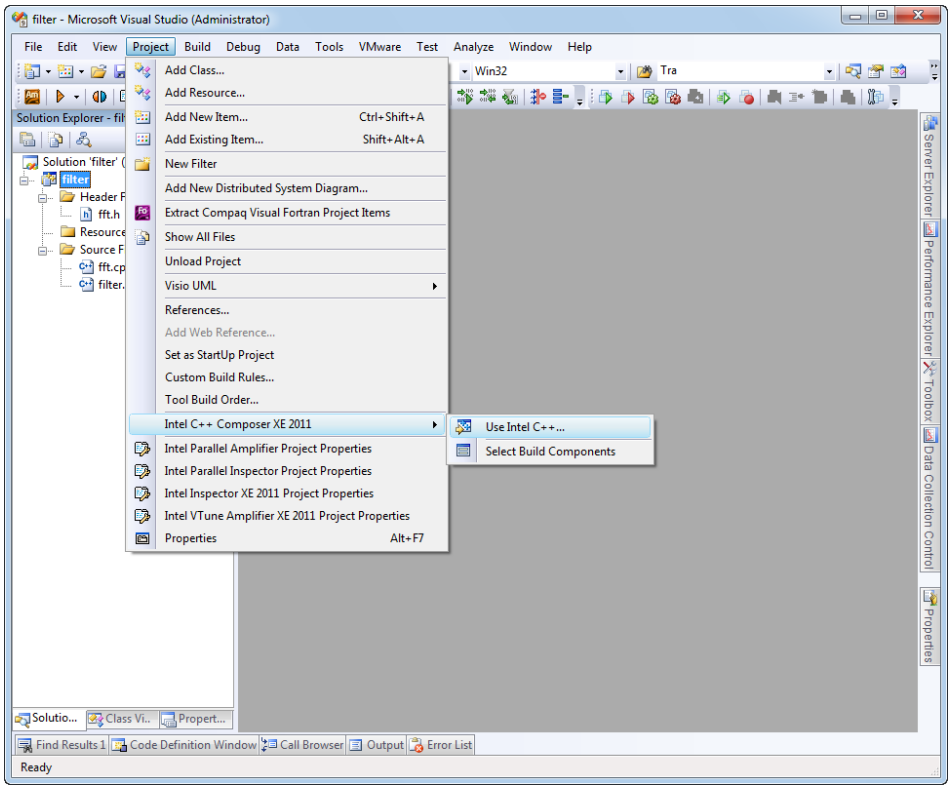


Рис. 39. Конвертация проекта

8. Оптимизация. Возможности компилятора

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\04. filter (Intel power)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\04. filter (Intel power)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 85.6 с (Рис. 40). Использование оптимизирующего компилятора Intel C++ Compiler привело к сокращению времени выполнения программы на 35.2 с. Переход на использование компилятора Intel осуществляется очень просто за счёт использования мастера конвертации проектов. Полученный в результате выигрыш составил 29.1 %.

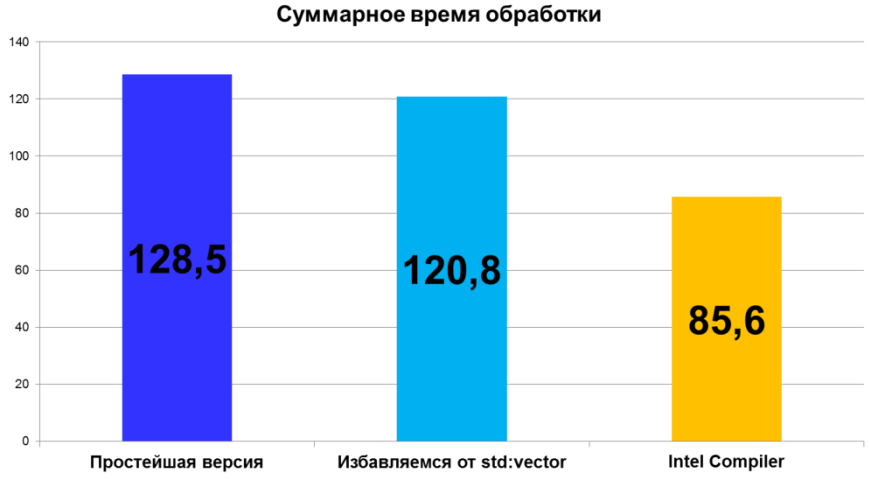


Рис. 40. Прогресс оптимизации

Для принятия решения о направлениях дальнейшей оптимизации выполним поиск горячих точек в программе, используя инструмент Intel VTune Amplifier XE (Рис. 28):

- используйте **Release**-версию программы;
- выберите режим работы **Hotspots** (Рис. 29).

Большую часть времени выполняется вычисление БПФ (Рис. 41). Функция **`Butterfly()`** исчезла из профиля, т.к. компилятор встроил её код непосредственно в функцию **`SerialFFTCalculation()`** для уменьшения объема исполняемого кода.

/Function /Call Stack	CPU Time	Module
SerialFFTCalculation	61.2%	filter.exe
ProcessFrame	15.3%	filter.exe
SerialInverseFFT1D	4.6%	filter.exe
cvWaitKey	4.2%	highgui110.dll
HighGUIProc	4.0%	highgui110.dll
pow	3.1%	filter.exe
SerialFFT2D	2.7%	filter.exe
main	1.8%	filter.exe
icvGetBitmapData	0.8%	highgui110.dll
[Import thunk __libm_sse2_sincos]	0.5%	filter.exe
[AcXtrnal.DLL]	0.4%	AcXtrnal.DLL
Selected 1 row(s):	61.2%	

Рис. 41. Результат профилирования

Для дальнейшего анализа обратимся к ассемблерному листингу программы (Рис. 42). 22.4% времени работы программы занимает выполнение функции по смещению 0x4748. Эта функция присутствует дважды в коде **`SerialFFTCalculation()`** и является первоочередным кандидатом для оптимизации.

Line	Source	CPU T...
37	void SerialFFTCalculation(complex<double> *sig	0.0%
38	{	1.0%
39	if(size==1)	0.2%
40	return;	
41		
42	double const coeff=2.0*PI/size;	0.4%
43		
44	SerialFFTCalculation(signal, first, size/2,	0.6%
45	SerialFFTCalculation(signal, first + size/2	0.2%
46		
47	for (int j=first; j<first+size/2; j++)	0.7%
48	if(forward)	0.4%
49	Butterfly(signal, complex<double>(cos(-j	28.5%
50	else	
51	Butterfly(signal, complex<double>(cos(j*	28.7%
52	}	0.5%
	Highlighted 1 row(s):	

Address	Line	Assembly	CPU Time
0x4300	87	mov dword ptr [esp+0xa0], e	
0x4307	87	mov dword ptr [esp+0xac], e	0.0%
0x430e		**Block 6:**	
0x430e	87	movzx eax, byte ptr [ebp+0x	0.4%
0x4312	87	test eax, eax	0.1%
0x4314	87	jz 0x43e1 <Block 11>	
0x431a		**Block 7:**	
0x431a	87	cvtsi2sd xmm0, dword ptr [e	0.0%
0x4323	87	mulsd xmm0, qword ptr [esp+	0.2%
0x432c	87	call 0x4748	11.4%
0x4331		**Block 8:**	
0x4331	87	movsd qword ptr [esp+0x18],	0.2%
0x4337	87	lea eax, ptr [esp+0x28]	
0x433b	87	mov ecx, dword ptr [esp+0x1	
0x433f	87	movsd qword ptr [esp+0x20],	0.7%
0x4345	87	mov dword ptr [eax], ecx	
		Selected 2 row(s):	22.4%

Рис. 42. Профиль функции **`SerialFFTCalculation`**

Для того чтобы определить, какая функция вызывается по смещению 0x4748, посмотрим на ассемблерный листинг программы в среде разработки Microsoft Visual Studio. Для этого:

- поставьте точку останова (breakpoint) в теле функции **`SerialFFTCalculation()`**, нажав левой кнопкой мыши на серой полосе слева от программного кода (появится красный круглый маркер, Рис. 43);

```
43
44    SerialFFTCalculation( si
45    SerialFFTCalculation( si
46
47  for (int  j=first; j<fir
48    if(forward)
49      Butterfly(signal, cc
50    else
51      Butterfly(signal, cc
52  }
53
```

Рис. 43. Установка точки останова

- нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**;

```
43
44    SerialFFTCalculation( si
45    SerialFFTCalculation( si
46
47  for (int  j=first; j<fir
48    if(forward)
49      Butterfly(signal, cc
50    else
51      Butterfly(signal, cc
52  }
53
```

Рис. 44. Остановка выполнения программы на точке останова

- после того, как выполнение программы остановится на точке останова (Рис. 44), нажмите **Alt+8** или выполните команду **Disassembly** пункта меню **Debug –> Windows** (Рис. 45).

```
  SerialFFTCalculation( signal, first, size/2, step, offset, forward);
  SerialFFTCalculation( signal, first + size/2, size/2, step, offset, forward);

  for (int  j=first; j<first+size/2; j++ )
    if(forward)
0136430E  movzx       eax,byte ptr [forward]
01364312  test        eax,eax
01364314  je          SerialFFTCalculation+1Alh (13643Elh)
      Butterfly(signal, complex<double>(cos(-j*coeff), sin(-j*coeff)), j , size/2, step, offset );
0136431A  cvtsi2sd    xmm0,dword ptr [esp+0ACh]
01364323  mulsd       xmm0,mmword ptr [esp+98h]
0136432C  call        __libm_sse2_sincos (1364748h)
01364331  movsd       mmword ptr [esp+18h],xmm1
01364337  lea         eax,[esp+28h]
0136433B  mov         ecx,dword ptr [esp+18h]
0136433F  movsd       mmword ptr [esp+20h],xmm0
01364345  mov         dword ptr [eax],ecx
01364347  mov         ecx,dword ptr [esp+20h]
0136434B  mov         dword ptr [eax+8],ecx
```

Рис. 45. Ассемблерный листинг функции **`SerialFFTCalculation`**

Функция по смещению 0x4748 выполняет вычисление тригонометрических функций sin и cos, которые передаются в параметры функции **`Butterfly()`**.

9. Оптимизация. Эффективное вычисление тригонометрических функций

9.1. Вещественная арифметика

Многие вычислительные программы требуют использования математических функций, таких как sin, cos, tg, arcsin и др. Востребованность таких функций достаточно высока, поэтому в современных архитектурах процессоров Intel предусмотрена аппаратная реализация наиболее важных математических функций[7]. Для разработки эффективных реализаций вычислительных алгоритмов программисту полезно понимать особенности реализации математических функций. Рассмотрим функцию sin и оценим трудоемкость аппаратной реализации этой функции. Для этого рассмотрим программный код, содержащий явный вызов функции вычисления синуса. Для этого используем вставку на языке ассемблера:

```
double sum = 0.0 ;
for (int i=0; i<COUNT; i++)
{
  double x = i, v = 0.0;
  __asm
  {
    fld x
    fsin
    fstp v
  }

  sum += v;
}
```

На выполнение одной итерации этого цикла требуется 91 такт процессора на тестовой системе (Таблица 1). Это огромная величина для процессора, сопоставимая со временем доступа к оперативной памяти. Примем полученное значение за эталонное.

На практике программист не ведет разработку с использованием вставок ассемблера. Разработчик обычно использует встроенные функции языка программирования, например, из библиотеки **`math.h`**. Библиотеки в большинстве своём не используют аппаратные реализации из блока сопроцессора, т.к. математические функции можно вычислять эффективнее за счёт различных приближений, например, полученных в результате разложения функций в ряды (например, в ряд Тейлора [5]). Программный код,

---

[7] В современных процессорах содержится специализированный блок, позволяющий выполнять вычисления с вещественной арифметикой. Этот блок называется математическим сопроцессором.

содержащий вычисление функции sin с использованием библиотеки **math.h**, представлен ниже:

```
double sum = 0.0;
for (int i=0; i<COUNT; i++)
  sum += sin((double)i);
```

Т.к. функция sin реализуется за счёт приближеных вычислений, то у разработчика есть возможность управления точностью. В зависимости от настроек авторы библиотек математических функций гарантируют разное число верных знаков в результате вычисления функции. Увеличение точности приводит к существенному росту вычислительных затрат. Для одних приложений (научные и инженерные расчеты) повышенная точность является критичной для получения верного результата, для некоторых других (визуализация в играх) – не играет определяющей роли, а на первый план выходит время расчетов. В настройках проекта Microsoft Visual Studio предусмотрены специальные ключи компилятора, которые определяют используемую модель вещественной арифметики (Рис. 46).

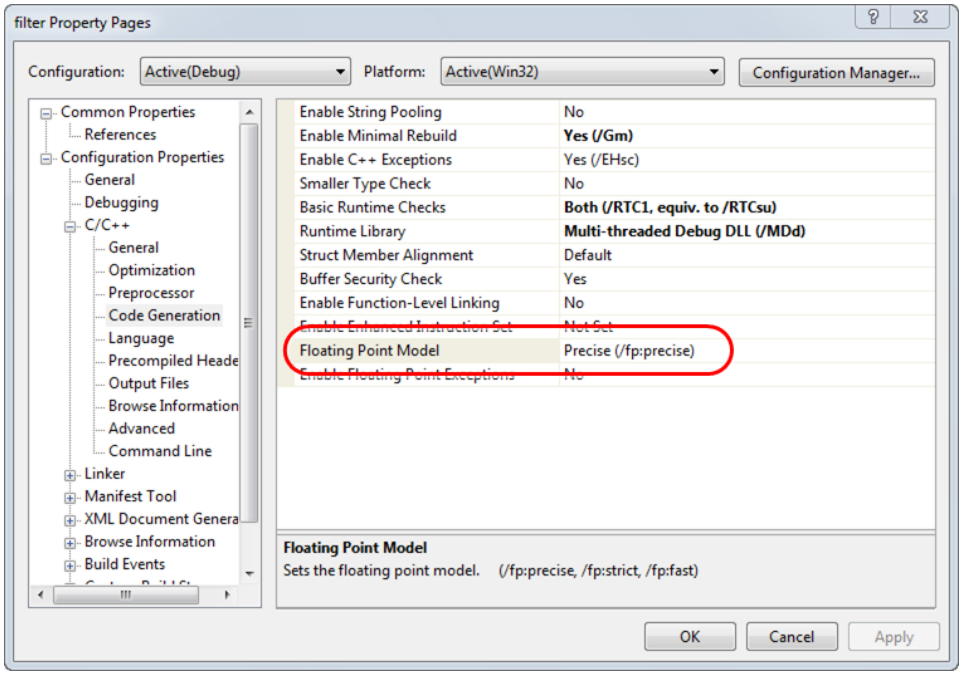


Рис. 46. Выбор модели вычислений

Компилятор Microsoft Visual Studio поддерживает следующие модели вычислений[8]:

- **`/fp:precise`** – модель, соответствующая большинству требований, предъявляемых к приложениям с плавающей запятой (в частности, в этом режиме выполняется упрощение выражений, встречающихся в программе);
- **`/fp:strict`** – самая строгая модель с плавающей запятой;
- **`/fp:fast`** – модель, позволяющая генерировать самую быструю версию кода.

Если производительность является самым важным требованием, лучше использовать ключ **`/fp:fast`**.

В таблицах ниже представлены результаты экспериментов со всеми моделями с использованием компиляторов Microsoft Visual Studio и Intel C++ Compiler, входящего в поставку Parallel Studio XE. Компилятор Intel поддерживает все ключи, приведённые выше, и один дополнительный – **`/fp:fast2`**, который выполняет более агрессивную оптимизацию.

Таблица 2. Время вычисления функции **`sin`** (Microsoft VS)

Ключ **`/fp:precise`**, тактов	Ключ **`/fp:strict`**, тактов	Ключ **`/fp:fast`**, тактов
126	119	89

Таблица 3. Эффективность вычисления функции **`sin`** (Intel Compiler)

Ключ **`/fp:precise`**, тактов	Ключ **`/fp:strict`**, тактов	Ключ **`/fp:fast`**, тактов	Ключ **`/fp:fast2`**, тактов
122	123	41	40

Погрешность вычислений в рассматриваемых примерах была очень маленькой даже при использовании менее точных моделей, различия наблюдались только в 14 знаке переменной **`sum`**. Применение оптимизирующего компилятора Intel может значительно ускорить программу при использовании ключа **`/fp:fast`**.

Заметим, что в данной работе приведено лишь краткое введение в большую и сложную тему – вычисление математических функций. Так, деталь-

---

[8] По материалам документации MSDN [http://msdn.microsoft.com/ru-ru/library/e7s85ffb%28v=vs.90%29.aspx].

ного рассмотрения заслуживает вопрос точности вычислений и соответствия стандартам. Также может быть продолжено изучение вопросов, связанных с производительностью реализаций математических функций. Желающие могут выполнить сравнение скорости работы функций сопроцессора (fsin и другие), функций из библиотек LibM компиляторов Microsoft и Intel (используются при подключении библиотеки **math.h**), функций из библиотеки Short Vector Math Library (SVML) – составной части Intel C++ Compiler, вызов которых производится при векторизации циклов, и, наконец, функций из библиотеки Intel Math Kernel Library, вернее ее части – Vector Math Library (VML), позволяющих вычислять значения для набора аргументов, лежащих в буфере оперативной памяти. При этом рекомендуется управлять точностью, чтобы обеспечить корректность сравнения.

9.2. Эффективное вычисление тригонометрических функций

На предыдущем этапе с помощью профилировщика Intel VTune Amplifier XE было обнаружено, что вычисление функций sin и cos происходит очень долго (Рис. 42).

Предыдущий этап оптимизации состоял в использовании компилятора Intel C++ Compiler. Это дало существенное ускорение в первую очередь за счёт того, что компилятор Intel использует ключ **`/fp:fast`** по умолчанию в **Release** версии проекта[9]. Таким образом, все возможности простой компиляторной оптимизации исчерпаны. Дальнейшее ускорение вычислений функций sin и cos может быть продолжено только за счёт алгоритмической оптимизации. Вспомним тригонометрические тождества из школьного курса:

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta) \tag{22}$$

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) \tag{23}$$

На каждой итерации алгоритма БПФ необходимо вычислять $\cos(\frac{2k\pi}{N})$ и $\sin(\frac{2k\pi}{N})$. Зафиксируем N и воспользуемся формулами (22) и (23), полагая, что $\alpha = \frac{2(k-1)\pi}{N}$, $\beta = \frac{2\pi}{N}$, получаем:

$$\sin\left(\frac{2k\pi}{N}\right) = \sin\left(\frac{2(k-1)\pi}{N}\right)\cos\left(\frac{2\pi}{N}\right) + \cos\left(\frac{2(k-1)\pi}{N}\right)\sin\left(\frac{2\pi}{N}\right) \tag{24}$$

[9] Исходя из нашего опыта, в расчетных программах, активно использующих математические функции, переход к использованию Intel C/C++ Compiler чаще всего приводит к ускорению в 1,5 – 2,5 раза по сравнению с Microsoft C++ Compiler и gcc.

$$\cos\left(\frac{2k\pi}{N}\right) = \cos\left(\frac{2(k-1)\pi}{N}\right)\cos\left(\frac{2\pi}{N}\right) - \sin\left(\frac{2(k-1)\pi}{N}\right)\sin\left(\frac{2\pi}{N}\right) \qquad (25)$$

Для вычисления синуса и косинуса по формулам (24) и (25) достаточно знать значения этих функций на предыдущей итерации счётчика p. Таким образом, вместо того, чтобы выполнять трудоёмкие расчёты тригонометрических функций, можно выполнить два умножения и одно сложение. Значения коэффициентов поворота $\cos\left(\frac{2\pi}{N}\right)$ и $\sin\left(\frac{2\pi}{N}\right)$ достаточно посчитать один раз, т.к. N фиксированно.

При вычислении БПФ в цикле по **`j`** от **`first`** до **`first+size/2`** необходимо посчитать все значения **`sin(j*coeff)`** и **`cos(j*coeff)`**, где **`coeff=2.0*PI/size`**. Модифицированный алгоритм будет иметь следующий вид:

1. Вычисляем первые значения синуса и косинуса: **`sin(first*coeff)`** и **`cos(first*coeff)`**.
2. Вычисляем коэффициенты поворота, не зависящие от итерации цикла: **`sin(coeff)`** и **`cos(coeff)`**.
3. В цикле по **`j`** для вычисления **`sin(j*coeff)`** и **`cos(j*coeff)`** будем использовать значения синуса и косинуса на итерации **`j-1`**: **`sin((j-1)*coeff)`**, **`cos((j-1)*coeff)`** и коэффициенты поворота в соответствии с ранее выведенными соотношениями.

Внесите изменения в функцию **`SerialFFTCalculation`** *в файле* **fft.cpp** *в соответствии с приведённым выше алгоритмом*:

```
void SerialFFTCalculation(complex<double> *signal,
  int first, int size, int step, int offset,
  bool forward=true)
{
  if(size==1)
    return;

  double const coeff=2.0*PI/size;

  //Reverse computing
  double wR = cos(coeff),
         wI = sin(coeff);
  double uR = cos(coeff * first),
         uI = sin(coeff * first);

  //Forward computing
  double wRf = wR,
         wIf = -wI;
  double uRf = uR,
         uIf = -uI;
```

```
  double uRtmp;

  SerialFFTCalculation(signal, first, size/2,
                       step, offset, forward);
  SerialFFTCalculation(signal, first + size/2,
                      size/2, step, offset, forward);

  for (int  j=first; j<first+size/2; j++)
    if(forward)
    {
      Butterfly(signal, complex<double>(uRf, uIf),
                j , size/2, step, offset);
      uRtmp=uRf;
      uRf=uRf*wRf-uIf*wIf;
      uIf=uIf*wRf+uRtmp*wIf;
    }
    else
    {
      Butterfly(signal, complex<double>(uR, uI),
                j , size/2, step, offset);
      uRtmp=uR;
      uR=uR*wR-uI*wI;
      uI=uI*wR+uRtmp*wI;
    }
}
```

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\05. filter (fastest sin)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\05. filter (fastest sin)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 77с. (Рис. 47), т.е. был получен выигрыш в 10%.

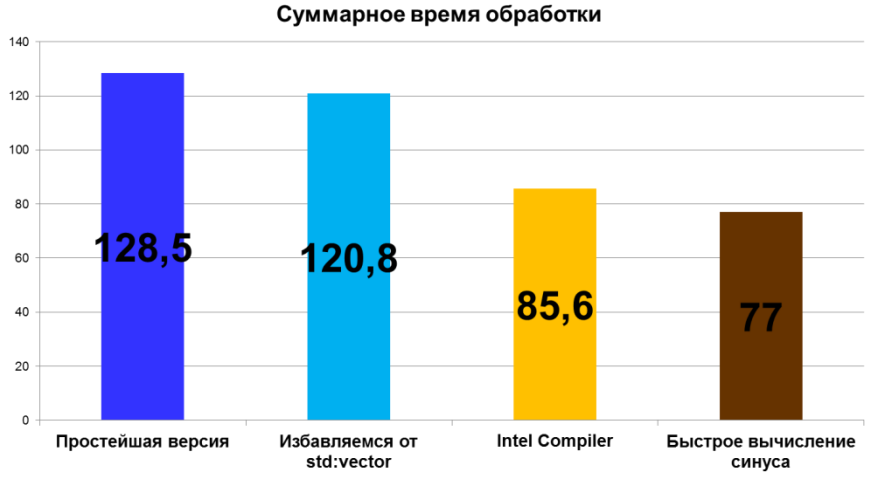


Рис. 47. Прогресс оптимизации

Достаточно низкое ускорение, полученное после оптимизации вычисления тригонометрических функций, вызвано в первую очередь тем, что в функции **`SerialFFTCalculation()`** сохраняется значительное количество вычислений sin и cos (Рис. 48).

```
37 void SerialFFTCalculation(complex<double> *signal, int first, int size, int step, int offset, bool forward=true)
38 {
39   if(size==1)
40     return;
41
42   double const coeff=2.0*PI/size;
43
44   //Reverse computing
45   double wR = cos(coeff),
46          wI = sin(coeff);
47
48   double uR = cos(coeff * first),
49          uI = sin(coeff * first);
50
51   //Forward computing
52   double wRf = wR,
53          wIf = -wI;
54
55   double uRf = uR,
56          uIf = -uI;
```

Рис. 48. Тригонометрические функции в **`SerialFFTCalculation`**

В исходной версии проекта требовалось $n \cdot \log n$ вычислений синусов и косинусов, где n – размер массива для выполнения одномерного БПФ. То есть для $n = 256$ потребуется вычислить 2048 синусов и косинусов. В модифицированной версии требуется $4 \cdot (n - 1)$ вычислений синусов и косинусов, где n – размер массива для выполнения одномерного БПФ. То есть для $n = 256$ потребуется вычислить 1020 синусов и косинусов. Количество вычислений тригонометрических функций сократилось почти в 2 раза. Для больших значений n эффективность модифицированной версии будет ещё выше.

Для принятия решения о направлении дальнейшей оптимизации выполним поиска горячих точек в программе, используя инструмент Intel VTune Amplifier XE (Рис. 28):

- используйте **Release**-версию программы;
- выберите режим работы **Hotspots** (Рис. 29).

Снова большую часть времени выполняется функция вычисления БПФ (Рис. 49).

/Function /Call Stack	CPU Time	Module
SerialFFTCalculation	58.2%	filter.exe
ProcessFrame	16.4%	filter.exe
SerialInverseFFT1D	5.1%	filter.exe
HighGUIProc	4.6%	highgui110.dll
cvWaitKey	4.4%	highgui110.dll
pow	3.3%	filter.exe
SerialFFT2D	2.4%	filter.exe
main	1.9%	filter.exe
icvGetBitmapData	0.7%	highgui110.dll
cvSetTrackbarPos	0.6%	highgui110.dll
icvFlipVert_8u_C1R	0.5%	cxcore110.dll
Selected 1 row(s):	58.2%	

Рис. 49. Результат профилирования

Функция **`Butterfly()`** – наиболее часто вызываемая функция. На её выполнение расходуется около 40% времени работы программы (Рис. 50).

Line	Source	CPU Time
62		
63	for (int j=first; j<first+size/2; j++)	0.5%
64	if(forward)	0.6%
65	{	
66	Butterfly(signal, complex<double>(uRf, uIf), j , size/2, step, offset);	20.3%
67	uRtmp=uRf;	0.2%
68	uRf=uRf*wRf-uIf*wIf;	0.3%
69	uIf=uIf*wRf+uRtmp*wIf;	0.2%
70	}	
71	else	
72	{	
73	Butterfly(signal, complex<double>(uR, uI), j , size/2, step, offset);	19.1%
74	uRtmp=uR;	0.1%
75	uR=uR*wR-uI*wI;	0.4%
76	uI=uI*wR+uRtmp*wI;	0.2%
77	}	
78	}	0.9%
	Selected 1 row(s):	

Рис. 50. Профиль функции **`SerialFFTCalculation`**

Обратим внимание на основной вычислительный цикл функции **`SerialFFTCalculation()`**. На каждой итерации этого цикла выполняется проверка условия, которое определяет, какое ПФ вычисляется: прямое или обратное. Совершенно очевидно, что такая проверка не зависит от итерации цикла и может быть выполнена перед циклом. Более того такую проверку можно полностью устранить, заметив что вычисление прямого и обратного ПФ отличаются только знаком переменных **`wI`**, **`uI`** и **`wIf`**, **`uIf`**.

Избавьтесь от проверки условия внутри цикла в функции **`SerialFFTCalculation()`** *в файле* **fft.cpp**:

```
void SerialFFTCalculation(complex<double> *signal,
  int first, int size, int step, int offset,
  bool forward=true)
{
  if(size==1)
    return;

  double const coeff=2.0*PI/size;
  double wR = cos(coeff),
         wI = sin(coeff);
  double uR = cos(coeff * first),
         uI = sin(coeff * first);
  double uRtmp;

  if(forward)
  {
    wI *= -1;  uI *= -1;
  }

  SerialFFTCalculation(signal, first, size/2, step,
                       offset, forward);
  SerialFFTCalculation(signal, first + size/2, size/2,
                       step, offset, forward);
```

```
  for (int  j=first; j<first+size/2; j++ )
  {
    Butterfly(signal, complex<double>(uR, uI), j , size/2,
              step, offset );
    uRtmp=uR;
    uR=uR*wR-uI*wI;
    uI=uI*wR+uRtmp*wI;
  }
}
```

10. Оптимизация. Избавление от лишних ветвлений

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\06. filter (excess branch)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\06. filter (excess branch)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке Проводника Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 75.9с (Рис. 51), т.е. был получен выигрыш в 1.4%.

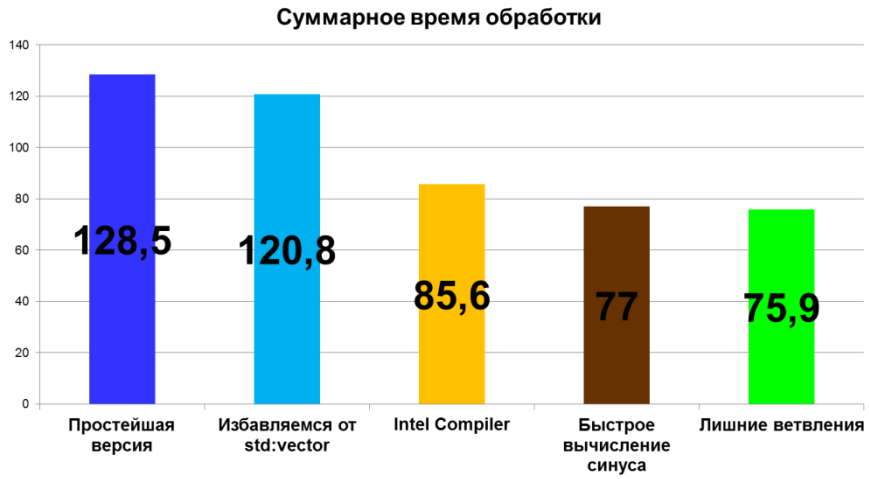


Рис. 51. Прогресс оптимизации

Невысокое ускорение вызвано тем, что глубина рекурсии функции **`SerialFFTCalculation()`** достаточно маленькая, так как обрабатываемый сигнал одномерного БПФ содержит всего 256 точек (для используемого тестового видео). При обработке изображения большего размера выигрыш от выполненной оптимизации будет больше.

Используя инструмент Intel VTune Amplifier XE (Рис. 28), убедитесь, что функция вычисления БПФ является наиболее «горячей» точкой программы:

- используйте **Release**-версию программы;
- выберите режим работы **Hotspots** (Рис. 29).

Дальнейший шаг оптимизации функции вычисления БПФ будет заключаться в реализации ее итеративной версии. Каждый алгоритм, сформулированный рекурсивно, может быть реализован итеративно[10]. Часто итеративная версия проще для понимания и оптимизации, а также избавлена от накладных расходов на организацию рекурсии и ряда сопутствующих проблем.

Реализуйте итеративный алгоритм БПФ (раздел 2.3.4). Избавьтесь от «лишних» вызовов функций (встройте функции **`Butterfly`***) и реализуйте отдельно вычислительную часть прямого и обратного одномерного БПФ:*

```
void SerialFFTCalculation(complex<double> *mas, int size,
  int step, int off)
```

[10] Исходя из нашего опыта, устранение рекурсии в среднем приносит выигрыш около 20%.

```
{
  int m=0;
  for(int tmp_size = size; tmp_size>1; tmp_size/=2,
      m++ );//size = 2^m

  complex<double> *_mas = mas + off;

  for (int p=0; p<m; p++ )
  {
    int butterflyOffset = 1 << (p+1) ;
    int butterflySize = butterflyOffset >> 1 ;

    double alpha=-2.0*PI/butterflyOffset;
    double wR=cos( alpha  ), wI=sin( alpha );

    for (int i=0; i<size/butterflyOffset; i++ )
    {
      double uR=1.0, uI=0.0;
      double uRtmp;

      int offset = i * butterflyOffset;

      for (int  j=0; j<butterflySize; j++ )
      {
        complex<double> tem =
           _mas[(j + offset +  butterflySize)*step] *
           complex<double>(uR, uI);

        _mas[(j + offset + butterflySize)*step] =
           _mas[(j + offset)*step] - tem;

        _mas[(j + offset)*step] += tem;

        uRtmp=uR;
        uR=uR*wR-uI*wI;
        uI=uI*wR+uRtmp*wI;
      }
    }
  }
}

void SerialInverseFFTCalculation(complex<double> *mas,
  int size, int step, int off)
{
  int m=0;
  for(int tmp_size = size; tmp_size>1; tmp_size/=2,
      m++ );//size = 2^m

  complex<double> *_mas = mas + off;
```

```
  for (int p=0; p<m; p++ )
  {
    int butterflyOffset = 1 << (p+1) ;
    int butterflySize = butterflyOffset >> 1 ;

    double alpha=2.0*PI/butterflyOffset;
    double wR=cos( alpha  ), wI=sin( alpha );

    for (int i=0; i<size/butterflyOffset; i++ )
    {
      double uR=1.0, uI=0.0;
      double uRtmp;

      int offset = i * butterflyOffset;

      for (int  j=0; j<butterflySize; j++ )
      {
        complex<double> tem =
           _mas[(j + offset +  butterflySize)*step] *
           complex<double>(uR, uI);

        _mas[(j + offset + butterflySize)*step] =
           _mas[(j + offset)*step] - tem;

        _mas[(j + offset)*step] += tem;

        uRtmp=uR;
        uR=uR*wR-uI*wI;
        uI=uI*wR+uRtmp*wI;
      }
    }
  }
}
```

Внесите изменения в функции вычисления прямого и обратного БПФ:

```
void SerialFFT1D(complex<double> *inputSignal,
  complex<double> *outputSignal,
  int size, int step, int offset)
{
  BitReversing(inputSignal, outputSignal, size,
               step, offset);
  SerialFFTCalculation(outputSignal, size, step, offset);
}

void SerialInverseFFT1D(complex<double> *inputSignal,
  complex<double> *outputSignal,
  int size, int step, int offset)
{
  BitReversing(inputSignal, outputSignal, size,
```

```
                step, offset);
  SerialInverseFFTCalculation(outputSignal, size,
                              step, offset);

  complex<double> _size(size);

  for (int  j=0; j<size; j++ )
    outputSignal[offset + step*j] /= _size;
}
```

11. Оптимизация. Итеративная реализация алгоритма

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\07. filter (kill recursion)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\07. filter (kill recursion)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 60.9с (Рис. 52). Был получен выигрыш в 19.6%. Во

многом это связано с тем, что в итеративной версии практически полностью отсутствует вычисление тригонометрических функций.

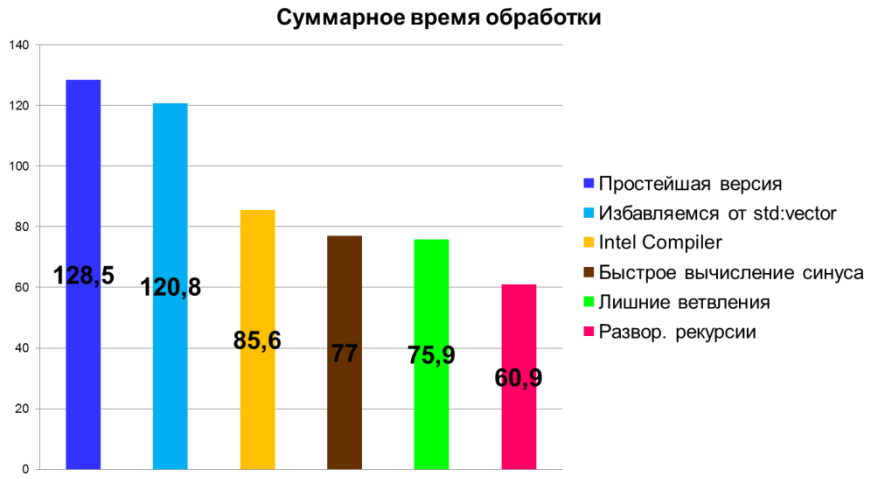


Рис. 52. Прогресс оптимизации

Для принятия решения о направлении дальнейшей оптимизации выполним поиск «горячих точек» в программе, используя инструмент Intel VTune Amplifier XE (Рис. 28):

- используйте **Release**-версию программы;
- выберите режим работы **Hotspots** (Рис. 29).

Достаточно предсказуемо выглядит результат работы профилировщика. Функции, выполняющие БПФ, работают наибольшее время (Рис. 53).

/Function /Call Stack	CPU Time	Module
SerialInverseFFT1D	29.7%	filter.exe
SerialFFT2D	29.1%	filter.exe
ProcessFrame	20.6%	filter.exe
cvWaitKey	5.5%	highgui110.dll
HighGUIProc	5.2%	highgui110.dll
pow	3.7%	filter.exe
main	2.9%	filter.exe
icvGetBitmapData	1.0%	highgui110.dll
[AcXtrnal.DLL]	0.7%	AcXtrnal.DLL
icvFlipVert_8u_C1R	0.5%	cxcore110.dll
SerialInverseFFT2D	0.4%	filter.exe
Selected 1 row(s):	29.7%	

Рис. 53. Результат профилировки

Профиль не даёт достаточно информации для анализа, поэтому обратимся к ассемблерному листингу функции **`SerialInverseFFT1D()`** (Рис. 54). 20.6% времени работы программы выполнялись две функции по смещениям 0x7080 и 0x707c. Эти функции присутствуют в профиле по 2 раза.

Address	Line	Assembly	CPU Time
0x43e3	136	movsd qword ptr [ebp-0xa0],	0.0%
0x43eb	136	movsd qword ptr [ebp-0x98],	
0x43f3	136	push edx	0.1%
0x43f4	136	call dword ptr [0x40707c]	8.9%
0x43fa		**Block 28:**	
0x43fa	136	mov eax, dword ptr [ebp-0x4	0.1%
0x43fd	136	lea edx, ptr [ebp-0x90]	
0x4403	136	push edx	
0x4404	136	lea ecx, ptr [eax+ebx*1]	
0x4407	136	mov dword ptr [ebp-0x44], e	0.2%
0x440a	136	push ecx	
0x440b	136	lea eax, ptr [ebp-0x80]	
0x440e	136	push eax	
0x440f	136	call dword ptr [0x407080]	3.4%
0x4415		**Block 29:**	
0x4415	136	add esp, 0x18	0.1%
		Selected 4 row(s):	20.6%

Рис. 54. Профиль функции **`SerialInverseFFT1D`**

Для того чтобы определить, какие функции выполняются по смещениям 0x7080 и 0x707c, посмотрим на ассемблерный листинг программы в среде разработки Microsoft Visual Studio. Для этого:

- поставьте точку останова (breakpoint) в теле функции **`SerialInverseFFT1D()`**, нажав левой кнопкой мыши на серой полосе слева от программного кода (появится красный круглый маркер);
- нажмите клавишу **F5** или выполните команду **Start Debugging** пункта меню **Debug**;
- после того как выполнение программы остановится на точке останова, нажмите **Alt+8** или выполните команду **Disassembly** пункта меню **Debug –> Windows** (Рис. 55).

```
00CC43E3  movsd       mmword ptr [ebp-0A0h],xmm0
00CC43EB  movsd       mmword ptr [ebp-98h],xmm1
00CC43F3  push        edx
00CC43F4  call        dword ptr [__imp_std::operator*<double> (0CC707Ch)]
00CC43FA  mov         eax,dword ptr [ebp-48h]
00CC43FD  lea         edx,[ebp-90h]
00CC4403  push        edx
00CC4404  lea         ecx,[eax+ebx]
00CC4407  mov         dword ptr [ebp-44h],ecx
00CC440A  push        ecx
00CC440B  lea         eax,[ebp-80h]
00CC440E  push        eax
00CC440F  call        dword ptr [__imp_std::operator-<double> (0CC7080h)]
00CC4415  add         esp,18h
00CC4418  movsd       xmm1,mmword ptr [ebp-78h]
00CC441D  inc         edi
```

Рис. 55. Ассемблерный листинг **`SerialInverseFFT1D`**

Функции по смещениям 0x7080 и 0x707c являются операторами умножения и вычитания класса **`std::complex`**. Компилятор не в состоянии выполнить оптимизации для сложных типов данных. Наиболее удобными для компиляторной оптимизации являются встроенные типы данных.

Ранее мы отказались от использования контейнера **`std::vector`** и это привело к ощутимому приросту производительности. Поступим таким же образом с классом, представляющим комплексные числа, – **`std::complex`**. Будем представлять комплексное число в виде последовательности из двух элементов типа **`double`** (Рис. 56).

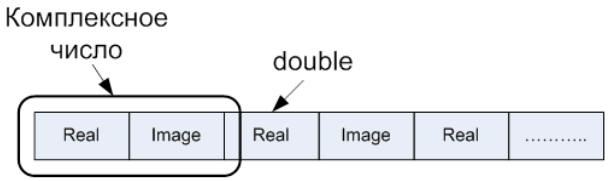


Рис. 56. Представление комплексного числа в массиве элементов типа **`double`**

Прототипы функций в файле **fft.h** *примут следующий вид*:

```
void SerialFFT2D(double *inputSignal, double *outputSignal,
  int w, int h);

void SerialInverseFFT2D(double *inputSignal,
  double *outputSignal, int w, int h);
```

Прототипы функций в файле **fft.cpp** *примут следующий вид*:

```
…
void BitReversing(double *inputSignal,
  double *outputSignal, int size, int step, int offset)
…

void Butterfly(double *signal, complex<double> u,
  int offset, int butterflySize, int step, int off)
…

void SerialFFTCalculation(double *signal,
  int first, int size, int step,
  int offset, bool forward=true)
…

void SerialFFT1D(double *inputSignal,
  double *outputSignal, int size, int step, int offset)
…

void SerialInverseFFT1D(double *inputSignal,
  double *outputSignal, int size, int step, int offset)
…
```

Т.к. оператор индексирования в предыдущей версии возвращал объект, содержащий действительную и мнимую часть комплексного числа, то в модифицированной версии придётся поменять индексацию. Внесите изменения в функцию **BitReversing** *в файле* **fft.cpp***:*

```
void BitReversing(double *inputSignal,
  double *outputSignal, int size, int step, int offset)
{
  …

    outputSignal[2*revInd*step + 2*offset] =
       inputSignal[2*ind*step + 2*offset];

    outputSignal[2*revInd*step + 2*offset + 1] =
       inputSignal[2*ind*step + 2*offset + 1];
  }
}
```

Внесите изменения в функции **SerialFFTCalculation** *и* **SerialInverseFFTCalculation** *в файле* **fft.cpp***:*

```
void SerialFFTCalculation(double *mas, int size, int step,
  int off)
{
  int m=0;
  for(int tmp_size = size; tmp_size>1; tmp_size/=2,
      m++ );//size = 2^m

  double *_mas = mas + 2*off;

  for (int p=0; p<m; p++ )
  {
    int butterflyOffset = 1 << (p+1) ;
    int butterflySize = butterflyOffset >> 1 ;

    double alpha=-2.0*PI/butterflyOffset;
    double wR=cos( alpha  ), wI=sin( alpha );

    double uRtmp;
    double temI;

    for (int i=0; i<size/butterflyOffset; i++ )
    {
      double uR=1.0, uI=0.0;

      int offset = i * butterflyOffset;

      for (int  j=0; j<butterflySize; j++ )
      {
```

```
        double temR;
        temR =
          _mas[2 * (j + offset + butterflySize)*step] *
          uR -
          _mas[2 * (j + offset + butterflySize)*step + 1] *
          uI;

        temI =
          _mas[2 * (j + offset + butterflySize)*step] *
          uI +
          _mas[2 * (j + offset + butterflySize)*step + 1] *
          uR;

        _mas[2 * (j + offset +  butterflySize)*step] =
           _mas[2 * (j + offset)*step] - temR;
        _mas[2 * (j + offset +  butterflySize)*step + 1] =
           _mas[2 * (j + offset)*step + 1] - temI;

        _mas[2 * (j + offset)*step] += temR;
        _mas[2 * (j + offset)*step + 1] += temI;

        uRtmp=uR;
        uR=uR*wR-uI*wI;
        uI=uI*wR+uRtmp*wI;
      }
    }
  }
}

void SerialInverseFFTCalculation(double *mas, int size,
  int step, int off)
{
  int m=0;
  for(int tmp size = size; tmp size>1; tmp size/=2,
      m++ );//size = 2^m

  double *_mas = mas + 2*off;

  …
      for (int  j=0; j<butterflySize; j++ )
      {
        double temR =
          _mas[2 * (j + offset + butterflySize)*step] *
          uR -
          _mas[2 * (j + offset + butterflySize)*step + 1] *
          uI,
               temI =
          _mas[2 * (j + offset + butterflySize)*step] *
          uI +
          _mas[2 * (j + offset + butterflySize)*step + 1] *
```

```
          uR;

        _mas[2 * (j + offset +  butterflySize)*step] =
          _mas[2 * (j + offset)*step] - temR;
        _mas[2 * (j + offset +  butterflySize)*step + 1] =
          _mas[2 * (j + offset)*step + 1] - temI;

        _mas[2 * (j + offset)*step] += temR;
        _mas[2 * (j + offset)*step + 1] += temI;

        uRtmp=uR;
        uR=uR*wR-uI*wI;
        uI=uI*wR+uRtmp*wI;
      }
  …
}
```

Внесите изменения в функции **SerialInverseFFT1D**, **SerialInverseFFT2D** *и* **SerialFFT2D** *в файле* **fft.cpp**:

```
void SerialInverseFFT1D(double *inputSignal,
  double *outputSignal, int size, int step, int offset)
{
  …

  for (int  j=0; j<size; j++ )
  {
    outputSignal[2*(offset+j*step)]/=_size;
    outputSignal[2*(offset+j*step)+1]/=_size;
  }
}

void SerialInverseFFT2D(double *inputSignal,
  double *outputSignal, int w, int h)
{
  double *tem=new double[2*w*h];

  …
}

void SerialFFT2D(double *inputSignal, double *outputSignal,
  int w, int h)
{
  double *tem = new double[2*w*h];

  …
}
```

В файле **filter.cpp** *выполните следующие замены в функции* **`ProcessFrame`**:

- **`outR.real()`** *на* **`outR[2*i]`**;
- **`outR.imag()`** *на* **`outR[2*i+1]`**;
- **`outG.real()`** *на* **`outG[2*i]`**;
- **`outG.imag()`** *на* **`outG[2*i+1]`**;
- **`outB.real()`** *на* **`outB[2*i]`**;
- **`outB.imag()`** *на* **`outB[2*i+1]`**;
- **`inpR.real()`** *на* **`inpR[2*i]`**;
- **`inpG.real()`** *на* **`inpG[2*i]`**;
- **`inpB.real()`** *на* **`inpB[2*i]`**.

Внесите изменения в функцию **`ProcessFrame`** *в файле* **filter.cpp**:

```
void ProcessFrame(IplImage *frame, IplImage *outFrame,
  IplImage* filter, IplImage* freq, FilterType t, int * r)
{
  int w = frame->width,
      h = frame->height,
      r = *_r,
      size = w * h;

  double *f=new double[size];
  double *inpR = new double[2*size],
         *inpG = new double[2*size],
         *inpB = new double[2*size],
         *outR = new double[2*size],
         *outG = new double[2*size],
         *outB = new double[2*size];

  double maxR = 0.0 , maxG = 0.0, maxB = 0.0;
  double minR = 0.0 , minG = 0.0, minB = 0.0;

  //Fill inp<R|G|B> arrays
  for (int i=0; i<size; i++)
  {
    inpB[2*i] = pow(-1.0,i/w+i%w) * ((unsigned char*)
                                  frame->imageData)[i*3 + 0];
    inpB[2*i+1] = 0.0;
  }

  for (int i=0; i<size; i++)
  {
    inpG[2*i] = pow(-1.0,i/w+i%w) * ((unsigned char*)
```

```
                                    frame->imageData)[i*3 + 1];
    inpG[2*i+1] = 0.0;
  }

  for (int i=0; i<size; i++)
  {
    inpR[2*i] = pow(-1.0,i/w+i%w) * ((unsigned char*)
                                    frame->imageData)[i*3 + 2];
    inpR[2*i+1] = 0.0;
  }

  …

  //Apply filter
  for (int i=0; i<size; i++)
  {
    outR[2*i] = outR[2*i] * f[i];
    outR[2*i+1] = outR[2*i+1] * f[i];
  }

  for (int i=0; i<size; i++)
  {
    outG[2*i] = outG[2*i] * f[i];
    outG[2*i+1] = outG[2*i+1] * f[i];
  }

  for (int i=0; i<size; i++)
  {
    outB[2*i] = outB[2*i] * f[i];
    outB[2*i+1] = outB[2*i+1] * f[i];
  }

  …
}
```

12. Оптимизация. Использование встроенных типов данных

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\08. filter (eliminate complex)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;

- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\08. filter (eliminate complex)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: "**filter.exe ..\Videos\TestVideo1.avi -s -gl**". Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите "**filter.exe ..\Videos\TestVideo1.avi -s -gl**" и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 29.4с (Рис. 57). Был получен выигрыш в 51.7%. Такое большое ускорение было получено в первую очередь за счёт того, что при использовании простых типов данных компилятор смог выполнить ряд эффективных оптимизаций кода.

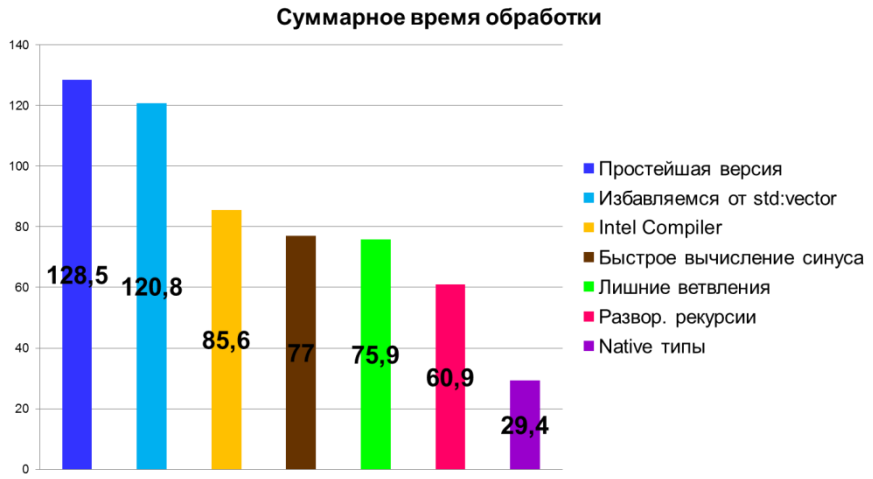


Рис. 57. Прогресс оптимизации

Вычисление одномерного БПФ состоит из двух этапов: перестановка элементов и непосредственно вычисления (Рис. 58).

```
66 void SerialFFT1D(complex<double> *inputSignal, complex<double> *outputSignal, int size, int step, int offset)
67 {
68   BitReversing(inputSignal, outputSignal, size, step, offset);
69   SerialFFTCalculation(outputSignal, size, step, offset);
70 }
71
```

Рис. 58. Функция **`SerialFFT1D`**

Оценим “вклад” перестановки элементов в вычисление БПФ с помощью Intel VTune Amplifier XE (Рис. 28):

- используйте **Release**-версию программы;
- выберите режим работы **Hotspots** (Рис. 29).

Функция, выполняющая перестановку элементов массива с помощью бит-реверсирования, занимает почти половину времени вычисления БПФ (Рис. 59), поэтому она является следующим кандидатом на оптимизацию.

Line	Source	CPU Time
70		
71	void SerialFFT1D(double *inputSignal, double *outputSignal, int size, int step, int offset)	
72	{	
73	BitReversing(inputSignal, outputSignal, size, step, offset);	10.4%
74	SerialFFTCalculation(outputSignal, size, step, offset);	11.5%
75	}	
76		
	Selected 1 row(s):	

Рис. 59. Профиль функции **`SerialFFT1D`**

13. Оптимизация. Предвычисления

Бит-реверсирование – это преобразование двоичного числа, заключающееся в изменении порядка следования бит на противоположный. В пункте 2.3.2 была представлена реализация этого алгоритма в виде цикла. Таким образом, для пересчёта каждого индекса необходимо выполнить цикл из n итераций, где n – это минимальное количество бит, необходимое для хранения индексов массива (если размер $size$ обрабатываемого массива является степень двойки, то $n = \log_2 size$). Такой алгоритм малоэффективен.

Вместо алгоритмического вычисления реверсивного значения индекса при выполнении этапа перестановки предлагается заранее посчитать ряд значений и во время реверсирования подставлять посчитанные значения. Рассматриваемый алгоритм будет состоять из следующих шагов:

- заведём таблицу, в которой будут содержаться все однобайтовые величины и их реверсивные аналоги (всего 256 значений);

- исходное число, которое необходимо реверсировать, разделим на байты;
- каждый байт заменим на соответствующий аналог из таблицы;
- если это необходимо ($size$ не является величиной кратной 8), выполним сдвиг вправо полученного результата.

Таблица содержит однобайтовые величины (256 элементов), так как минимальный объём памяти, с которым может работать процессор, это один байт, а содержать в таблице величины, некратные одному байту, невыгодно с точки зрения эффективности их обработки и хранения. Можно завести таблицу, содержащую двухбайтовые величины, на 65536 записей, но эффективность такой реализации не будет значительно отличаться от рассматриваемой, а дополнительной памяти будет расходоваться значительно больше.

Рассмотрим пример, в котором необходимо реверсировать число, содержащее 19 значимых бит, т.е. размер обрабатываемого массива равен 2^{19} (Рис. 60). Цветами на рисунке помечены разные биты трехбайтового представления числа, старшие 5 бит помечены белым цветом, т.к. их содержимое равное нулю. Реверсивное число так же должно быть записано в 3 байта.

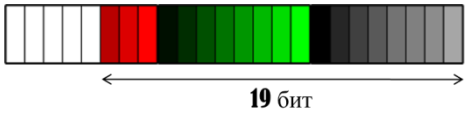


Рис. 60. Исходное число

В соответствии с алгоритмом записываем младший байт в инверсивном порядке на старшую позицию в реверсивном числе (Рис. 61).

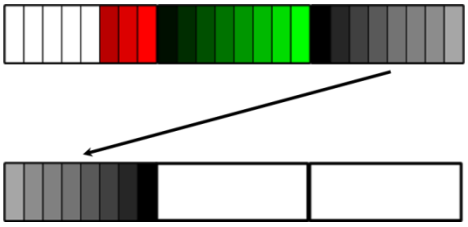


Рис. 61. Реверсирование младшего байта

Реверсивный аналог второго байта запишется на среднюю позицию в реверсивном числе (Рис. 62).

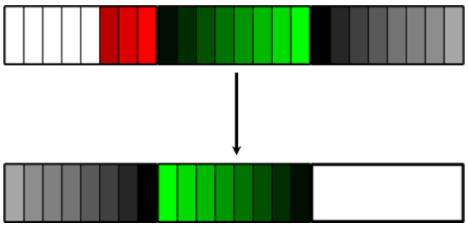


Рис. 62. Реверсирование среднего байта

Реверсивный аналог старшего байта запишется на младшую позицию в реверсивном числе (Рис. 63).

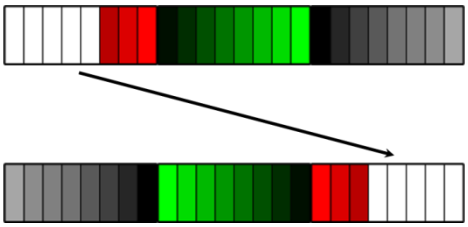


Рис. 63. Реверсирование старшего байта

Полученное к текущему моменту число содержит в младших битах нули, т.к. размер обрабатываемого числа был не кратен 8. Выполняем сдвиг на 5 бит вправо для того, чтобы получить реверсивный аналог исходного числа (Рис. 64).

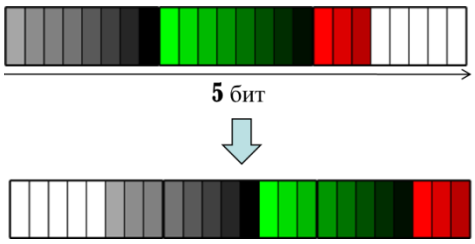


Рис. 64. Сдвиг числа

Эффективность рассматриваемого алгоритма во многом зависит от того как будет реализована таблица и метод выделения байта в числе.

Наиболее эффективным способом задания таблицы будет создание статической таблицы с заранее посчитанными значениями (сами реверсивные значения можно определить, написав соответствующую программу). В этом случае не придётся высчитывать реверсивные числа программно при создании таблицы, а самое главное то, что компилятор сможет оптимизировать хранение и использование такой таблицы.

Для выделения байта из числа можно воспользоваться объединениями (**`union`**). В этом случае число типа **`int`** можно будет интерпретировать

как массив типа **char** и тем самым получить доступ к байтам этого числа без дополнительных вычислений.

Заметим следующее свойство бит-реверсирования: пусть число y является реверсивным для числа x, тогда число $y+1$ будет реверсивным для числа $x+\frac{size}{2}$, где $size$ – размер обрабатываемого массива (при допущении, что $size$ является степенью двойки и $x<\frac{size}{2}$). Используем этот факт при реализации модифицированной версии функции, выполняющей перестановку элементов массива.

Заведите статическую таблицу с реверсивными значениями байтов на 256 записей и внесите изменения в функцию **BitReversing** *в файле* **fft.cpp***:*

```
unsigned char rev[256]={
0, 128, 64, 192, 32, 160, 96, 224, 16, 144, 80, 208, 48,
176, 112, 240, 8, 136, 72, 200, 40, 168, 104, 232, 24, 152,
88, 216, 56, 184, 120, 248, 4, 132, 68, 196, 36, 164, 100,
228, 20, 148, 84, 212, 52, 180, 116, 244, 12, 140, 76, 204,
44, 172, 108, 236, 28, 156, 92, 220, 60, 188, 124, 252, 2,
130, 66, 194, 34, 162, 98, 226, 18, 146, 82, 210, 50, 178,
114, 242, 10, 138, 74, 202, 42, 170, 106, 234, 26, 154, 90,
218, 58, 186, 122, 250, 6, 134, 70, 198, 38, 166, 102, 230,
22, 150, 86, 214, 54, 182, 118, 246, 14, 142, 78, 206, 46,
174, 110, 238, 30, 158, 94, 222, 62, 190, 126, 254, 1, 129,
65, 193, 33, 161, 97, 225, 17, 145, 81, 209, 49, 177, 113,
241, 9, 137, 73, 201, 41, 169, 105, 233, 25, 153, 89, 217,
57, 185, 121, 249, 5, 133, 69, 197, 37, 165, 101, 229, 21,
149, 85, 213, 53, 181, 117, 245, 13, 141, 77, 205, 45, 173,
109, 237, 29, 157, 93, 221, 61, 189, 125, 253, 3, 131, 67,
195, 35, 163, 99, 227, 19, 147, 83, 211, 51, 179, 115, 243,
11, 139, 75, 203, 43, 171, 107, 235, 27, 155, 91, 219, 59,
187, 123, 251, 7, 135, 71, 199, 39, 167, 103, 231, 23, 151,
87, 215, 55, 183, 119, 247, 15, 143, 79, 207, 47, 175, 111,
239, 31, 159, 95, 223, 63, 191, 127, 255};

union IntUni
{
  unsigned int integer;
  unsigned char byte[4];
};

void BitReversing(double *inputSignal,
  double *outputSignal, int size, int step, int offset)
{
  int m=0;
  for(int tmp_size = size; tmp_size>1; tmp_size/=2,
      m++ );//size = 2^m
```

```
  IntUni i, j;
  j.integer=0;

  double *_inputSignal = inputSignal + 2*offset;
  double *_outputSignal = outputSignal + 2*offset;

  for(i.integer=0; i.integer<size/2; i.integer++)
  {
    j.byte[0]=rev[i.byte[3]];
    j.byte[1]=rev[i.byte[2]];
    j.byte[2]=rev[i.byte[1]];
    j.byte[3]=rev[i.byte[0]];

    j.integer = j.integer >> (32-m);

    _outputSignal[2*i.integer*step] =
                        _inputSignal[2*j.integer*step];
    _outputSignal[2*i.integer*step+1] =
                        _inputSignal[2*j.integer*step+1];
    _outputSignal[(2*i.integer+size)*step] =
                        _inputSignal[(2*j.integer+2)*step];
    _outputSignal[(2*i.integer+size)*step+1] =
                       _inputSignal[(2*j.integer+2)*step+1];
  }
}
```

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\09. filter (precalculate)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\09. filter (precalculate)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 26с. (Рис. 65). Выигрыш по сравнению с предыдущей версией составил 11.6%.

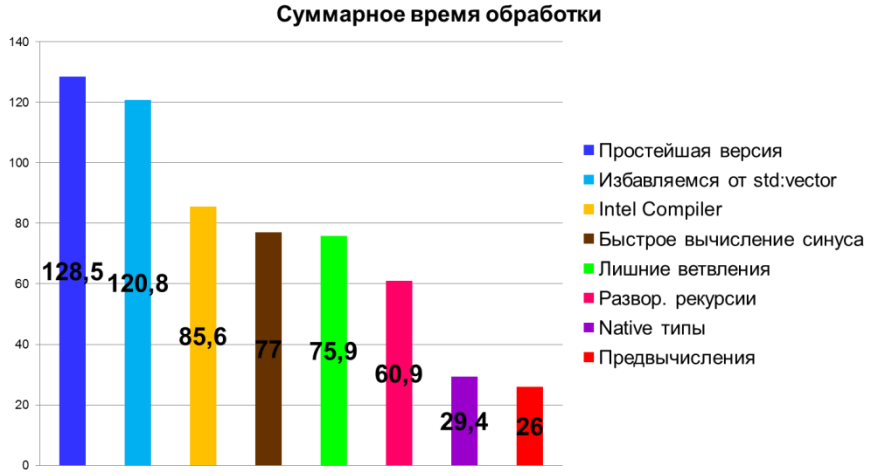


Рис. 65. Прогресс оптимизации

14. Оптимизация. Упрощение выражений

Для принятия решения о направлении дальнейшей оптимизации выполним поиск горячих точек в программе, используя инструмент Intel VTune Amplifier XE (Рис. 28):

- используйте **Release**-версию программы;
- выберите режим работы **Hotspots** (Рис. 29).

В результате предшествующей оптимизации функции, вычисляющей БПФ, функция `ProcessFrame()` стала требовать существенного времени на фоне общего времени работы приложения (Рис. 66).

/Function /Call Stack	CPU Time	Module
SerialInverseFFT1D	30.1%	filter.exe
SerialFFT1D	29.7%	filter.exe
ProcessFrame	25.1%	filter.exe
cvSetTrackbarPos	8.9%	highgui110.dll
main	2.5%	filter.exe
cvWaitKey	1.4%	highgui110.dll
[AcXtrnal.DLL]	0.6%	AcXtrnal.DLL
icvUpdateTrackbar	0.5%	highgui110.dll
icvFlipVert_8u_C1R	0.2%	cxcore110.dll
[Import thunk __libm_sse2_log]	0.2%	filter.exe
icvGetBitmapData	0.2%	highgui110.dll
Selected 1 row(s):	25.1%	

Рис. 66. Результат профилирования

В основном **`ProcessFrame()`** выполняет вычисление математических функций (Рис. 67): извлечение корня, возведение в степень, логарифм, экспонента.

Line	Source	CPU Time
183	for (int i=0; i<size; i++)	0.1%
184	{	
185	if (sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1]) != 0)	0.7%
186	if(maxB < log(sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1])))	6.4%
187	maxB = log(sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1]));	0.2%
188		
189	if (sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1]) != 0)	
190	if(minB > log(sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1])))	4.8%
191	minB = log(sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1]));	0.2%
192	}	
	Selected 1 row(s):	6.4%

Рис. 67. Профиль функции **`ProcessFrame`**

Для начала избавимся от функций **`pow`**, которая возводит в степень число «**`-1`**», т.к. вместо возведения в степень будет достаточно умножения на «**`-1`**» (Рис. 68).

```
73   for (int i=0; i<size; i++)
74   {
75     inpB[2*i] = pow(-1.0,i/w+i%w) * ((unsigned char*)frame->imageData)[i*3 + 0];
76     inpB[2*i+1] = 0.0;
77   }
```

```
for (int i=0; i<h; i++)
{
  a = pow(-1.0, i);
  int b = a;
  for (int j=0; j<w; j++)
  {
    inpB[2*(i*w + j)]=b * ((unsigned char*)frame->imageData)[(i*w + j)*3 + 0];
    inpB[2*(i*w + j) + 1]=0.0;
    b *= -1;
  }
  a *= -1;
}
```

Рис. 68. Вычисление «**`-1`**» в степени

Во всех остальных случаях заменим вычисление функции **pow** на соответствующее умножение (Рис. 69).

```
96        case GaussLow:
97          f[i+j*w]=exp( -(pow(double(i-w/2), 2) + pow(double(j-h/2), 2) ) / (2 * pow((double)r, 2) ) );//Ga
98          break;
99        case GaussHigh:
100         f[i+j*w]=1 - exp( -(pow(double(i-w/2), 2) + pow(double(j-h/2), 2) ) / (2 * pow((double)r, 2) ) );
101         break;
```

```
case GaussLow:
  f[i+j*w]=exp( -( (i-w/2)*(i-w/2) + (j-h/2)*(j-h/2) ) / (2.0*r*r ) );//Gau
  break;
case GaussHigh:
  f[i+j*w]=1 - exp( -( (i-w/2)*(i-w/2) + (j-h/2)*(j-h/2) ) / (2.0*r*r ) );/
  break;
```

Рис. 69. Возведение в степень

Функция **ProcessFrame()** содержит большое количество избыточных вычислений, которые можно выполнить один раз. Избавимся от повторных вычислений выражений (Рис. 70).

```
147   for (int i=0; i<size; i++)
148   {
149     if (sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]) != 0)
150       if(maxR < log( sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]) ) )
151         maxR = log( sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]) );
152
153     if (sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]) != 0)
154       if(minR > log( sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]) ) )
155         minR = log( sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]) );
156   }
```

```
for (int i=0; i<size; i++)
{
  double t = sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]);
  double p = log(t);

  if (t != 0)
  {
    if(maxR < p )
      maxR = p;

    if(minR > p )
      minR = p;
  }
}
```

Рис. 70. Повторные вычисления

15. Распараллеливание. Разработка простейшей OpenMP-версии программы

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\10. filter (fast math)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\10. filter (fast math)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 18.6с. (Рис. 71). Выигрыш по сравнению с предыдущей версией составил 28.2%.

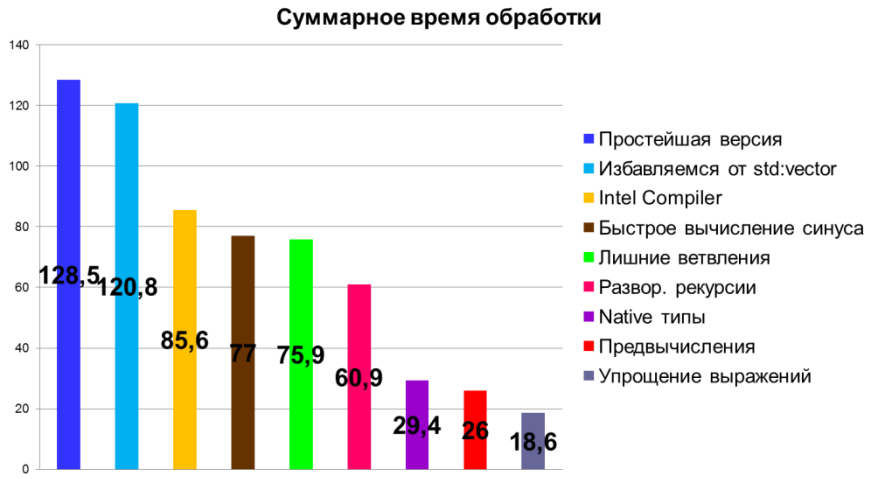


Рис. 71. Прогресс оптимизации

Многие современные вычислительные системы имеют несколько вычислительных процессоров/ядер. Наша тестовая система (Таблица 1) содержит два четырехъядерных процессора (8 вычислительных ядер) (Рис. 72). Выполним распараллеливание функции, выполняющей расчет БПФ, с помощью OpenMP [7].

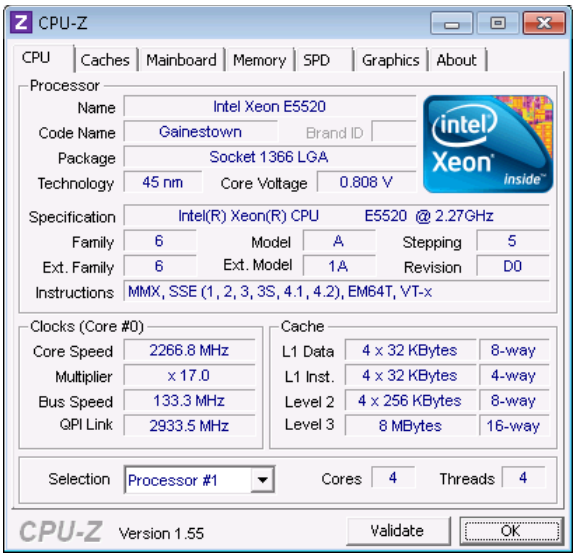


Рис. 72. Информация о процессоре, полученная утилитой CPU-Z

Добавьте строку, выполняющую распараллеливание, **`#pragma omp parallel for`** *перед «средним» циклом функций* **`SerialFFTCalculation`** *и* **`SerialInverseFFTCalculation`** *в файле* **fft.cpp***:*

```
void SerialFFTCalculation(double *mas, int size,
```

```
  int step, int off)
{
  …

  for (int p=0; p<m; p++ )
  {
    int butterflyOffset = 1 << (p+1) ;
    int butterflySize = butterflyOffset >> 1 ;

    double alpha=-2.0*PI/butterflyOffset;
    double wR=cos( alpha  ), wI=sin( alpha );

    double uRtmp;
    double temI;

    #pragma omp parallel for
    for (int i=0; i<size/butterflyOffset; i++ )
    {
      …
    }
  }
}

void SerialInverseFFTCalculation(double *mas, int size,
  int step, int off)
{
  …

  for (int p=0; p<m; p++ )
  {
    int butterflyOffset = 1 << (p+1) ;
    int butterflySize = butterflyOffset >> 1 ;

    double alpha=2.0*PI/butterflyOffset;
    double wR=cos( alpha  ), wI=sin( alpha );

    #pragma omp parallel for
    for (int i=0; i<size/butterflyOffset; i++ )
    {
      …
    }
}
```

Перед сборкой проекта необходимо включить поддержку OpenMP в настройках проекта. Для этого:

- откройте свойства проекта, нажав **Alt+F7** или выбрав пункт **Properties** в меню **Project**;

- перейдите к пункту **Language** настроек компилятора **C/C++**;
- в опции **OpenMP Support** из выпадающего списка установите значение **Generate Parallel Code (/openmp, equiv. to /Qopenmp)** (Рис. 73);

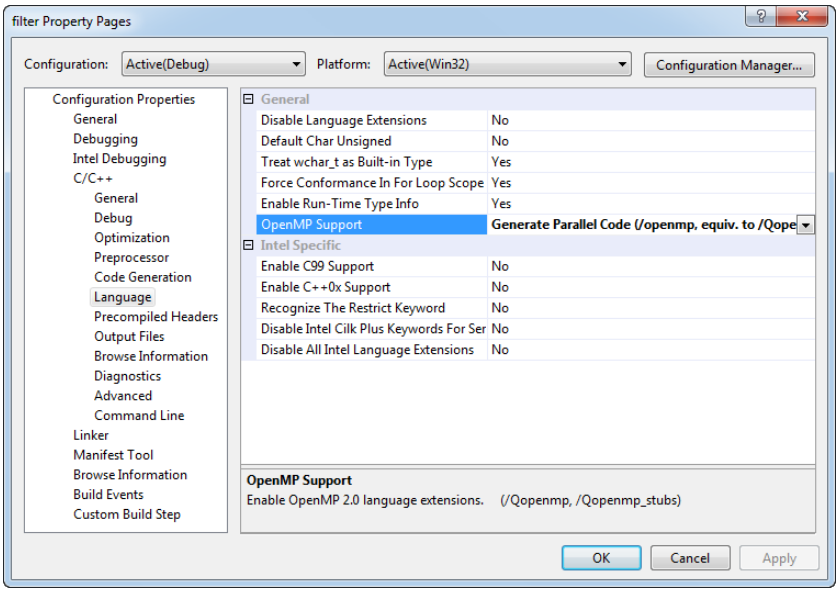


Рис. 73. Включение OpenMP-расширения

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\11. filter (parallel)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\11. filter (parallel)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**. Запустите приложение, нажав **Ctrl+F5** или выполнив команду **Start Without Debugging** пункта меню **Debug**. Отфильтрованное изображение содержит шумы (Рис. 74), которые вызваны одной из типичных ошибок параллельной программы.

Рис. 74. Некорректное изображение

С помощью Intel Inspector XE 2011 найдите одну из типичных ошибок в параллельной программе:

- используйте **Debug**-версию программы;
- выберите режим работы **Threading Error Analysis / Locate Deadlocks and Date Races** (Рис. 75);
- нажмите **ESC** после начала обработки первого кадра (достаточно одного обработанного кадра).

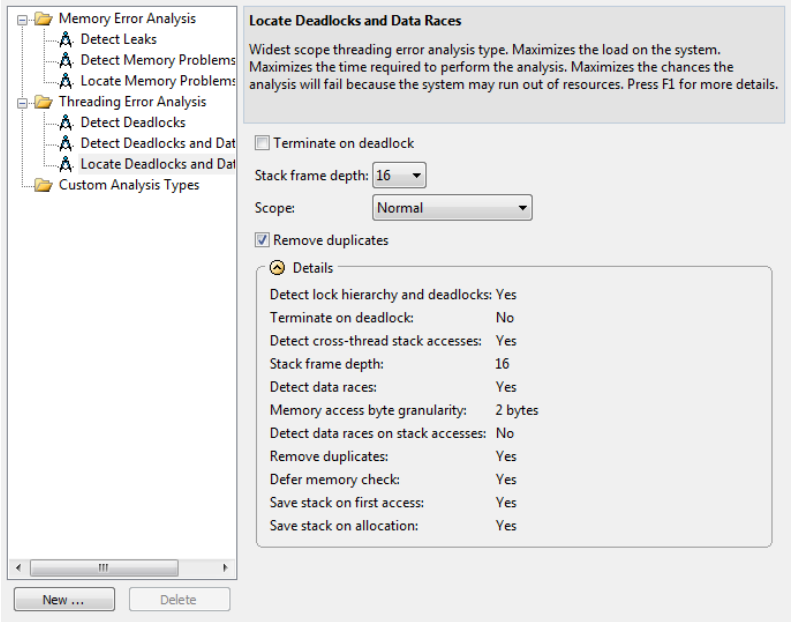


Рис. 75. Intel Inspector XE 2011. Ошибки параллельных программ

После завершения программы и обработки собранных данных Intel Inspector XE выдаст список найденных ошибок. Среди них будет две ошибки типа «гонки данных» (**`Date Race`**). Эти ошибки вызваны тем, что потоки используют переменные **`uRtmp`** (Рис. 77) и **`temI`** (Рис. 76) параллельно для записи и чтения. В результате может сложиться такая ситуация, что один поток использует данные, которые посчитал другой поток. Таким образом, БПФ выполняется с ошибками, а отфильтрованное изображение содержит шумы.

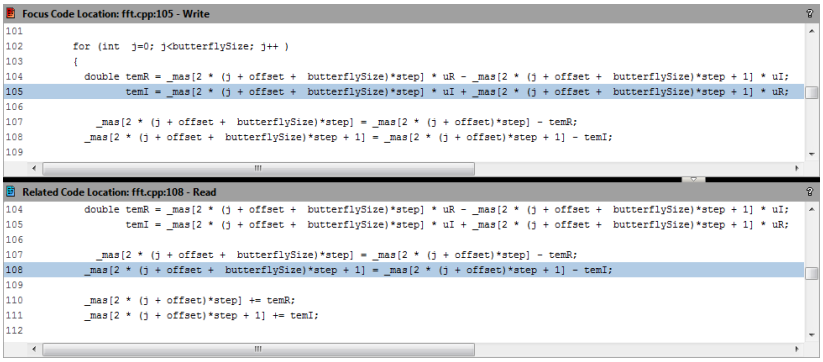


Рис. 76. «Гонки данных», вызванные переменной **`temI`**

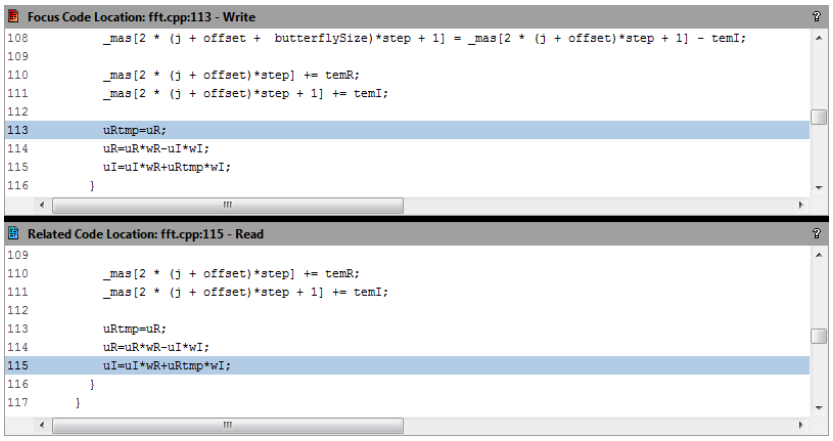


Рис. 77. «Гонки данных», вызванные переменной **`uRtmp`**

Чтобы исправить найденные ошибки, необходимо воспользоваться одним из следующих способов:

- использовать ключевое слово **`private`**:

```
#pragma omp parallel for private(temI, uRtmp)
```

- объявить переменные **`uRtmp`** и **`temI`** локально.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build** (не забудьте включить поддержку OpenMP в настрой-

ках проекта). Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы.

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 32с., что почти в 2 раза медленнее, чем последовательная версия программы. Для определения причин замедления программы воспользуйтесь Intel VTune Amplifier XE 2011:

- используйте **Release**-версию программы;
- выберите режим работы **Algorithm Analysis / Concurrency**;
- выполните команду **New… / Copy from current** и настройте конфигурацию в соответствии с Рис. 78;

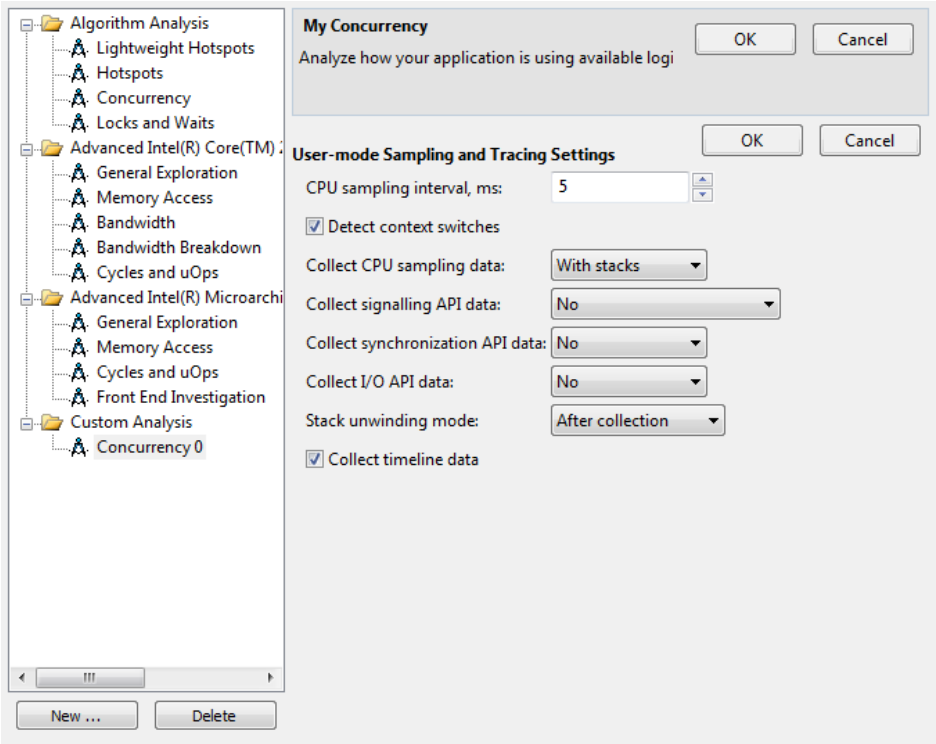


Рис. 78. Intel VTune Amplifier XE. Конфигурация анализа параллельных программ

- нажмите «OK» и завершите программу после обработки нескольких кадров при помощи клавиши **ESC**.

После завершения программы и обработки собранных данных Intel VTune Amplifier XE отобразит профиль приложения (Рис. 79).

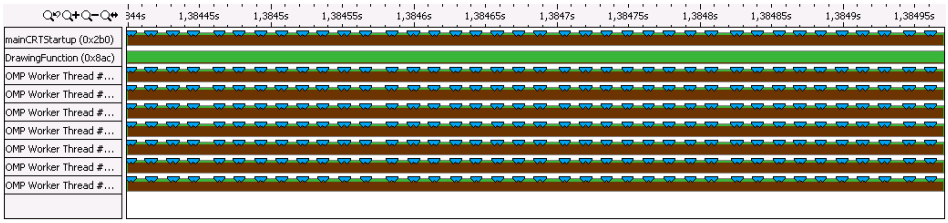


Рис. 79. Часть собранного профиля

На профиле отображены все потоки программы: восемь OpenMP-потоков и один поток, выполняющий визуализацию (функция **`DrawingFunction`**). Всё время на указанном фрагменте профиля работали одновременно 8 потоков OpenMP, то есть достигалась максимальная загрузка всех вычислительных ядер системы. Синими маркерами на профиле отображены параллельные секции OpenMP. Можно заметить, что их достаточно много. Обратим внимание на увеличенный фрагмент профиля (Рис. 80).

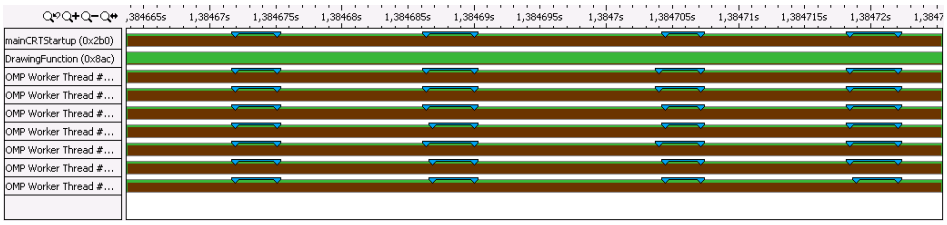


Рис. 80. Увеличенный фрагмент профиля

Каждая параллельная секция выполняется менее, чем за 20мкс.! Замедление программы вызвано тем, что накладные расходы на создание параллельных секций значительно превосходят объём вычислений в этих секциях. Функция **SerialFFTCalculation()** обрабатывает малые массивы данных (на рассматриваемом тестовом видео всего 256 элементов), поэтому её распараллеливание неактуально. Отмените модификации, сделанные в данном разделе.

Выполним распараллеливание двух циклов функции **SerialFFT2D()**. Каждая итерация в этих циклах может выполняться независимо, т.к. на каждой итерации обрабатывается отдельная строка (столбец) матрицы, задающей изображение. Параллельную реализацию поместим в функцию **ParallelFFT2D()**. Аналогично поступим с функцией **SerialInverseFFT2D()**.

```
void ParallelFFT2D(double *inputSignal,
  double *outputSignal, int w, int h)
{
  double *tem = new double[2*w*h];

  #pragma omp parallel for
  for(int i=0; i<w; i++)
    SerialFFT1D(inputSignal, tem, h, w, i);

  #pragma omp parallel for
  for(int j=0; j<h; j++)
    SerialFFT1D(tem, outputSignal, w, 1, h*j);

  delete[] tem;
}

void ParallelInverseFFT2D(double *inputSignal,
  double *outputSignal, int w, int h)
{
  double *tem=new double[2*w*h];

  #pragma omp parallel for
  for(int i=0; i<w; i++)
```

```
    SerialInverseFFT1D(inputSignal, tem, h, w, i);

  #pragma omp parallel for
  for(int j=0; j<h; j++)
    SerialInverseFFT1D(tem, outputSignal, w, 1, h*j);

  delete[] tem;
}
```

15.1. Гонки данных

Рассмотрим ситуацию, называемую «гонкой данных». Пусть есть общая переменная **data**, доступная нескольким потокам для чтения и записи. Каждый поток должен выполнить инкремент этой переменной (то есть выполнить код **data++**). Для этого процессору необходимо выполнить три операции: чтение значения переменной из оперативной памяти в регистр процессора, инкремент регистра, запись посчитанного значения в переменную (в оперативную память).

Возможны две реализации (с точностью до перестановки потоков) одновременного выполнения такого кода двумя потоками. Наиболее ожидаемое поведение приложения представлено на Рис. 81. Сначала поток 0 выполняет чтение переменной, инкремент регистра и запись его значения в переменную, а потом поток 1 выполнит ту же последовательность действий. Таким образом, после завершения работы приложения значение общей переменной будет равно **data+2**.

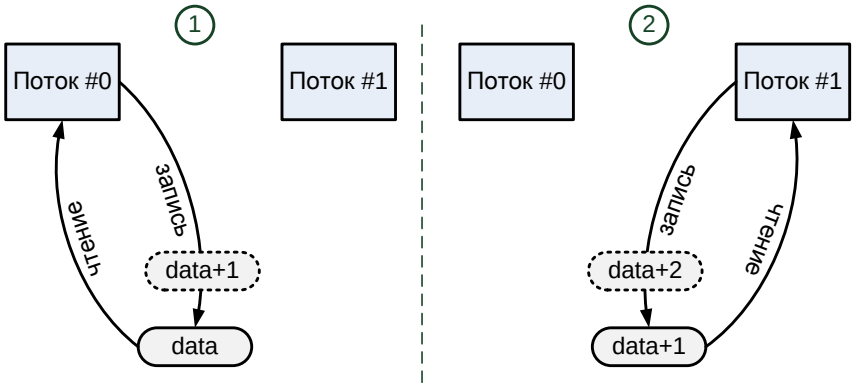


Рис. 81. Вариант реализации «гонки данных»

Другое возможное поведение представлено на Рис. 82. Поток 0 выполняет чтение значения переменной в регистр и инкремент этого регистра, и в этот же момент времени поток 1 выполняет чтение переменной **data**. Так как для каждого потока имеется свой набор регистров, то поток 0 продолжит выполнение и запишет в переменную значение **data+1**. Поток 1 также выполнит инкремент регистра (значение переменной **data** было прочи-

тано ранее из оперативной памяти, до записи потоком 0 значения **data+1**) и сохранит значение **data+1** (свое) в общую переменную. Таким образом, после завершения работы приложения, значение переменной будет равно **data+1**. Обе реализации могут наблюдаться как на многоядерной (многопроцессорной) системе, так и на однопроцессорной, одноядерной.

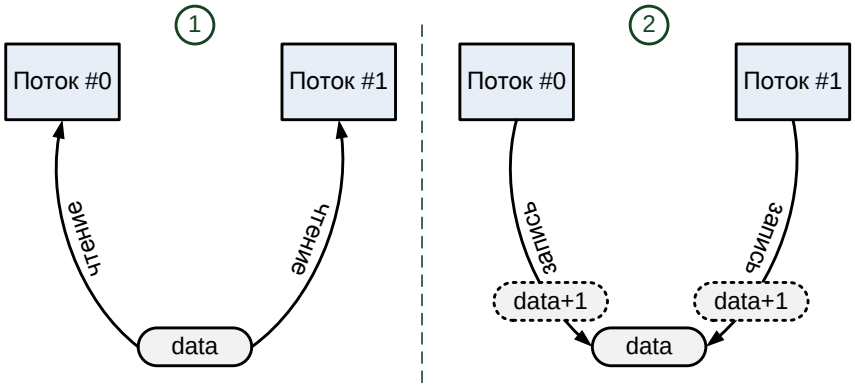


Рис. 82. Вариант реализации “гонки данных”

Таким образом, в зависимости от порядка выполнения команд результат работы приложения может меняться.

Решить рассмотренную проблему можно за счёт использования механизма, обеспечивающего синхронизацию выполнения потоков (с помощью такого механизма можно обеспечить выполнение части кода не более чем одним потоком в каждый момент времени), или за счёт создания локальных копий переменных для каждого потока.

16. Распараллеливание. Эффективная OpenMP-реализация программы

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\12. filter (effective parallel)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\12. filter (effective parallel)**;

- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: "**filter.exe ..\Videos\TestVideo1.avi -s -gl**". Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите "**filter.exe ..\Videos\TestVideo1.avi -s -gl**" и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 8.4с. (Рис. 83).

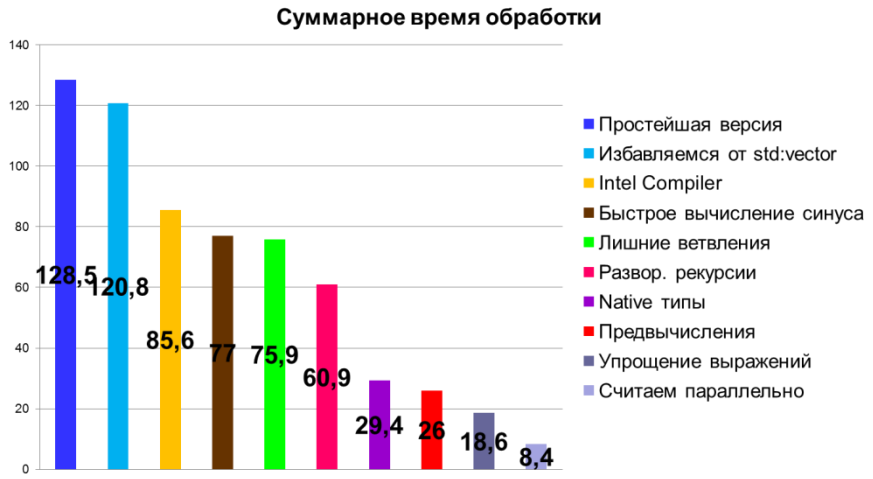


Рис. 83. Прогресс оптимизации

Выигрыш параллельной версии составил 54.9%. Общее ускорение по сравнению с последовательной версией всего 2.2, при том, что доступно 8 вычислительных ядер! Дело в том, что выполнялось распараллеливание только части программы, поэтому ускорение и не могло получиться максимальным.

Было выполнено распараллеливание той части кода, которая занимала 51% от общего времени (последовательная версия программы, Рис. 84). По закону Амдала [8] получается, что максимальное ускорение примерно равно двум (что и наблюдается).

Call Stack	CPU Time	CPU Time:Total
_tmainCRTStartup	0s	72.9%
main	2.076s	72.9%
ProcessFrame	2.784s	66.1%
SerialFFT1D	8.407s	26.0%
SerialInverseFFT1D	8.055s	24.9%
__svml_log2	1.361s	4.2%
[AcXtrnal.DLL]	0.644s	2.0%
[Import thunk __libm_sse2_log]	0.061s	0.2%
__svml_log2	0.015s	0.0%
[Import thunk __libm_sse2_exp]	0.010s	0.0%
cvSetTrackbarPos	0.096s	0.3%
Selected 2 row(s):	16.462s	51.0%

Рис. 84. Профиль последовательной версии программы

Intel VTune Amplifier XE 2011 также позволяет убедиться, что большую часть времени параллельные секции одновременно обрабатывают 8 потоков (нижняя часть графика).

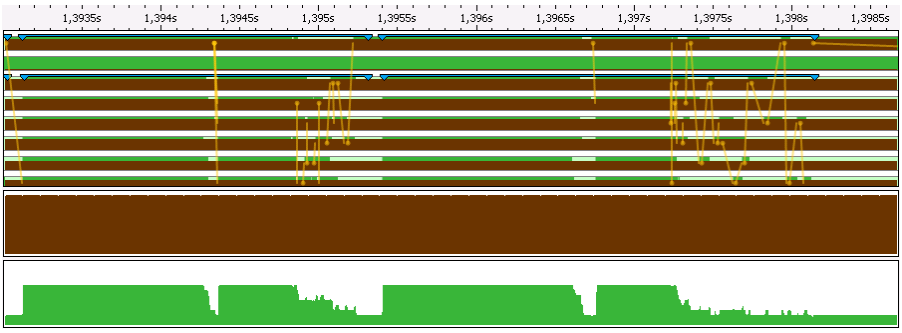


Рис. 85. Часть профиля параллельной версии программы

Распараллелим функцию **`ProcessFrame()`**. Также выполним следующие этапы оптимизации:

- вынесем **`switch`** из тела цикла;
- будем выделять/освобождать динамическую память один раз.

Выполните объявление переменных, хранящих фильтр и обработанные сигналы, в строке 54 файла **filter.cpp**:

```
double *f;
double *inpR, *inpG, *inpB,
       *outR, *outG, *outB;
```

Выделение памяти под эти переменные добавьте в начало функции **main**, *а освобождение памяти – в конец функции* **main**.

Выполните распараллеливание всех циклов функции **ProcessFrame** *в файле* **filter.cpp**:

```
void ProcessFrame(IplImage *frame, IplImage *outFrame,
  IplImage* filter, IplImage* freq, FilterType t, int *_r)
{
  int w = frame->width,
      h = frame->height,
      r = *_r,
      size = w * h;

  int a;

  double maxR = 0.0 , maxG = 0.0, maxB = 0.0;
  double minR = 0.0 , minG = 0.0, minB = 0.0;

  //Fill inp<R|G|B> arrays
  #pragma omp parallel for private(a)
  for (int i=0; i<h; i++)
  {
    a = pow(-1.0, i);
    int b = a;
    for (int j=0; j<w; j++)
    {
      inpB[2*(i*w + j)]=b *
         ((unsigned char*)frame->imageData)[(i*w + j)*3];
      inpB[2*(i*w + j) + 1]=0.0;
      b *= -1;
    }
  }

  …

  //Make filter
  switch(t)
  {
    case GaussLow:
      #pragma omp for
      for (int i=0; i<w; i++)
        for (int j=0; j<h; j++)
          f[i+j*w]=
             exp( -( (i-w/2)*(i-w/2) + (j-h/2)*(j-h/2) ) /
             (2.0*r*r ) );
      break;
    …
  }

  //Show filter
```

```
#pragma omp parallel
{
  #pragma omp for nowait
  for(int i=0; i<size; i++)
    filter->imageData[i] = (char) (255 * f[i]);

  //Apply filter
  #pragma omp for nowait
  for (int i=0; i<size; i++)
  {
    outR[2*i] = outR[2*i] * f[i];
    outR[2*i+1] = outR[2*i+1] * f[i];
  }
  …
}

//Find max value
for (int i=0; i<size; i++)
{
  double t =
     sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]);

  if (t != 0)
  {
    maxR = minR = log(t);
    break;
  }
}

#pragma omp parallel
{
  double max = maxR;
  double min = minR;

  #pragma omp for nowait
  for (int i=0; i<size; i++)
  {
    double t =
       sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]);
    double p = log(t);

    if (t != 0)
    {
      if(max < p )
        max = p;

      if(min > p )
        min = p;
    }
```

```
    }

    #pragma omp critical
    {
      if(max > maxR)
        maxR = max;

      if(min < minR)
        minR = min;
    }
    …

  //Show frequency area with filter
  #pragma omp parallel
  {
    #pragma omp for nowait
    for (int i=0; i<size; i++)
    {
      double t =
         sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]);
      double delR = maxR - minR;

      if (t != 0)
        freq->imageData[3*i+0] =
           (char) (255 * ( log(t) - minR ) / delR );
      else
        freq->imageData[3*i+0] = 0;
    }
    …
  }

  #pragma omp parallel for private(a)
  for (int i=0; i<h; i++)
  {
    a = pow(-1.0, i);
    int b = a;
    for (int j=0; j<w; j++)
    {
      outFrame->imageData[3*(i*w + j)+0] =
         (char) (b * inpB[2*(i*w + j)]);
      outFrame->imageData[3*(i*w + j)+1] =
         (char) (b * inpG[2*(i*w + j)]);
      outFrame->imageData[3*(i*w + j)+2] =
         (char) (b * inpR[2*(i*w + j)]);

      b *= -1;
    }
  }
}
```

17. Оптимизация. Эффективное использование кеш-памяти

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\13. filter (full parallel)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\13. filter (full parallel)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 4.4 с (Рис. 86). Выигрыш по сравнению с предыдущей версией составил 46.6 %. Время обработки одного кадра составило 0.015 с.

До настоящего момента все эксперименты проводились на одном видеофрагменте, поэтому вся оптимизация проводилась, по сути, для этого файла (следовательно, и размера картинки). Проведём эксперимент на видеофрагментах с разрешением 512x512 и 1024x1024 точек.

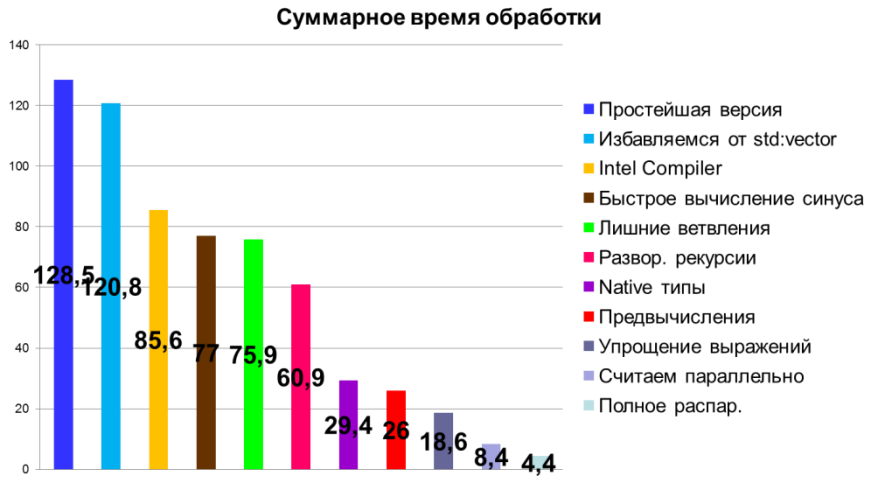


Рис. 86. Прогресс оптимизации

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo2.avi -s -gl**”.

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 24.9 с. Время обработки одного кадра составило 0.065 с.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo3.avi -s -gl**”.

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 293.5 с. Время обработки одного кадра составило 0.481 с.

Основным вычислительным алгоритмом в рассматриваемой задаче является вычисление двумерного БПФ. Таким образом, трудоёмкость обработки одного кадра составляет: $O(\mathrm{h}\cdot\mathrm{w}\cdot\log\mathrm{w}+w\cdot h\cdot\log\mathrm{h})$, где w – количество точек по горизонтали в изображении, а h – по вертикали. Рассмотрим изображение по горизонтали и вертикали увеличенное в два раза. Трудоёмкость обработки такого изображения составит: $O(4\cdot\mathrm{h}\cdot\mathrm{w}\cdot\log(2\cdot\mathrm{w})+4\cdot w\cdot h\cdot\log(2\cdot\mathrm{h}))$.

Таким образом, время обработки кадра увеличенного в два раза по горизонтали и вертикали должно увеличиться в $\frac{4\cdot\mathrm{h}\cdot\mathrm{w}\cdot\log(2\cdot\mathrm{w})+4\cdot w\cdot h\cdot\log(2\cdot\mathrm{h})}{\mathrm{h}\cdot\mathrm{w}\cdot\log\mathrm{w}+w\cdot h\cdot\log\mathrm{h}}$ раз.

$$\frac{4\cdot\mathrm{h}\cdot\mathrm{w}\cdot\log(2\cdot\mathrm{w})+4\cdot w\cdot h\cdot\log(2\cdot\mathrm{h})}{\mathrm{h}\cdot\mathrm{w}\cdot log\mathrm{w}+w\cdot h\cdot log\mathrm{h}}=$$
$$=4\cdot\frac{log2+logw+log2+logh}{log\mathrm{w}+log\mathrm{h}}=4\cdot\left(1+2\cdot\left(\frac{log\mathrm{w}+log\mathrm{h}}{log2}\right)^{-1}\right)= \quad (26)$$

$$= 4 \cdot \left(1 + \frac{2}{\log_2 w + \log_2 h}\right)$$

Для $w = h = 256$ получаем замедление в $4 \cdot \left(1 + \frac{2}{8+8}\right) = 4.5$ раза. Для $w = h = 512$ получаем замедление в $4 \cdot \left(1 + \frac{2}{9+9}\right) \approx 4.4$ раза.

При переходе от изображения размером 256x256 к изображению размером 512x512 наблюдается замедление в $\frac{65}{15} \approx 4.3$ раза. При переходе от изображения размером 512x512 к изображению размером 1024x1024 наблюдается замедление в $\frac{481}{65} = 7.4$ раза. Результаты последнего эксперимента показывают, что замедление программы значительно больше, чем это предсказывалось теоретически.

Выполним профилировку приложения над **TestVideo1.avi**, используя инструмент Intel VTune Amplifier XE (Рис. 28):

- используйте **Release**-версию программы;
- выберите режим работы **Hotspots** (Рис. 29).

Вычисление одномерного БПФ по строкам работает в 10 раз быстрее, чем вычисление одномерного БПФ по столбцам (Рис. 87). Для **TestVideo2.avi** и **TestVideo3.avi** соотношения будут похожими.

Line	Source	CPU Time
168	void ParallelInverseFFT2D(double *inputSignal, double *outputSi	
169	{	
170	double *tem=new double[2*w*h];	
171		
172	#pragma omp parallel for	0.1%
173	for(int i=0; i<w; i++)	
174	SerialInverseFFT1D(inputSignal, tem, h, w, i);	24.5%
175		
176	#pragma omp parallel for	0.1%
177	for(int j=0; j<h; j++)	
178	SerialInverseFFT1D(tem, outputSignal, w, 1, h*j);	2.4%
179		
180	delete[] tem;	
181	}	
	Selected 1 row(s):	24.5%

Рис. 87. Профиль функции **`ParallelInverseFFT2D`**

Для дальнейшего анализа соберём более детальный профиль приложения над **TestVideo1.avi**, используя инструмент Intel VTune Amplifier XE (Рис. 28):

- используйте **Release**-версию программы;
- выберите режим работы **Advanced <Тип процессора> Analysis / Memory Access**.

Вычисление БПФ над строками приводит к тому, что появляется большое количество инструкций доступа к памяти, которые работают более 128 тактов процессора (Рис. 88), что является очень большой величиной для современного процессора.

Line	Source	CPU_CLK_... THREAD	INST_RETIRED. ANY	CPU_CLK_... REF	OFFCO... DATA_IN. LOCAL_...	MEM_INST_RETIRED. LATENCY_ABOVE_THRESHOLD_128
167						
168	void ParallelInverseFFT2D(double *inputSignal, double *					
169	{					
170	double *tem=new double[2*w*h];					
171						
172	#pragma omp parallel for	2,000,000	2,000,000			
173	for(int i=0; i<w; i++)					
174	SerialInverseFFT1D(inputSignal, tem, h, w, i);	24,602,000,000	13,396,000,000	36,498,000,000	13,500,000	21,984,000
175						
176	#pragma omp parallel for	138,000,000	86,000,000	162,000,000		
177	for(int j=0; j<h; j++)		2,000,000	2,000,000		
178	SerialInverseFFT1D(tem, outputSignal, w, i, h*j);	6,626,000,000	13,760,000,000	13,132,000,000	5,700,000	228,000
179						
180	delete[] tem;					
181	}					

Рис. 88. Детальный профиль функции **`ParallelInverseFFT2D`**

Современные процессоры содержат специальную память (кеш-память), позволяющую хранить часть обрабатываемых данных. Кеш-память характеризуется быстрым доступом к данным. Часто процессоры имеют многоуровневую иерархию кеш-памяти. Кэш 1-го уровня имеет небольшой размер, но очень высокую скорость работы, кэш 2-го уровня имеет большой размер, но меньшую скорость работы и т.д. Тестовая система имеет 3 уровня кеш-памяти (Рис. 89).

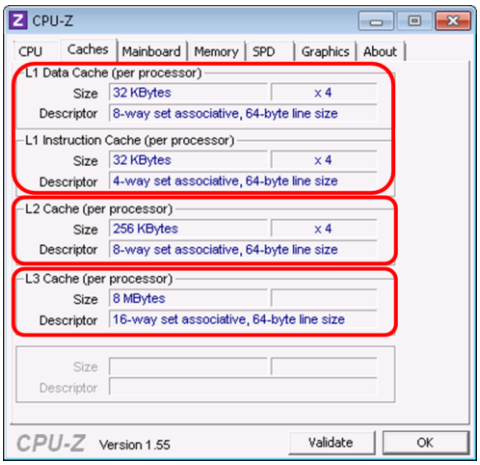


Рис. 89. Информация о кеш-памяти процессора, полученная утилитой CPU-Z

Для тестовой системы рассмотрим кеш-память 2-го уровня (Рис. 90):

- Размер кеш-памяти: 256 КБ.
- Ассоциативность: 8.
- Размер кеш-линейки: 64 Б.

Рис. 90. Модель кеш-памяти 2-го уровня

Рассмотрим компоненту цвета одного кадра видеофайла (Рис. 91).

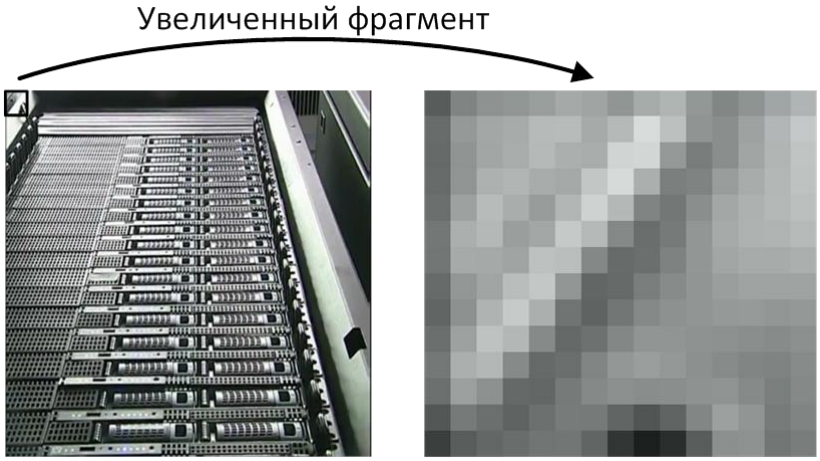


Рис. 91. Увеличенный фрагмент кадра

Рассмотрим, как будет происходить обработка первой строки при выполнении ДПФ. Первый элемент изображения занимает 16 байт (2 элемента типа double) и будет загружен в первую кеш-линейку (Рис. 92).

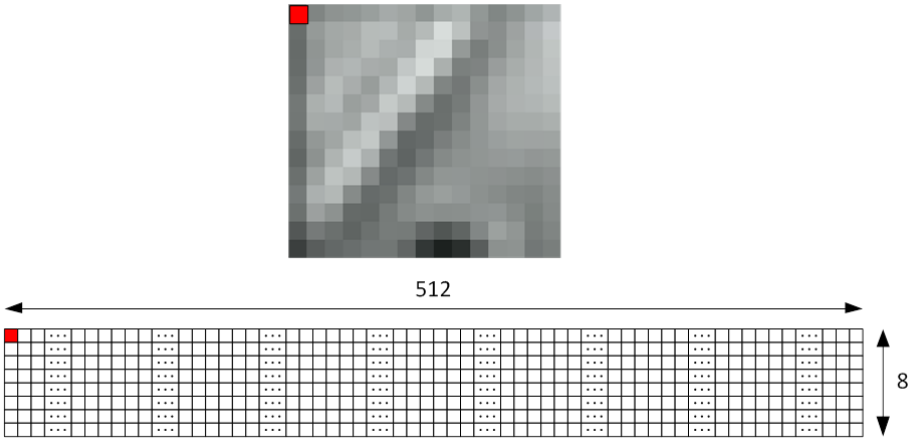


Рис. 92. Загрузка первого элемента изображения

Второй элемент изображения будет снова загружен в первую кеш-линейку, т.к. размер линейки кэша составляет 64 Б (Рис. 93).

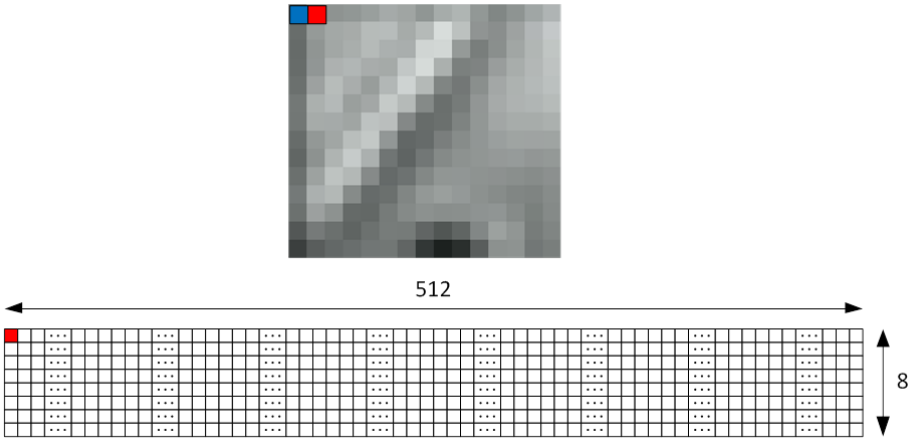


Рис. 93. Загрузка второго элемента строки изображения

Таким образом, первые 4 элемента изображения будет загружены в первую кеш-линейку. Пятый элемент изображения будет помещён во второй элемент кеш-памяти (Рис. 94).

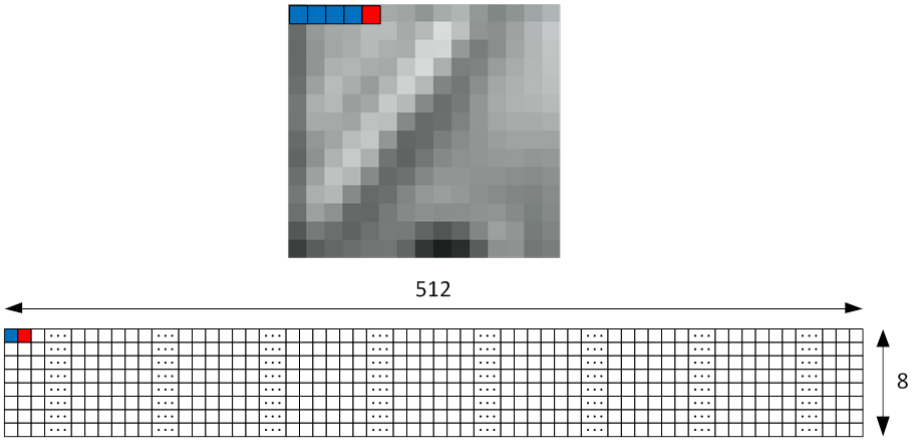


Рис. 94. Загрузка пятого элемента строки изображения

Строка тестового изображения занимает в памяти 4 КБ и может быть полностью загружена как в кеш-память 2-го уровня (Рис. 95), так и в более быстрою память 1-го уровня.

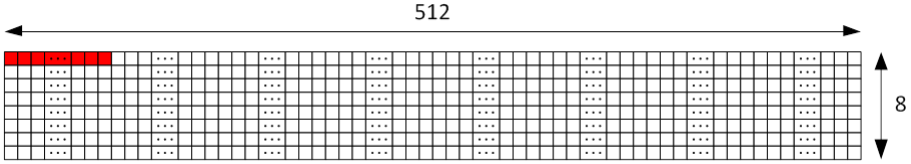


Рис. 95. Состояние кеш-памяти после загрузки всей строки изображения

Теперь рассмотрим, как будет происходить обработка первого столбца при выполнении ДПФ. Первый элемент изображения занимает 16 байт (2 элемента типа **`double`**) и будет загружен в первую кеш-линейку (Рис. 92).

Второй элемент столбца находится со смещением 4 КБ (256 * 16 Б), поэтому он будет загружен в кеш с таким же смещением (Рис. 96).

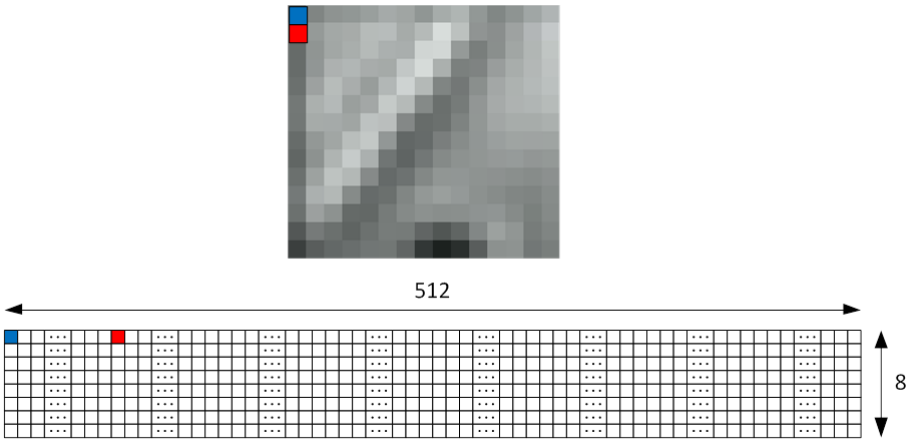


Рис. 96. Загрузка второго элемента столбца изображения

Каждый следующий элемент столбца смещён на 4 КБ. Поэтому после обработки первых 9 элементов столбца кеш-память будет использоваться неэффективно (Рис. 97).

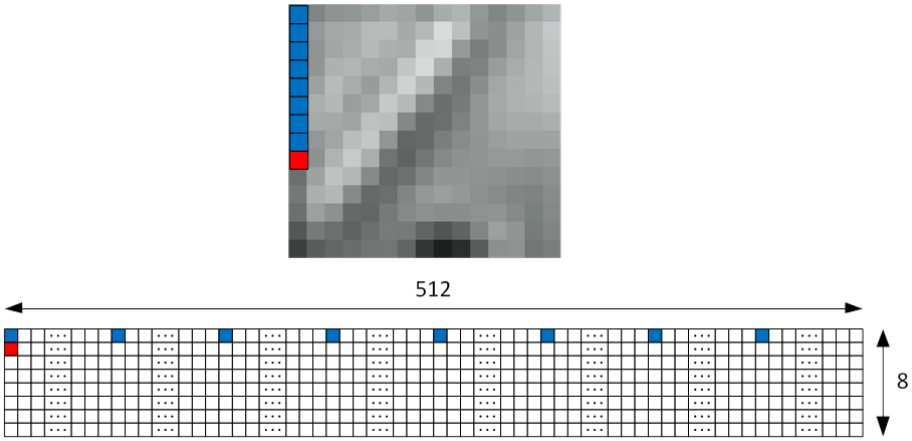


Рис. 97. Загрузка первых 9-ти элементов столбца изображения

Первые 64 элемента столбца заполнят $\frac{1}{64}$ часть линеек кеш-памяти. Очередной элемент столбца может быть помещён только в занятые позиции, поэтому один из находящихся в кеш-памяти элементов будет выгружен в кеш 3-го уровня (Рис. 98). Таким образом, повторное использование данных в кэше происходить не будет, а доступ к данным будет достаточно долгим.

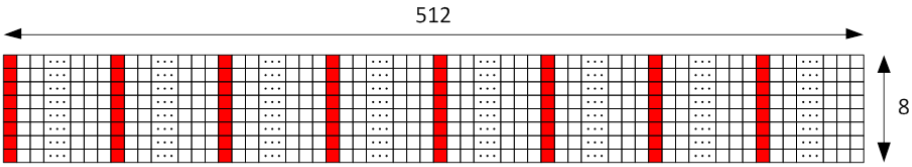


Рис. 98. Заполнение кеш-памяти после загрузки 64 элементов столбца изображения

Перед выполнением одномерного БПФ поместим обрабатываемый столбец в непрерывный массив (т.к. БПФ имеет трудоемкость $O(nlogn)$, то указанная оптимизация даст положительный эффект, поскольку трудоёмкость копирования линейна).

Во всех функциях в файле избавьтесь от параметров **`offset`** *и* **`step`**, *т.к. значения этих переменных будут всегда равны* **`0`** *и* **`1`** *соответственно*.

Выполните модификацию функций **`ParallelFFT2D`** *и* **`ParallelInverseFFT2D`** *в файле* **fft.cpp**:

```
void ParallelFFT2D(double *inputSignal,
  double *outputSignal, int w, int h)
{
  #pragma omp parallel
  {
```

```
    double *inW, *outW, *inH, *outH;

    inW = new double[2*w];
    outW = new double[2*w];
    inH = new double[2*h];
    outH = new double[2*h];

    #pragma omp for
    for(int i=0; i<w; i++)
    {
      for(int j=0; j<h; j++)
      {
        inH[2*j] = inputSignal[2*(j*w+i)];
        inH[2*j+1] = inputSignal[2*(j*w+i)+1];
      }

      SerialFFT1D(inH, outH, h);

      for(int j=0; j<h; j++)
      {
        outputSignal[2*(j*w+i)] = outH[2*j];
        outputSignal[2*(j*w+i)+1] = outH[2*j+1];
      }
    }

    #pragma omp for
    for(int j=0; j<h; j++)
    {
      for(int i=0; i<w; i++)
      {
        inW[2*i] = outputSignal[2*(j*w+i)];
        inW[2*i+1] = outputSignal[2*(j*w+i)+1];
      }

      SerialFFT1D(inW, outW, w);

      for(int i=0; i<w; i++)
      {
        outputSignal[2*(j*w+i)] = outW[2*i];
        outputSignal[2*(j*w+i)+1] = outW[2*i+1];
      }
    }

    delete[] inW;
    delete[] inH;
    delete[] outW;
    delete[] outH;
  }
}
```

```
void ParallelInverseFFT2D(double *inputSignal,
  double *outputSignal, int w, int h)
{
  #pragma omp parallel
  {
    double *inW, *outW, *inH, *outH;

    inW = new double[2*w];
    outW = new double[2*w];
    inH = new double[2*h];
    outH = new double[2*h];

    #pragma omp for
    for(int i=0; i<w; i++)
    {
      for(int j=0; j<h; j++)
      {
        inH[2*j] = inputSignal[2*(j*w+i)];
        inH[2*j+1] = inputSignal[2*(j*w+i)+1];
      }

      SerialInverseFFT1D(inH, outH, h);

      for(int j=0; j<h; j++)
      {
        outputSignal[2*(j*w+i)] = outH[2*j];
        outputSignal[2*(j*w+i)+1] = outH[2*j+1];
      }
    }

    #pragma omp for
    for(int j=0; j<h; j++)
    {
      for(int i=0; i<w; i++)
      {
        inW[2*i] = outputSignal[2*(j*w+i)];
        inW[2*i+1] = outputSignal[2*(j*w+i)+1];
      }

      SerialInverseFFT1D(inW, outW, w);

      for(int i=0; i<w; i++)
      {
        outputSignal[2*(j*w+i)] = outW[2*i];
        outputSignal[2*(j*w+i)+1] = outW[2*i+1];
      }
    }

    delete[] inW;
```

```
    delete[] inH;
    delete[] outW;
    delete[] outH;
  }
}
```

18. Библиотеки для вычисления БПФ

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\14. filter (on the fly)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\14. filter (on the fly)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 2.9с (Рис. 99). Выигрыш по сравнению с предыдущей версией составил 35.2%.

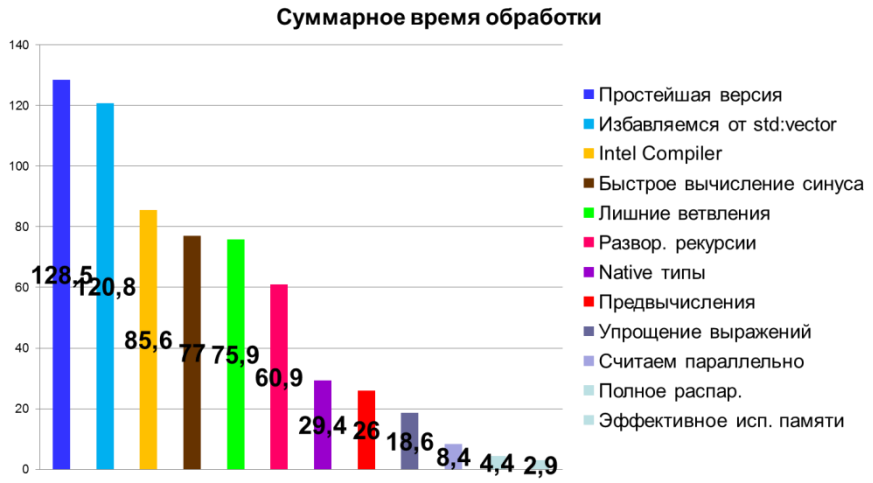


Рис. 99. Прогресс оптимизации

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo2.avi -s -gl**”.

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 16 с. Выигрыш по сравнению с предыдущей версией составил 35.6 %.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo3.avi -s -gl**”.

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 121.6с. Выигрыш по сравнению с предыдущей версией составил 58.5%.

Общепризнанным стандартом вычисления БПФ на данный момент является библиотека FFTW (Fastest Fourier Transform in the West). Эта библиотека предназначена для вычисления ДПФ и распространяется по лицензии GNU GPL.

Библиотека Intel MKL (Math Kernel Library) содержит реализации большого количества алгоритмов оптимизированных для процессоров Intel. Также эта библиотека имеет реализацию ДПФ (в том числе и интерфейсов FFTW).

Для того чтобы включить поддержку библиотеки MKL в проекте, необходимо зайти в меню **Select Build Components** (Рис. 100).

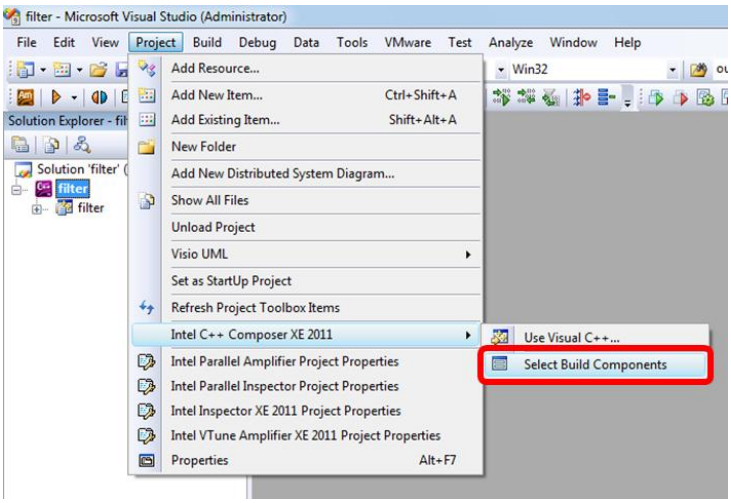


Рис. 100.

В меню **Select Build Components** можно указать какую версию MKL вы хотите использовать: последовательную или параллельную (Рис. 101).

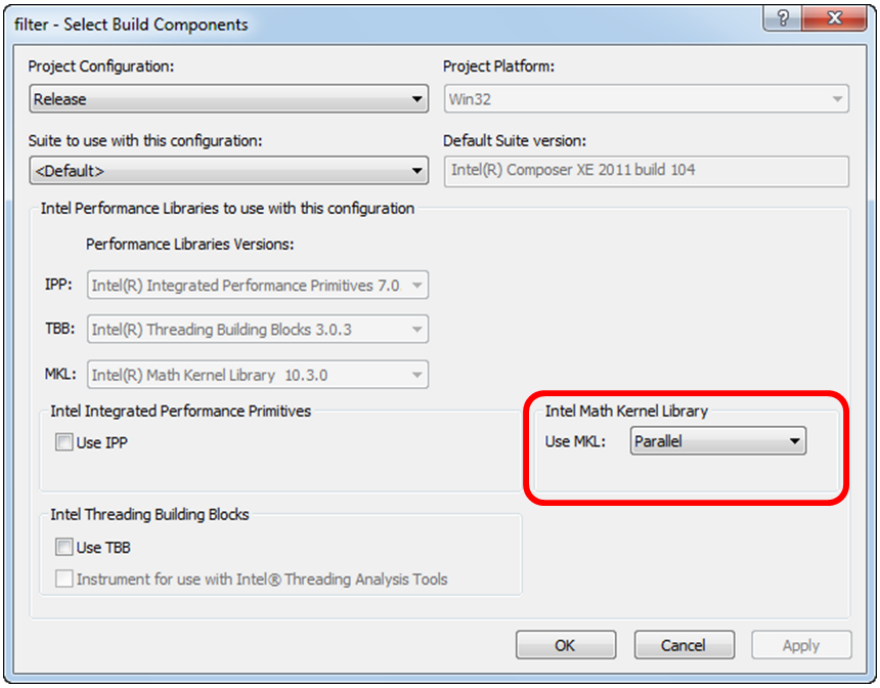


Рис. 101.

Среду разработки также можно настроить вручную, без использования мастера. Для этого необходимо:

- указать путь к заголовочным файлам в меню **Tools → Options…→ Projects and Solutions → VC++ Directories → Show directories for: Include files → New Line**: C:\Program Files\Intel\MKL\<version>\include;
- указать путь к статическим библиотекам для линковки в меню **Tools → Options…→ Projects and Solutions → VC++ Directories** → Show directories for: **Library files → New Line**: C:\Program Files\Intel\MKL\<version>\<arch>\lib, где <arch> = ia32 | em64t | ia64.

Далее в проекте необходимо подключить требуемые статические библиотеки для линковки:

- Базовая библиотека: **mkl_core.lib**
- Версия MKL:

  параллельная: **mkl_intel_thread.lib**

  последовательная: **mkl_sequential.lib**
- Тип интерфейса функций:

  cdecl: **mkl_intel_c.lib**

  stdcall: **mkl_intel_s.lib**

Для того чтобы выполнить ДПФ с помощью библиотеки MKL, необходимо подключить заголовочный файл **mkl_dfti.h** и вызвать функции в следующей последовательности:

1. Создание дескриптора, описывающего ДПФ (**`DftiCreateDescriptor`**).
2. Подтверждение дескриптора (**`DftiCommitDescriptor`**).
3. Вычисление прямого и/или обратного ДПФ (**`DftiComputeForward`**, **`DftiComputeBackward`**).
4. Освобождение дескриптора (**`DftiFreeDescriptor`**).

Создание дескриптора для вычисления одномерного ДПФ выполняется с помощью следующего кода:

```
DFTI_DESCRIPTOR_HANDLE dftHandle;
MKL_LONG status;
status = DftiCreateDescriptor( &dftHandle, DFTI_DOUBLE,
                               DFTI_COMPLEX, 1, size);
status = DftiCommitDescriptor( dftHandle );
```

Основные элементы кода:

- **`dftHandle`** – дескриптор.
- **`status`** – статус выполнения функции (**`status = 0`**, если операция выполнена успешно, в противном случае, используя функцию

DftiErrorMessage, можно получить содержательное сообщение о типе ошибки).

- **DFTI_DOUBLE** – тип элементов, над которыми будет выполняться ДПФ – элементы двойной точности (может быть **DFTI_SINGLE** – одинарная точность).
- **DFTI_COMPLEX** – комплексная область (может быть **DFTI_REAL** – действительная область).
- **1** – размерность ДПФ.
- **size** – количество элементов.

Создание дескриптора для вычисления двумерного ДПФ будет немного отличаться:

```
DFTI_DESCRIPTOR_HANDLE dftHandle;
MKL_LONG status;
int size[2] = {w,h};
status = DftiCreateDescriptor(&dftHandle, DFTI_DOUBLE,
  DFTI_COMPLEX, 2, size);
status = DftiCommitDescriptor(dftHandle);
```

Вычисление прямого ПФ происходит с помощью следующего кода:

```
status = DftiComputeForwarddftHandle, mas);
```

Вычисление обратного ПФ происходит с помощью следующего кода:

```
status = DftiComputeBackward (dftHandle, mas);
```

При выполнении обратного ДПФ по умолчанию функция **DftiComputeBackward()** не выполняет деление на количество элементов (Рис. 102).

$$x_p = \frac{1}{n}\sum_{p=0}^{n-1} y_p e^{\frac{kp}{n}2\pi i}, k = \overline{0, n-1}$$

Рис. 102. Обратное преобразование Фурье

Можно реализовать деление вручную после выполнения обратного ДПФ или перед подтверждением дескриптора с помощью операции **DftiCommitDescriptor()**, необходимо установить коэффициент масштабирования:

```
DftiSetValue(handle, DFTI_BACKWARD_SCALE, 1.0 / (double)n);
```

Освобождение дескриптора выполняется с помощью следующей команды:

```
status = DftiFreeDescriptor(&dftHandle);
```

Замените использование собственной разработки БПФ на реализацию из библиотеки MKL.

Продолжайте работать с теми исходными кодами, которые были получены после выполнения предыдущего этапа, или откройте проект **filter.sln** в директории **C:\ParallelCalculus\05_FastFourierTransform\15. filter (mkl)**, в котором уже выполнены указанные выше действия:

- запустите приложение **Microsoft Visual Studio 2008**;
- в меню **File** выполните команду **Open→Project/Solution…**;
- в диалоговом окне **Open Project** выберите папку **C:\ParallelCalculus\05_FastFourierTransform\15. filter (mkl)**;
- дважды щелкните на файле **filter.sln** или, выбрав файл, выполните команду **Open**.

Выполните сборку **Release**-версии проекта с помощью команды **Build Solution** в меню **Build**. Убедитесь, что проект не содержит ошибок компиляции и в результате линковки собирается исполняемый модуль **filter.exe**.

Запустите собранное приложение через командную строку: “**filter.exe ..\Videos\TestVideo1.avi -s -gl**”. Для этого выполните следующую последовательность действий:

- в «Проводнике» Windows (Windows Explorer) откройте директорию **Release**, содержащую собранный исполняемый модуль;
- в адресной строке «Проводника» Windows наберите команду **cmd** и нажмите **Enter** (Рис. 27);
- в запущенном интерпретаторе командной строки наберите “**filter.exe ..\Videos\TestVideo1.avi -s -gl**” и нажмите **Enter**;
- дождитесь завершения программы и занесите время суммарной фильтрации кадров (Total processing time) в таблицу (Таблица 4).

На рассматриваемой тестовой системе (Таблица 1) суммарное время обработки кадров составило 1.6 с. Выигрыш по сравнению с предыдущей версией составил 43.9 %.

19. Дополнительные задания

Дополнительные задания имеют разный уровень сложности. Некоторые из них являются достаточно трудоемкими и подходят в качестве тем зачетных для студентов, изучающих оптимизацию и параллельные численные методы.

1. Перейти от использования типа **`double`** к типу **`float`**. Оценить корректность получаемых данных и эффективность реализации.
2. Реализовать одну функцию вычисления БПФ для всех трёх компонент цвета. Оценить эффективность реализации.
3. Реализовать функцию, выполняющую перестановку элементов массива на основании таблицы с реверсивными двухбайтовыми числами (таблица на 65536 записей). Оценить эффективность реализации.
4. Реализовать функцию, выполняющую перестановку элементов массива сразу для всех строк/столбцов обрабатываемого изображения. Оценить эффективность реализации.
5. Реализовать in-place алгоритм БПФ без использования перестановки элементов массива. Оценить эффективность реализации.
6. Реализовать алгоритм БПФ на общий случай размера входного сигнала из [4]. Оценить эффективность реализации.
7. Оценить эффективность динамического планирования в OpenMP-версии по сравнению со статическим.
8. Реализовать одну глобальную параллельную секцию в программе в функции **`ProcessFrame`**. Оценить эффективность параллельной версии.

20. Литература

20.1. Использованные источники информации

1. Кудрявцев Л. Д. Краткий курс математического анализа. Т. 2. Диффефренциальное и интегральное исчисления функций многих переменных. Гармонический анализ: Учебник. — 3-е изд., перераб. — М.: ФИЗМАТЛИТ, 2005. — 424 с.
2. Тропченко А.Ю., Тропченко А.А. Методы сжатия изображений, аудиосигналов и видео. Учебное пособие по дисциплине «Теоретическая информатика» - Санкт-Петербург: , 2009. — 108 с.
3. Гонсалес Р., Вудс Р. Цифровая обработка изображений. Москва: Техносфера, 2005 г. — 1072 с.

20.2. Дополнительная литература

4. Нуссбаумер Г. Быстрое преобразование Фурье и алгоритмы вычисления сверток: Пер. с англ. — М.: Радио и связь, 1985. — 248 с.

5. Abramowitz, Milton; Stegun, Irene A. Handbook of mathematical functions: with formulas, graphs, and mathematical tables. Washington, D.C.: U.S. Dept. of Commerce, National Bureau of Standards, U.S. G.P.O., 1970.
6. Аммерааль Л. STL для программистов на C++. Пер. с англ. — М.: ДМК, 1999 – 240 с.
7. Левин М. П. Параллельное программирование с использованием OpenMP: учебное пособие. – М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2008. – 118 с.
8. Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley (русский перевод Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003).

20.3. Информационные ресурсы сети Интернет

9. Теория и практика цифровой обработки сигналов. Дискретное преобразование Фурье. [http://dsplib.ru/content/dft/dft.html].
10. FM680 Virtex-6™ XMC & PMC with User Definable Memory & I/O. [http://www.4dsp.com/FM680.php].

21. Приложения

Таблица 4. Результаты экспериментов

Версия программы	Суммарное время фильтрации кадров, с	Время обработки одного кадра, с

21.1. Исходная версия программы

fft.h

```
#ifndef _FFT_H
#define _FFT_H

#include <complex>
#include <vector>

#define PI (3.14159265358979323846)

void SerialFFT2D(
   vector<complex<double>> inputSignal,
   vector<complex<double>> &outputSignal, int w, int h);

void SerialInverseFFT2D(
   vector<complex<double>> inputSignal,
   vector<complex<double>> &outputSignal, int w, int h);

#endif
```

fft.cpp

```
#include "fft.h"

void BitReversing(
   vector<complex<double>> &inputSignal,
   vector<complex<double>> &outputSignal,
   int size, int step, int offset)
{
```

```
  int bitsCount = 0;

  //Определение количества бит для бит-реверсирования.
  //Получается, что bitsCount = log2(size).
  for( int tmp_size = size; tmp_size > 1;
       tmp_size/=2, bitsCount++ );

  //Выполнение бит-реверсирования
  //ind - индекс элемента в  массиве input
  //revInd - соответствующий индексу ind индекс
  // (бит-реверсивный) в массиве output
  for(int ind = 0; ind < size; ind++)
  {
    int mask = 1 << (bitsCount - 1);
    int revInd = 0;

    for(int i=0; i<bitsCount; i++)
    {
         bool val = ind & mask;
         revInd |= val << i;
         mask = mask >> 1;
    }

    outputSignal[revInd*step + offset] =
       inputSignal[ind*step + offset];
  }
}

void Butterfly(
   vector<complex<double>> &signal, complex<double> u,
   int offset, int butterflySize, int step, int off)
{
  complex<double> tem =
     signal[off + step*(offset +  butterflySize)] * u;

  signal[off + step*(offset +  butterflySize)] =
     signal[off + step*(offset)] - tem;

  signal[off + step*(offset)] += tem;
}

void SerialFFTCalculation(vector<complex<double>> &signal,
   int first, int size, int step, int offset,
   bool forward=true)
{
  if(size==1)
    return;

  double const coeff=2.0*PI/size;
```

```
  SerialFFTCalculation( signal, first, size/2,
     step, offset);
  SerialFFTCalculation( signal, first + size/2, size/2,
     step, offset, forward);

  for (int  j=first; j<first+size/2; j++ )
    if(forward)
      Butterfly(signal,
         complex<double>(cos(-j*coeff), sin(-j*coeff)),
         j , size/2, step, offset );
    else
      Butterfly(signal,
         complex<double>(cos(j*coeff), sin(j*coeff)),
         j , size/2, step, offset );
}

void SerialFFT1D(vector<complex<double>> &inputSignal,
   vector<complex<double>> &outputSignal, int size,
   int step, int offset)
{
  BitReversing(inputSignal, outputSignal, size,
     step, offset);

  SerialFFTCalculation(outputSignal, 0, size,
     step, offset);
}

void SerialFFT2D(vector<complex<double>> inputSignal,
   vector<complex<double>> &outputSignal, int w, int h)
{
  vector<complex<double>> tem(w*h);

  for(int i=0; i<w; i++)
    SerialFFT1D(inputSignal, tem, h, w, i);


  for(int j=0; j<h; j++)
    SerialFFT1D(tem, outputSignal, w, 1, h*j);
}

void SerialInverseFFT1D(
   vector<complex<double>> &inputSignal,
   vector<complex<double>> &outputSignal,
   int size, int step, int offset)
{
  BitReversing(inputSignal, outputSignal, size,
     step, offset);

  SerialFFTCalculation(outputSignal, 0, size,
```

```
     step, offset, false);

  for (int  j=0; j<size; j++ )
    outputSignal[offset + step*j]/=size;
}

void SerialInverseFFT2D(
   vector<complex<double>> inputSignal,
   vector<complex<double>> &outputSignal, int w, int h)
{
  vector<complex<double>> tem(w*h);

  for(int i=0; i<w; i++)
    SerialInverseFFT1D(inputSignal, tem, h, w, i);

  for(int j=0; j<h; j++)
    SerialInverseFFT1D(tem, outputSignal, w, 1, h*j);
}
```

filter.cpp

```
//OpenCV
#include "cxcore.h"
#include "highgui.h"

//IO and time
#include "stdio.h"
#include "windows.h"
#include <conio.h>

#include <complex>
#include <vector>

#include "fft.h"


#pragma comment(lib,"cxcore.lib")
#pragma comment(lib,"highgui.lib")

#define _WAIT_TIME 30

char helpString[] =
"\nUsage: \n\
\tfilter.exe <input video file> [options]\n\n\
Options:\n\
\t-s \
\t silent mode.\n\
\
\t-gl (default) \
\t Gauss lowpass filter.\n\
\
```

```
\t-gh \
\t Gauss highpass filter.\n\
\
\t-bl \
\t Battervort lowpass filter.\n\
\
\t-bh \
\t Battervort highpass filter.\n\
\n\
Example:\n\
\tfilter.exe video.avi -s -gh\
";

enum FilterType {GaussLow=0, GaussHigh=1,
                 BattervortLow=2, BattervortHigh=3};
char filters[4][50] = {"Gauss lowpass filter",
                       "Gauss highpass filter",
                       "Battervort lowpass filter",
                       "Battervort highpass filter"};
IplImage *inpFrame = NULL, *outFrame = NULL,
         *filter = NULL, *freq = NULL, *empty = NULL;
int chW = 0, chC = 0;
int frame=0, frames;
int r = 30
bool pause = false;
HANDLE threadInitMutex;
FilterType fType = GaussLow;

void ProcessFrame(IplImage *frame, IplImage *outFrame,
   IplImage* filter, IplImage* freq, FilterType t, int *_r)
{
  int w = frame->width,
      h = frame->height,
      r = * r,
      size = w * h;

  vector<double> f(size);
  vector<complex<double>> inpR, inpG, inpB,
     outR(size), outG(size), outB(size);

  double maxR = 0.0 , maxG = 0.0, maxB = 0.0;
  double minR = 0.0 , minG = 0.0, minB = 0.0;

  //Fill inp<R|G|B> arrays
  for (int i=0; i<size; i++)
    inpB.push_back(pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 0]);

  for (int i=0; i<size; i++)
```

```
    inpG.push_back(pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 1]);

  for (int i=0; i<size; i++)
    inpR.push_back(pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 2]);

  //Make filter
  for (int i=0; i<w; i++)
    for (int j=0;j<h; j++)
      switch(t)
      {
        case GaussLow:
          f[i+j*w]=
             exp( -(pow(double(i-w/2), 2) +
                    pow(double(j-h/2), 2) ) /
             (2 * pow((double)r, 2) ) );
          break;
        case GaussHigh:
          f[i+j*w]=
             1 - exp( -(pow(double(i-w/2), 2) +
                        pow(double(j-h/2), 2) ) /
             (2 * pow((double)r, 2) ) );
          break;
        case BattervortLow:
          f[i+j*w]=
             1 / ( 1 + (pow(double(i-w/2), 2) +
                        pow(double(j-h/2), 2) ) /
             pow((double)r, 2) );
          break;
        case BattervortHigh:
          f[i+j*w]=
             1 / ( 1 + pow((double)r, 2) /
             (pow(double(i-w/2), 2) +
              pow(double(j-h/2), 2) ) );
          break;
      };

  //Convert to frequency area
  SerialFFT2D(inpR, outR, w, h);
  SerialFFT2D(inpG, outG, w, h);
  SerialFFT2D(inpB, outB, w, h);

  //Show filter
  for(int i=0; i<size; i++)
    filter->imageData[i] = (char) (255 * f[i]);

  //Apply filter
  for (int i=0; i<size; i++)
    outR[i] = outR[i] * f[i];
```

```
for (int i=0; i<size; i++)
  outG[i] = outG[i] * f[i];

for (int i=0; i<size; i++)
  outB[i] = outB[i] * f[i];

//Find max value

for (int i=0; i<size; i++)
  if (sqrt(outR[i].real()*outR[i].real()+
           outR[i].imag()*outR[i].imag()) != 0)
  {
    maxR = minR =
       log( sqrt(outR[i].real()*outR[i].real()+
                 outR[i].imag()*outR[i].imag()));
    break;
  }

for (int i=0; i<size; i++)
{
  if (sqrt(outR[i].real()*outR[i].real()+
           outR[i].imag()*outR[i].imag()) != 0)
    if(maxR <
       log( sqrt(outR[i].real()*outR[i].real()+
                 outR[i].imag()*outR[i].imag()) ) )
      maxR =
         log( sqrt(outR[i].real()*outR[i].real()+
                   outR[i].imag()*outR[i].imag()) );

  if (sqrt(outR[i].real()*outR[i].real()+
           outR[i].imag()*outR[i].imag()) != 0)
    if(minR >
       log( sqrt(outR[i].real()*outR[i].real()+
                 outR[i].imag()*outR[i].imag()) ) )
      minR =
         log( sqrt(outR[i].real()*outR[i].real()+
                   outR[i].imag()*outR[i].imag()) );
}

for (int i=0; i<size; i++)
  if (sqrt(outG[i].real()*outG[i].real()+
           outG[i].imag()*outG[i].imag()) != 0)
  {
    maxG = minG =
       log( sqrt(outG[i].real()*outG[i].real()+
                 outG[i].imag()*outG[i].imag()) );
    break;
  }
```

```
for (int i=0; i<size; i++)
{
  if (sqrt(outG[i].real()*outG[i].real()+
           outG[i].imag()*outG[i].imag()) != 0)
    if(maxG <
       log( sqrt(outG[i].real()*outG[i].real()+
                 outG[i].imag()*outG[i].imag()) ) )
      maxG =
         log( sqrt(outG[i].real()*outG[i].real()+
                   outG[i].imag()*outG[i].imag()) ) ;

  if (sqrt(outG[i].real()*outG[i].real()+
           outG[i].imag()*outG[i].imag()) != 0)
    if(minG >
       log( sqrt(outG[i].real()*outG[i].real()+
                 outG[i].imag()*outG[i].imag()) ) )
      minG =
         log( sqrt(outG[i].real()*outG[i].real()+
                   outG[i].imag()*outG[i].imag()) ) ;
}

for (int i=0; i<size; i++)
  if (sqrt(outB[i].real()*outB[i].real()+
           outB[i].imag()*outB[i].imag()) != 0)
  {
    maxB = minB =
       log( sqrt(outB[i].real()*outB[i].real()+
                 outB[i].imag()*outB[i].imag()));
    break;
  }

for (int i=0; i<size; i++)
{
  if (sqrt(outB[i].real()*outB[i].real()+
           outB[i].imag()*outB[i].imag()) != 0)
    if(maxB <
       log( sqrt(outB[i].real()*outB[i].real()+
                 outB[i].imag()*outB[i].imag()) ) )
      maxB =
         log( sqrt(outB[i].real()*outB[i].real()+
              outB[i].imag()*outB[i].imag()) );

  if (sqrt(outB[i].real()*outB[i].real()+
           outB[i].imag()*outB[i].imag()) != 0)
    if(minB >
       log( sqrt(outB[i].real()*outB[i].real()+
                 outB[i].imag()*outB[i].imag()) ) )
      minB =
         log( sqrt(outB[i].real()*outB[i].real()+
```

```
                    outB[i].imag()*outB[i].imag()) );
}

//Show frequency area with filter
for (int i=0; i<size; i++)
  if (sqrt(outR[i].real()*outR[i].real()+
           outR[i].imag()*outR[i].imag()) != 0)
    freq->imageData[3*i+0] = (char) (255 *
       ( (log( sqrt(outR[i].real()*outR[i].real()+
                  outR[i].imag()*outR[i].imag())) -
        minR ) / (maxR - minR) ) );
  else
    freq->imageData[3*i+0] = 0;

for (int i=0; i<size; i++)
  if (sqrt(outG[i].real()*outG[i].real()+
           outG[i].imag()*outG[i].imag()) != 0)
    freq->imageData[3*i+1] = (char) (255 *
       ( (log( sqrt(outG[i].real()*outG[i].real()+
                    outG[i].imag()*outG[i].imag())) -
          minG ) / (maxG - minG) ) );
  else
    freq->imageData[3*i+1] = 0;

for (int i=0; i<size; i++)
  if (sqrt(outB[i].real()*outB[i].real()+
           outB[i].imag()*outB[i].imag()) != 0)
    freq->imageData[3*i+2] = (char) (255 *
       ( (log( sqrt(outB[i].real()*outB[i].real()+
                    outB[i].imag()*outB[i].imag())) -
          minB ) / (maxB - minB) ) );
  else
    freq->imageData[3*i+2] = 0;

//Convert to frequency area
SerialInverseFFT2D(outR, inpR, w, h);
SerialInverseFFT2D(outG, inpG, w, h);
SerialInverseFFT2D(outB, inpB, w, h);

for (int i=0; i<size; i++)
  outFrame->imageData[3*i+0] =
     (char) (pow(-1.0,i/w+i%w) * inpB[i].real());

for (int i=0; i<size; i++)
  outFrame->imageData[3*i+1] =
     (char) (pow(-1.0,i/w+i%w) * inpG[i].real());

for (int i=0; i<size; i++)
  outFrame->imageData[3*i+2] =
```

```
        (char) (pow(-1.0,i/w+i%w) * inpR[i].real());
}

void OnWindowClick(int ev, int x, int y, int flags,
                   void* param)
{
  if(ev == CV_EVENT_LBUTTONDOWN)
  {
    if(*(bool*)param)
      printf("Resume!\n");
    else
      printf("Pause!!!\n");

    *(bool*)param = !(*(bool*)param);
  }
}

DWORD WINAPI DrawingFunction( LPVOID lpParam )
{
  cvNamedWindow("Output",CV_WINDOW_AUTOSIZE);
  cvShowImage("Output", empty);
  cvCreateTrackbar( "Frame", "Output", &frame, frames,
                    NULL );
  cvMoveWindow("Output", 50, 200);

  char fStr[100];
  sprintf(fStr, "Filter: %s", filters[fType]);
  cvNamedWindow(fStr,CV_WINDOW_AUTOSIZE);
  cvShowImage(fStr, empty);
  cvCreateTrackbar( "Radius", fStr, &r,
                    cvGetSize(empty).width, 0 );
  cvMoveWindow(fStr, 400, 200);

  cvNamedWindow("Frequency area",CV WINDOW AUTOSIZE);
  cvMoveWindow("Frequency area", 750, 200);

  cvSetMouseCallback("Output", OnWindowClick, &pause);
  cvSetMouseCallback(fStr, OnWindowClick, &pause);
  cvSetMouseCallback("Frequency area", OnWindowClick,
                     &pause);

  ReleaseSemaphore(threadInitMutex, 1, NULL);

  while(frame != frames)
  {
    cvShowImage("Output", outFrame);
    cvShowImage(fStr, filter);
     cvShowImage("Frequency area", freq);
    chW = cvWaitKey(_WAIT_TIME);
```

```
    if( chW == 27 || chC == 27)  //Esc
      break;
  }

  cvDestroyWindow("Frequency area");
  cvDestroyWindow("Output");

  return 0;
}

int main(int argc, char** argv)
{
  const char* fileName =
     argc>=2 ? argv[1] : "Videos\\TestVideo1.avi";
  CvCapture* film = NULL;
  int w, h;
  int msec;
  LARGE_INTEGER start, finish, totStart;
  LARGE_INTEGER fr;
  double wrkTime = 0.0;
  CvSize size;
  HANDLE threadH;
  DWORD threadId;
  char codec[5];
  bool silent = false;
  FILE *f;

  //1. Parse each string in "argv" array
  for(int i=2; i < argc; i++)
  {
    //Silent mode
    if( string(argv[i]) == "-s" )
    {
      silent = true;
      continue;
    }

    //Gauss lowpass filter
    if( string(argv[i]) == "-gl" )
    {
      fType = GaussLow;
      continue;
    }

    //Gauss highpass filter
    if( string(argv[i]) == "-gh" )
    {
      fType = GaussHigh;
      continue;
```

```
    }

    //Battervort lowpass filter
    if( string(argv[i]) == "-bl" )
    {
      fType = BattervortLow;
      continue;
    }

    //Battervort highpass filter
    if( string(argv[i]) == "-bh" )
    {
      fType = BattervortHigh;
      continue;
    }

    printf("%s\n", helpString);
    exit(1);


  //2.1. Video file is present?
  printf("Video file: %s\n", fileName);
  f = fopen(fileName,"r");
  if (f == NULL)
  {
    printf("Error: File not found!\n");
    printf("%s\n", helpString);
    return -1;
  }


  //2.2. Load video file
  film=cvCreateFileCapture(fileName);
  if(!film)
  {
    printf("Error: Unsupported video codec!\n");
    pnintf("%s\n", helpString);
    return -1;
  }

  QueryPerformanceFrequency(&fr);

  //3. Get video info
  frames =
     cvGetCaptureProperty(film, CV_CAP_PROP_FRAME_COUNT);
  int val =
     (int) cvGetCaptureProperty(film, CV_CAP_PROP_FOURCC);
  memcpy(codec, &val, 4);
  codec[4]=0;
```

```
w = (int) cvGetCaptureProperty(film,
                               CV_CAP_PROP_FRAME_WIDTH);
h = (int) cvGetCaptureProperty(film,
                               CV_CAP_PROP_FRAME_HEIGHT);
size.height = h;
size.width = w;

//4.1. Check up height and width for correct value
int bits = 0;
for( int tmp = h; tmp > 1; tmp/=2, bits++);
if( h != 1<<bits )
{
  printf("Error: Height should be power of 2!\n");
  cvReleaseCapture(&film);
  return -1;
}

bits = 0;
for( int tmp = w; tmp > 1; tmp/=2, bits++);
if( w != 1<<bits )
{
  printf("Error: Width should be power of 2!\n");
  cvReleaseCapture(&film);
  return -1;
}

//4.2. Create mutex for thread init
threadInitMutex = CreateSemaphore(NULL, 0, 1, NULL);
if(threadInitMutex==NULL)
{
  printf("Error: CreateMutex error: %d\n",
         GetLastError());
  return -1;
}

//5. Print info
printf("Loaded!\n");
printf("Codec: \t%s\n", codec);
printf("Total frames: %i\n", frames);
printf("Picture size: %i x %i\n", w, h);
printf("Filter: %s\n" filters[fType]);
printf("\n");
printf("Click on window to \"Pause\"\n");
printf("Press ESC to \"Exit\"\n");
printf("\n");

//6. Create images
empty = cvCreateImage(size, IPL_DEPTH_8U, 1);
outFrame = cvCreateImage(size, IPL_DEPTH_8U, 3);
```

```
filter = cvCreateImage(size, IPL_DEPTH_8U, 1);
freq = cvCreateImage(size, IPL_DEPTH_8U, 3);

//7. Create and wait thread for drawing
if (!silent)
{
  threadH = CreateThread(NULL, 0, DrawingFunction, NULL,
                         0, &threadId);
  WaitForSingleObject(threadInitMutex, INFINITE);
}

printf("Log:\n");

//8. Variable of start time is installed
QueryPerformanceCounter(&totStart);

//9. Process frames
while(frame != frames)
{
  //9.1. Get next frame
  cvSetCaptureProperty( film, CV_CAP_PROP_POS_FRAMES,
                        frame);
  msec = (int)cvGetCaptureProperty( film,
                                    CV_CAP_PROP_POS_MSEC);
  inpFrame = cvQueryFrame(film);

  //9.2. Process current frames
  QueryPerformanceCounter(&start);
  ProcessFrame(inpFrame, outFrame, filter, freq, fType,
               &r);
  QueryPerformanceCounter(&finish);

  //9.3. Increase count of total frames
  totFrames++;

  //9.4. Compute and print time
  printf("Frame: %5i, position: %6i ms, processing time:
         %f \n", frame, msec,
         (start.QuadPart - finish.QuadPart) /
         (double)fr.QuadPart);
  wrkTime += (finish.QuadPart-start.QuadPart) /
             (double)fr.QuadPart;

  //9.5. "ESC" is pressed?
  if(_kbhit())
    chC = _getch();

  //9.6. The thread is completed?
  if( WaitForSingleObject(threadH, 1) == WAIT_OBJECT_0)
    break;
```

```
     //9.7. Break if ESC pressed
    if (silent)
      if( chC == 27 )  //Esc
        break;

    //9.8. Compute next frame and move trackbar
    if(!pause)
    {
      frame++;
      cvSetTrackbarPos( "Frame", "Output", frame);
    }
  }

  QueryPerformanceCounter(&finish);

  //10. Print summary
  printf("Done!\n");
  printf("\n");
  printf("Total time: %f\n",
         (finish.QuadPart-totStart.QuadPart) /
         (double)fr.QuadPart);
  printf("Total frames: %i\n", totFrames);
  printf("Total processing time: %f\n", wrkTime);
  if(totFrames)
    printf("Average processing time per frame: %f\n",
           wrkTime/totFrames);

  //11. Release resources
  if (!silent)
    CloseHandle(threadH);


  cvReleaseImage(&freq);
  cvReleaseImage(&outFrame);
  cvReleaseImage(empty);

  CloseHandle(threadInitMutex);

  return 0;
}
```

21.2. Корректная версия программы

fft.h

```
#ifndef _FFT_H
#define _FFT_H
```

```
#include <complex>
#include <vector>

using namespace std;

#define PI (3.14159265358979323846)

void SerialFFT2D(
                 vector<complex<double>> inputSignal,
                 vector<complex<double>> &outputSignal,
                 int w, int h);

void SerialInverseFFT2D(
                 vector<complex<double>> inputSignal,
                 vector<complex<double>> &outputSignal,
                 int w, int h);

#endif
```

fft.cpp

```
#include "fft.h"

void BitReversing(vector<complex<double>> &inputSignal,
                  vector<complex<double>> &outputSignal,
                  int size, int step, int offset)
{
  int bitsCount = 0;

  //Определение  количества  бит  для  бит-реверсирования.
Получается, что bitsCount = log2(size).
  for( int tmp_size = size; tmp_size > 1;
       tmp_size/=2, bitsCount++ );

  //Выполнение бит-реверсирования
  //ind - индекс элемента в  массиве input
  //revInd - соответствующие индексу ind индекс
  // (бит-реверсивный) в массиве output
  for(int ind = 0; ind < size; ind++)
  {
    int mask = 1 << (bitsCount - 1);
    int revInd = 0;

    for(int i=0; i<bitsCount; i++)
    {
        bool val = ind & mask;
        revInd |= val << i;
        mask = mask >> 1;
    }
```

```
    outputSignal[revInd*step + offset] =
       inputSignal[ind*step + offset];
  }
}

void Butterfly(vector<complex<double>> &signal,
               complex<double> u, int offset,
               int butterflySize, int step, int off)
{
  complex<double> tem =
     signal[off + step*(offset +  butterflySize)] * u;

  signal[off + step*(offset +  butterflySize)] =
     signal[off + step*(offset)] - tem;

  signal[off + step*(offset)] += tem;
}

void SerialFFTCalculation(vector<complex<double>> &signal,
                          int first, int size, int step,
                          int offset, bool forward=true)
{
  if(size==1)
    return;

  double const coeff=2.0*PI/size;

  SerialFFTCalculation(signal, first, size/2,
                       step, offset, forward);
  SerialFFTCalculation(signal, first + size/2,
                       size/2, step, offset, forward);

  for (int  j=first; j<first+size/2; j++ )
    if(forward)
      Butterfly(signal,
         complex<double>(cos(-j*coeff), sin(-j*coeff)),
         j , size/2, step, offset );
    else
      Butterfly(signal,
         complex<double>(cos(j*coeff), sin(j*coeff)),
         j , size/2, step, offset );
}

void SerialFFT1D(vector<complex<double>> &inputSignal,
                 vector<complex<double>> &outputSignal,
                 int size, int step, int offset)
{
  BitReversing(inputSignal, outputSignal, size,
               step, offset);
```

```
  SerialFFTCalculation(outputSignal, 0, size,
                       step, offset);
}

void     SerialFFT2D(vector<complex<double>>     inputSignal,
vector<complex<double>> &outputSignal, int w, int h)
{
  vector<complex<double>> tem(w*h);

  for(int i=0; i<w; i++)
    SerialFFT1D(inputSignal, tem, h, w, i);


  for(int j=0; j<h; j++)
    SerialFFT1D(tem, outputSignal, w, 1, h*j);
}

void SerialInverseFFT1D(
    vector<complex<double>> &inputSignal,
    vector<complex<double>> &outputSignal,
    int size, int step, int offset)
{
  BitReversing(inputSignal, outputSignal, size,
               step, offset);
  SerialFFTCalculation(outputSignal, 0, size,
                       step, offset, false);

  for (int  j=0; j<size; j++ )
    outputSignal[offset + step*j]/=size;
}

void SerialInverseFFT2D(
   vector<complex<double>> inputSignal,
   vector<complex<double>> &outputSignal, int w, int h)
{
  vector<complex<double>> tem(w*h);

  for(int i=0; i<w; i++)
    SerialInverseFFT1D(inputSignal, tem, h, w, i);

  for(int j=0; j<h; j++)
    SerialInverseFFT1D(tem, outputSignal, w, 1, h*j);
}
```

filter.cpp

```
//OpenCV
#include "cxcore.h"
#include "highgui.h"

//IO and time
```

```
#include "stdio.h"
#include "windows.h"
#include <conio.h>

#include <complex>
#include <vector>

#include "fft.h"

using namespace std;

#pragma comment(lib,"cxcore.lib")
#pragma comment(lib,"highgui.lib")

#define _WAIT_TIME 30

char helpString[] =
"\nUsage: \n\
\tfilter.exe <input video file> [options]\n\n\
Options:\n\
\t-s \
\t silent mode.\n\
\
\t-gl (default) \
\t Gauss lowpass filter.\n\
\
\t-gh \
\t Gauss highpass filter.\n\
\
\t-bl \
\t Battervort lowpass filter.\n\
\
\t-bh \
\t Battervort highpass filter.\n\
\n\
Example:\n\
\tfilter.exe video.avi -s -gh\
";

enum FilterType {GaussLow=0, GaussHigh=1, BattervortLow=2,
                 BattervortHigh=3};
char filters[4][50] = {"Gauss lowpass filter",
                       "Gauss highpass filter",
                       "Battervort lowpass filter",
                       "Battervort highpass filter"};
IplImage *inpFrame = NULL, *outFrame = NULL,
         *filter = NULL, *freq = NULL, *empty = NULL;
int chW = 0, chC = 0;
int frame=0, frames;
```

```
int r = 30;
bool pause = false;
HANDLE threadInitMutex;
FilterType fType = GaussLow;

void ProcessFrame(IplImage *frame, IplImage *outFrame,
                  IplImage* filter, IplImage* freq,
                  FilterType t, int *_r)
{
  int w = frame->width,
      h = frame->height,
      r = *_r,
      size = w * h;

  vector<double> f(size);
  vector<complex<double>> inpR, inpG, inpB,
     outR(size), outG(size), outB(size);

  double maxR = 0.0 , maxG = 0.0, maxB = 0.0;
  double minR = 0.0 , minG = 0.0, minB = 0.0;

  //Fill inp<R|G|B> arrays
  for (int i=0; i<size; i++)
    inpB.push_back(pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 0]);

  for (int i=0; i<size; i++)
    inpG.push_back(pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 1]);

  for (int i=0; i<size; i++)
    inpR.push_back(pow(-1.0,i/w+i%w) *
       ((unsigned char*)frame->imageData)[i*3 + 2]);

  //Make filter
  for (int i=0; i<w; i++)
    for (int j=0;j<h; j++)
      switch(t)
      {
        case GaussLow:
          f[i+j*w]=
             exp( -(pow(double(i-w/2), 2) +
                    pow(double(j-h/2), 2) ) /
             (2 * pow((double)r, 2) ) );
          break;
        case GaussHigh:
          f[i+j*w]=
             1 - exp( -(pow(double(i-w/2), 2) +
                        pow(double(j-h/2), 2) ) /
             (2 * pow((double)r, 2) ) );
```

```
        break;
      case BattervortLow:
        f[i+j*w]=
           1 / ( 1 + (pow(double(i-w/2), 2) +
                      pow(double(j-h/2), 2) ) /
           pow((double)r, 2) );
        break;
      case BattervortHigh:
        f[i+j*w]=
           1 / ( 1 + pow((double)r, 2) /
           (pow(double(i-w/2), 2) +
            pow(double(j-h/2), 2) ) );
        break;
    };

//Convert to frequency area
SerialFFT2D(inpR, outR, w, h);
SerialFFT2D(inpG, outG, w, h);
SerialFFT2D(inpB, outB, w, h);

//Show filter
for(int i=0; i<size; i++)
  filter->imageData[i] = (char) (255 * f[i]);

//Apply filter
for (int i=0; i<size; i++)
  outR[i] = outR[i] * f[i];

for (int i=0; i<size; i++)
  outG[i] = outG[i] * f[i];

for (int i=0; i<size; i++)
  outB[i] = outB[i] * f[i];

//Find max value

for (int i=0; i<size; i++)
  if (sqrt(outR[i].real()*outR[i].real()+
           outR[i].imag()*outR[i].imag()) != 0)
  {
    maxR = minR =
       log( sqrt(outR[i].real()*outR[i].real()+
                 outR[i].imag()*outR[i].imag()));
    break;
  }

for (int i=0; i<size; i++)
{
  if (sqrt(outR[i].real()*outR[i].real()+
```

```
            outR[i].imag()*outR[i].imag()) != 0)
    if(maxR <
       log( sqrt(outR[i].real()*outR[i].real()+
                 outR[i].imag()*outR[i].imag()) ) )
      maxR =
         log( sqrt(outR[i].real()*outR[i].real()+
                   outR[i].imag()*outR[i].imag()) );

  if (sqrt(outR[i].real()*outR[i].real()+
            outR[i].imag()*outR[i].imag()) != 0)
    if(minR >
       log( sqrt(outR[i].real()*outR[i].real()+
                 outR[i].imag()*outR[i].imag()) ) )
      minR =
         log( sqrt(outR[i].real()*outR[i].real()+
                   outR[i].imag()*outR[i].imag()) );
}

for (int i=0; i<size; i++)
  if (sqrt(outG[i].real()*outG[i].real()+
            outG[i].imag()*outG[i].imag()) != 0)
  {
    maxG = minG =
       log( sqrt(outG[i].real()*outG[i].real()+
                 outG[i].imag()*outG[i].imag()) );
    break;
  }

for (int i=0; i<size; i++)
{
  if (sqrt(outG[i].real()*outG[i].real()+
            outG[i].imag()*outG[i].imag()) != 0)
    if(maxG <
       log( sqrt(outG[i].real()*outG[i].real()+
                 outG[i].imag()*outG[i].imag()) ) )
      maxG =
         log( sqrt(outG[i].real()*outG[i].real()+
                   outG[i].imag()*outG[i].imag()) ) ;

  if (sqrt(outG[i].real()*outG[i].real()+
            outG[i].imag()*outG[i].imag()) != 0)
    if(minG >
       log( sqrt(outG[i].real()*outG[i].real()+
                 outG[i].imag()*outG[i].imag()) ) )
      minG =
         log( sqrt(outG[i].real()*outG[i].real()+
                   outG[i].imag()*outG[i].imag()) ) ;
}

for (int i=0; i<size; i++)
```

```
    if (sqrt(outB[i].real()*outB[i].real()+
             outB[i].imag()*outB[i].imag()) != 0)
    {
      maxB = minB =
         log( sqrt(outB[i].real()*outB[i].real()+
                   outB[i].imag()*outB[i].imag()));
      break;
    }

  for (int i=0; i<size; i++)
  {
    if (sqrt(outB[i].real()*outB[i].real()+
             outB[i].imag()*outB[i].imag()) != 0)
      if(maxB <
         log( sqrt(outB[i].real()*outB[i].real()+
                   outB[i].imag()*outB[i].imag()) ) )
        maxB =
           log( sqrt(outB[i].real()*outB[i].real()+
                outB[i].imag()*outB[i].imag()) );

    if (sqrt(outB[i].real()*outB[i].real()+
             outB[i].imag()*outB[i].imag()) != 0)
      if(minB >
         log( sqrt(outB[i].real()*outB[i].real()+
                   outB[i].imag()*outB[i].imag()) ) )
        minB =
           log( sqrt(outB[i].real()*outB[i].real()+
                     outB[i].imag()*outB[i].imag()) );
  }

  //Show frequency area with filter
  for (int i=0; i<size; i++)
    if (sqrt(outR[i].real()*outR[i].real()+
             outR[i].imag()*outR[i].imag()) != 0)
      freq->imageData[3*i+0] = (char) (255 *
         ( (log( sqrt(outR[i].real()*outR[i].real()+
                      outR[i].imag()*outR[i].imag())) -
          minR ) / (maxR - minR) ) );
    else
      freq->imageData[3*i+0] = 0;

  for (int i=0; i<size; i++)
    if (sqrt(outG[i].real()*outG[i].real()+
             outG[i].imag()*outG[i].imag()) != 0)
      freq->imageData[3*i+1] = (char) (255 *
         ( (log( sqrt(outG[i].real()*outG[i].real()+
                      outG[i].imag()*outG[i].imag())) -
           minG ) / (maxG - minG) ) );
    else
```

```
      freq->imageData[3*i+1] = 0;

  for (int i=0; i<size; i++)
    if (sqrt(outB[i].real()*outB[i].real()+
             outB[i].imag()*outB[i].imag()) != 0)
      freq->imageData[3*i+2] = (char) (255 *
         ( (log( sqrt(outB[i].real()*outB[i].real()+
                      outB[i].imag()*outB[i].imag())) -
            minB ) / (maxB - minB) ) );
    else
      freq->imageData[3*i+2] = 0;

  //Convert to frequency area
  SerialInverseFFT2D(outR, inpR, w, h);
  SerialInverseFFT2D(outG, inpG, w, h);
  SerialInverseFFT2D(outB, inpB, w, h);

  for (int i=0; i<size; i++)
    outFrame->imageData[3*i+0] =
       (char) (pow(-1.0,i/w+i%w) * inpB[i].real());

  for (int i=0; i<size; i++)
    outFrame->imageData[3*i+1] =
       (char) (pow(-1.0,i/w+i%w) * inpG[i].real());

  for (int i=0; i<size; i++)
    outFrame->imageData[3*i+2] =
       (char) (pow(-1.0,i/w+i%w) * inpR[i].real());
}

void OnWindowClick(int ev, int x, int y, int flags,
                   void* param)
{
  if(ev == CV EVENT LBUTTONDOWN)
  {
    if(*(bool*)param)
      printf("Resume!\n");
    else
      printf("Pause!!!\n");

    *(bool*)param = !(*(bool*)param);
  }
}

DWORD WINAPI DrawingFunction( LPVOID lpParam )
{
  cvNamedWindow("Output",CV_WINDOW_AUTOSIZE);
  cvShowImage("Output", empty);
  cvCreateTrackbar( "Frame", "Output", &frame, frames,
                    NULL );
```

```
  cvMoveWindow("Output", 50, 200);

  char fStr[100];
  sprintf(fStr, "Filter: %s", filters[fType]);
  cvNamedWindow(fStr,CV_WINDOW_AUTOSIZE);
  cvShowImage(fStr, empty);
  cvCreateTrackbar( "Radius", fStr, &r,
                  cvGetSize(empty).width, 0 );
  cvMoveWindow(fStr, 400, 200);

  cvNamedWindow("Frequency area",CV_WINDOW_AUTOSIZE);
  cvMoveWindow("Frequency area", 750, 200);

  cvSetMouseCallback("Output", OnWindowClick, &pause);
  cvSetMouseCallback(fStr, OnWindowClick, &pause);
  cvSetMouseCallback("Frequency area", OnWindowClick,
                    &pause);

  ReleaseSemaphore(threadInitMutex, 1, NULL);

  while(frame != frames)
  {
    cvShowImage("Output", outFrame);
    cvShowImage(fStr, filter);
     cvShowImage("Frequency area", freq);
    chW = cvWaitKey(_WAIT_TIME);

    if( chW == 27 || chC == 27)  //Esc
      break;
  }

  cvDestroyAllWindows();

  return 0;
}

int main(int argc, char** argv)
{
  const char* fileName =
     argc>=2 ? argv[1] : "Videos\\TestVideo1.avi";
  CvCapture* film = NULL;
  int w, h;
  int msec;
  LARGE_INTEGER start, finish, totStart;
  LARGE_INTEGER fr;
  double wrkTime = 0.0;
  CvSize size;
  HANDLE threadH;
  DWORD threadId;
```

```
  char codec[5];
  bool silent = false;
  FILE *f;
  int totFrames = 0;

  //1. Parse each string in "argv" array
  for(int i=2; i < argc; i++)
  {
    //Silent mode
    if( string(argv[i]) == "-s" )
    {
      silent = true;
      continue;
    }

    //Gauss lowpass filter
    if( string(argv[i]) == "-gl" )
    {
      fType = GaussLow;
      continue;
    }

    //Gauss highpass filter
    if( string(argv[i]) == "-gh" )
    {
      fType = GaussHigh;
      continue;
    }

    //Battervort lowpass filter
    if( string(argv[i]) == "-bl" )
    {
      fType = BattervortLow;
      continue;
    }

    //Battervort highpass filter
    if( string(argv[i]) == "-bh" )
    {
      fType = BattervortHigh;
      continue;
    }

    printf("%s\n", helpString);
    exit(1);
  }

  //2.1. Video file is present?
  printf("Video file: %s\n", fileName);
  f = fopen(fileName,"r");
```

```
if (f == NULL)
{
  printf("Error: File not found!\n");
  printf("%s\n", helpString);
  return -1;
}
fclose(f);

//2.2. Load video file
film=cvCreateFileCapture(fileName);
if(!film)
{
  printf("Error: Unsupported video codec!\n");
  printf("%s\n", helpString);
  return -1;
}

QueryPerformanceFrequency(&fr);

//3. Get video info
frames =
   cvGetCaptureProperty(film, CV_CAP_PROP_FRAME_COUNT);
int val =
   (int) cvGetCaptureProperty(film, CV_CAP_PROP_FOURCC);
memcpy(codec, &val, 4);
codec[4]=0;

w = (int) cvGetCaptureProperty(film,
                               CV_CAP_PROP_FRAME_WIDTH);
h = (int) cvGetCaptureProperty(film,
                               CV_CAP_PROP_FRAME_HEIGHT);
size.height = h;
size.width = w;

//4.1. Check up height and width for correct value
int bits = 0;
for( int tmp = h; tmp > 1; tmp/=2, bits++);
if( h != 1<<bits )
{
  printf("Error: Height should be power of 2!\n");
  cvReleaseCapture(&film);
  return -1;
}

bits = 0;
for( int tmp = w; tmp > 1; tmp/=2, bits++);
if( w != 1<<bits )
{
  printf("Error: Width should be power of 2!\n");
```

```
    cvReleaseCapture(&film);
    return -1;
  }

  //4.2. Create mutex for thread init
  threadInitMutex = CreateSemaphore(NULL, 0, 1, NULL);
  if(threadInitMutex==NULL)
  {
    printf("Error:      CreateMutex      error:      %d\n",
GetLastError());
    return -1;
  }

  //5. Print info
  printf("Loaded!\n");
  printf("Codec: \t%s\n", codec);
  printf("Total frames: %i\n", frames);
  printf("Picture size: %i x %i\n", w, h);
  printf("Filter: %s\n", filters[fType]);
  printf("\n");
  printf("Click on window to \"Pause\"\n");
  printf("Press ESC to \"Exit\"\n");
  printf("\n");

  //6. Create images
  empty = cvCreateImage(size, IPL_DEPTH_8U, 1);
  outFrame = cvCreateImage(size, IPL_DEPTH_8U, 3);
  filter = cvCreateImage(size, IPL_DEPTH_8U, 1);
  freq = cvCreateImage(size, IPL_DEPTH_8U, 3);

  //7. Create and wait thread for drawing
  if (!silent)
  {
    threadH = CreateThread(NULL, 0, DrawingFunction, NULL,
                           0, &threadId);
    WaitForSingleObject(threadInitMutex, INFINITE);
  }

  printf("Log:\n");

  //8. Variable of start time is installed
  QueryPerformanceCounter(&totStart);

  //9. Process frames
  while(frame != frames)
  {
    //9.1. Get next frame
    cvSetCaptureProperty( film, CV_CAP_PROP_POS_FRAMES,
                          frame);
    msec = (int)cvGetCaptureProperty(film,
```

```
                                        CV_CAP_PROP_POS_MSEC);
    inpFrame = cvQueryFrame(film);

    //9.2. Process current frames
    QueryPerformanceCounter(&start);
    ProcessFrame(inpFrame, outFrame, filter, freq,
                 fType, &r);
    QueryPerformanceCounter(&finish);

    //9.3. Increase count of total frames
    totFrames++;

    //9.4. Compute and print time
    printf("Frame: %5i, position: %6i ms, processing time:
           %f \n", frame, msec,
           (finish.QuadPart-start.QuadPart) /
           (double)fr.QuadPart);
    wrkTime += (finish.QuadPart-start.QuadPart) /
               (double)fr.QuadPart;

    //9.5. "ESC" is pressed?
    if(_kbhit())
      chC = _getch();

    //9.6. The thread is completed?
    if( WaitForSingleObject(threadH, 1) == WAIT_OBJECT_0)
      break;

     //9.7. Break if ESC pressed
    if (silent)
      if( chC == 27 )  //Esc
        break;

    //9.8. Compute next frame and move trackbar
    if(!pause)
    {
      frame++;
      cvSetTrackbarPos( "Frame", "Output", frame);
    }
  }

  QueryPerformanceCounter(&finish);

  //10. Print summary
  printf("Done!\n");
  printf("\n");
  printf("Total time: %f\n",
         (finish.QuadPart-totStart.QuadPart) /
         (double)fr.QuadPart);
```

```
  printf("Total frames: %i\n", totFrames);
  printf("Total processing time: %f\n", wrkTime);
  if(totFrames)
    printf("Average processing time per frame: %f\n",
           wrkTime/totFrames);

  //11. Release resources
  if (!silent)
    CloseHandle(threadH);

  cvReleaseImage(&freq);
  cvReleaseImage(&filter);
  cvReleaseImage(&outFrame);
  cvReleaseImage(&empty);

  CloseHandle(threadInitMutex);

  cvReleaseCapture(&film);

  return 0;
}
```

21.3. Финальная версия программы

fft.h

```
#ifndef _FFT_H
#define _FFT_H

#include <complex>
#include <vector>

using namespace std;

#define PI (3.14159265358979323846)

void ParallelFFT2D(double *inputSignal,
                   double *outputSignal,
                   int w, int h);

void ParallelInverseFFT2D(double *inputSignal,
                          double *outputSignal,
                          int w, int h);

#endif
```

fft.cpp

```
#include "fft.h"

#include "intrin.h"
```

```
unsigned char rev[256]={
0, 128, 64, 192, 32, 160, 96, 224, 16, 144, 80, 208, 48,
176, 112, 240, 8, 136, 72, 200, 40, 168, 104, 232, 24, 152,
88, 216, 56, 184, 120, 248, 4, 132, 68, 196, 36, 164, 100,
228, 20, 148, 84, 212, 52, 180, 116, 244, 12, 140, 76, 204,
44, 172, 108, 236, 28, 156, 92, 220, 60, 188, 124, 252, 2,
130, 66, 194, 34, 162, 98, 226, 18, 146, 82, 210, 50, 178,
114, 242, 10, 138, 74, 202, 42, 170, 106, 234, 26, 154, 90,
218, 58, 186, 122, 250, 6, 134, 70, 198, 38, 166, 102, 230,
22, 150, 86, 214, 54, 182, 118, 246, 14, 142, 78, 206, 46,
174, 110, 238, 30, 158, 94, 222, 62, 190, 126, 254, 1, 129,
65, 193, 33, 161, 97, 225, 17, 145, 81, 209, 49, 177, 113,
241, 9, 137, 73, 201, 41, 169, 105, 233, 25, 153, 89, 217,
57, 185, 121, 249, 5, 133, 69, 197, 37, 165, 101, 229, 21,
149, 85, 213, 53, 181, 117, 245, 13, 141, 77, 205, 45, 173,
109, 237, 29, 157, 93, 221, 61, 189, 125, 253, 3, 131, 67,
195, 35, 163, 99, 227, 19, 147, 83, 211, 51, 179, 115, 243,
11, 139, 75, 203, 43, 171, 107, 235, 27, 155, 91, 219, 59,
187, 123, 251, 7, 135, 71, 199, 39, 167, 103, 231, 23, 151,
87, 215, 55, 183, 119, 247, 15, 143, 79, 207, 47, 175, 111,
239, 31, 159, 95, 223, 63, 191, 127, 255};

union IntUni
{
  unsigned int integer;
  unsigned char byte[4];
};

void BitReversing(double *inputSignal,
                  double *outputSignal, int size)
{
  int m=0;
  for(int  tmp size  =  size;  tmp size>1;  tmp size/=2,  m++
);//size = 2^m

  IntUni i, j;
  j.integer=0;

  for(i.integer=0; i.integer<size/2; i.integer++)
  {
    j.byte[0]=rev[i.byte[3]];
    j.byte[1]=rev[i.byte[2]];
    j.byte[2]=rev[i.byte[1]];
    j.byte[3]=rev[i.byte[0]];

    j.integer = j.integer >> (32-m);

    outputSignal[2*i.integer] = inputSignal[2*j.integer];
```

```
    outputSignal[2*i.integer+1]                                =
inputSignal[2*j.integer+1];
    outputSignal[2*i.integer+size]                             =
inputSignal[2*j.integer+2];
    outputSignal[2*i.integer+1+size]                           =
inputSignal[2*j.integer+1+2];
  }
}

void SerialFFTCalculation(double *mas, int size)
{
  int m=0;
  for(int tmp_size = size; tmp_size>1; tmp_size/=2,
      m++ );//size = 2^m

  for (int p=0; p<m; p++ )
  {
    int butterflyOffset = 1 << (p+1) ;
    int butterflySize = butterflyOffset >> 1 ;

    double alpha=-2.0*PI/butterflyOffset;
    double wR=cos( alpha  ), wI=sin( alpha );

    for (int i=0; i<size/butterflyOffset; i++ )
    {
      double uR=1.0, uI=0.0;
      double uRtmp;

      int offset = i * butterflyOffset;

      for (int  j=0; j<butterflySize; j++ )
      {
        double temR =
           mas[2 * (j + offset +  butterflySize)] * uR -
           mas[2 * (j + offset +  butterflySize) + 1] * uI,
               temI =
           mas[2 * (j + offset +  butterflySize)] * uI +
           mas[2 * (j + offset +  butterflySize) + 1] * uR;

        mas[2 * (j + offset +  butterflySize)] =
           mas[2 * (j + offset)] - temR;
        mas[2 * (j + offset +  butterflySize) + 1] =
           mas[2 * (j + offset) + 1] - temI;

        mas[2 * (j + offset)] += temR;
        mas[2 * (j + offset) + 1] += temI;

        uRtmp=uR;
        uR=uR*wR-uI*wI;
        uI=uI*wR+uRtmp*wI;
```

```
      }
    }
  }
}

void SerialFFT1D(double *inputSignal,
                 double *outputSignal, int size)
{
  BitReversing(inputSignal, outputSignal, size);
  SerialFFTCalculation(outputSignal, size);
}

void ParallelFFT2D(double *inputSignal,
                   double *outputSignal, int w, int h)
{
  #pragma omp parallel
  {
    double *inW, *outW, *inH, *outH;

    inW = new double[2*w];
    outW = new double[2*w];
    inH = new double[2*h];
    outH = new double[2*h];


    #pragma omp for
    for(int i=0; i<w; i++)
    {
      for(int j=0; j<h; j++)
      {
        inH[2*j] = inputSignal[2*(j*w+i)];
        inH[2*j+1] = inputSignal[2*(j*w+i)+1];
      }

      SerialFFT1D(inH, outH, h);

      for(int j=0; j<h; j++)
      {
        outputSignal[2*(j*w+i)] = outH[2*j];
        outputSignal[2*(j*w+i)+1] = outH[2*j+1];
      }
    }

    #pragma omp for
    for(int j=0; j<h; j++)
    {
      for(int i=0; i<w; i++)
      {
        inW[2*i] = outputSignal[2*(j*w+i)];
```

```
        inW[2*i+1] = outputSignal[2*(j*w+i)+1];
      }

      SerialFFT1D(inW, outW, w);

      for(int i=0; i<w; i++)
      {
        outputSignal[2*(j*w+i)] = outW[2*i];
        outputSignal[2*(j*w+i)+1] = outW[2*i+1];
      }
    }

    delete[] inW;
    delete[] inH;
    delete[] outW;
    delete[] outH;
  }
}

void SerialInverseFFTCalculation(double *mas, int size)
{
  int m=0;
  for(int tmp_size = size; tmp_size>1; tmp_size/=2,
      m++ );//size = 2^m

  for (int p=0; p<m; p++ )
  {
    int butterflyOffset = 1 << (p+1) ;
    int butterflySize = butterflyOffset >> 1 ;

    double alpha=2.0*PI/butterflyOffset;
    double wR=cos( alpha  ), wI=sin( alpha );

    for (int i=0; i<size/butterflyOffset; i++ )
    {
      double uR=1.0, uI=0.0;
      double uRtmp;

         int offset = i * butterflyOffset;

      for (int  j=0; j<butterflySize; j++ )
      {
        double temR =
           mas[2 * (j + offset +  butterflySize)] * uR -
           mas[2 * (j + offset +  butterflySize) + 1] * uI,
               temI =
           mas[2 * (j + offset +  butterflySize)] * uI +
           mas[2 * (j + offset +  butterflySize) + 1] * uR;

        mas[2 * (j + offset +  butterflySize)] =
```

```
            mas[2 * (j + offset)] - temR;
        mas[2 * (j + offset +  butterflySize) + 1] =
            mas[2 * (j + offset) + 1] - temI;

        mas[2 * (j + offset)] += temR;
        mas[2 * (j + offset) + 1] += temI;

        uRtmp=uR;
        uR=uR*wR-uI*wI;
        uI=uI*wR+uRtmp*wI;
      }
    }
  }
}

void SerialInverseFFT1D(double *inputSignal,
                        double *outputSignal,
                        int size)
{
  BitReversing(inputSignal, outputSignal, size);
  SerialInverseFFTCalculation(outputSignal, size);

  double _size(size);

  for (int  j=0; j<2*size; j++ )
    outputSignal[j]/=_size;
}

void ParallelInverseFFT2D(double *inputSignal,
                          double *outputSignal,
                          int w, int h)
{
  #pragma omp parallel
  {
    double *inW, *outW, *inH, *outH;

    inW = new double[2*w];
    outW = new double[2*w];
    inH = new double[2*h];
    outH = new double[2*h];

    #pragma omp for
    for(int i=0; i<w; i++)
    {
      for(int j=0; j<h; j++)
      {
        inH[2*j] = inputSignal[2*(j*w+i)];
        inH[2*j+1] = inputSignal[2*(j*w+i)+1];
      }
```

```
      SerialInverseFFT1D(inH, outH, h);

      for(int j=0; j<h; j++)
      {
        outputSignal[2*(j*w+i)] = outH[2*j];
        outputSignal[2*(j*w+i)+1] = outH[2*j+1];
      }
    }

    #pragma omp for
    for(int j=0; j<h; j++)
    {
      for(int i=0; i<w; i++)
      {
        inW[2*i] = outputSignal[2*(j*w+i)];
        inW[2*i+1] = outputSignal[2*(j*w+i)+1];
      }

      SerialInverseFFT1D(inW, outW, w);

      for(int i=0; i<w; i++)
      {
        outputSignal[2*(j*w+i)] = outW[2*i];
        outputSignal[2*(j*w+i)+1] = outW[2*i+1];
      }
    }

    delete[] inW;
    delete[] inH;
    delete[] outW;
    delete[] outH;
  }
}
```

filter.cpp

```
//OpenCV
#include "cxcore.h"
#include "highgui.h"

//IO and time
#include "stdio.h"
#include "windows.h"
#include <conio.h>

#include <string>

#include "fft.h"

using namespace std;
```

```
#pragma comment(lib,"cxcore.lib")
#pragma comment(lib,"highgui.lib")

#define _WAIT_TIME 30

char helpString[] =
"\nUsage: \n\
\tfilter.exe <input video file> [options]\n\n\
Options:\n\
\t-s \
\t silent mode.\n\
\
\t-gl (default) \
\t Gauss lowpass filter.\n\
\
\t-gh \
\t Gauss highpass filter.\n\
\
\t-bl \
\t Battervort lowpass filter.\n\
\
\t-bh \
\t Battervort highpass filter.\n\
\n\
Example:\n\
\tfilter.exe video.avi -s -gh\
";

enum FilterType {GaussLow=0,
                 GaussHigh=1,
                 BattervortLow=2,
                 BattervortHigh=3};
char filters[4][50] = {"Gauss lowpass filter",
                       "Gauss highpass filter",
                       "Battervort lowpass filter",
                       "Battervort highpass filter"};
IplImage *inpFrame = NULL, *outFrame = NULL,
         *filter = NULL, *freq = NULL, *empty = NULL;
int chW = 0, chC = 0;
int frame=0, frames;
int r = 30;
bool pause = false;
HANDLE threadInitMutex;
FilterType fType = GaussLow;

double *f;
double *inpR, *inpG, *inpB,
       *outR, *outG, *outB;
```

```
void ProcessFrame(IplImage *frame, IplImage *outFrame,
                  IplImage* filter, IplImage* freq,
                  FilterType t, int *_r)
{
  int w = frame->width,
      h = frame->height,
      r = *_r,
      size = w * h;

  int a;

  double maxR = 0.0 , maxG = 0.0, maxB = 0.0;
  double minR = 0.0 , minG = 0.0, minB = 0.0;

  //Fill inp<R|G|B> arrays
  #pragma omp parallel
  {
    #pragma omp for private(a) nowait
    for (int i=0; i<h; i++)
    {
      a = pow(-1.0, i);
      int b = a;
      for (int j=0; j<w; j++)
      {
        inpB[2*(i*w + j)]=b *
       ((unsigned char*)frame->imageData)[(i*w + j)*3 + 0];

        inpB[2*(i*w + j) + 1]=0.0;
        b *= -1;
      }
      a *= -1;
    }

    #pragma omp for private(a) nowait
    for (int i=0; i<h; i++)
    {
      a = pow(-1.0, i);
      int b = a;
      for (int j=0; j<w; j++)
      {
        inpG[2*(i*w + j)]=b *
       ((unsigned char*)frame->imageData)[(i*w + j)*3 + 1];

        inpG[2*(i*w + j) + 1]=0.0;
        b *= -1;
      }
      a *= -1;
    }
```

```
    #pragma omp for private(a) nowait
    for (int i=0; i<h; i++)
    {
      a = pow(-1.0, i);
      int b = a;
      for (int j=0; j<w; j++)
      {
        inpR[2*(i*w + j)]=b *
       ((unsigned char*)frame->imageData)[(i*w + j)*3 + 2];

        inpR[2*(i*w + j) + 1]=0.0;
        b *= -1;
      }
      a *= -1;
    }
  }

  //Make filter
  switch(t)
  {
    case GaussLow:
      #pragma omp for
      for (int i=0; i<w; i++)
        for (int j=0; j<h; j++)
          f[i+j*w]=
             exp( -( (i-w/2)*(i-w/2) + (j-h/2)*(j-h/2) ) /
             (2.0*r*r ) );
      break;
    case GaussHigh:
      #pragma omp for
      for (int i=0; i<w; i++)
        for (int j=0; j<h; j++)
          f[i+j*w]= 1 -
             exp( -( (i-w/2)*(i-w/2) + (j-h/2)*(j-h/2) ) /
             (2.0*r*r ) );
      break;
    case BattervortLow:
     #pragma omp for
      for (int i=0; i<w; i++)
        for (int j=0; j<h; j++)
          f[i+j*w]=1 /
             ( 1 + ( (i-w/2)*(i-w/2) + (j-h/2)*(j-h/2) ) /
               (1.0*r*r) );
      break;
    case BattervortHigh:
      #pragma omp for
      for (int i=0; i<w; i++)
        for (int j=0; j<h; j++)
          f[i+j*w]=1 /
```

```
            ( 1 + (1.0*r*r) /
              ( (i-w/2)*(i-w/2) + (j-h/2)*(j-h/2) ) );
    break;
};


//Convert to frequency area
ParallelFFT2D(inpR, outR, w, h);
ParallelFFT2D(inpG, outG, w, h);
ParallelFFT2D(inpB, outB, w, h);

//Show filter
#pragma omp parallel
{
  #pragma omp for nowait
  for(int i=0; i<size; i++)
    filter->imageData[i] = (char) (255 * f[i]);

  //Apply filter
  #pragma omp for nowait
  for (int i=0; i<size; i++)
  {
    outR[2*i] = outR[2*i] * f[i];
    outR[2*i+1] = outR[2*i+1] * f[i];
  }

  #pragma omp for nowait
  for (int i=0; i<size; i++)
  {
    outG[2*i] = outG[2*i] * f[i];
    outG[2*i+1] = outG[2*i+1] * f[i];
  }

  #pragma omp for nowait
  for (int i=0; i<size; i++)
  {
    outB[2*i] = outB[2*i] * f[i];
    outB[2*i+1] = outB[2*i+1] * f[i];
  }
}


//Find max value
for (int i=0; i<size; i++)
{
  double t =
    sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]);

  if (t != 0)
  {
```

```
      maxR = minR = log(t);
      break;
    }
  }

  #pragma omp parallel
  {
    double max = maxR;
    double min = minR;

    #pragma omp for nowait
    for (int i=0; i<size; i++)
    {
      double t =
         sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]);
      double p = log(t);

      if (t != 0)
      {
        if(max < p )
          max = p;

        if(min > p )
          min = p;
      }
    }

    #pragma omp critical
    {
      if(max > maxR)
        maxR = max;

      if(min < minR)
        minR = min;
    }
  }

  for (int i=0; i<size; i++)
  {
    double t =
       sqrt(outG[2*i]*outG[2*i]+outG[2*i+1]*outG[2*i+1]);

    if (t != 0)
    {
      maxG = minG = log(t);
      break;
    }
  }
```

```
#pragma omp parallel
{
  double max = maxG;
  double min = minG;

  #pragma omp for nowait
  for (int i=0; i<size; i++)
  {
    double t =
       sqrt(outG[2*i]*outG[2*i]+outG[2*i+1]*outG[2*i+1]);
    double p = log(t);

    if (t != 0)
    {
      if(max < p )
        max = p;

      if(min > p )
        min = p;
    }
  }

  #pragma omp critical
  {
    if(max > maxG)
      maxG = max;

    if(min < minG)
      minG = min;
  }
}

for (int i=0; i<size; i++)
{
  double t =
     sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1]);

  if (t != 0)
  {
    maxB = minB = log(t);
    break;
  }
}

#pragma omp parallel
{
  double max = maxB;
  double min = minB;

  #pragma omp for nowait
```

```
    for (int i=0; i<size; i++)
    {
      double t =
         sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1]);
      double p = log(t);

      if (t != 0)
      {
        if(max < p )
          max = p;

        if(min > p )
          min = p;
      }
    }

    #pragma omp critical
    {
      if(max > maxB)
        maxB = max;

      if(min < minB)
        minB = min;
    }
  }

  //Show frequency area with filter
  #pragma omp parallel
  {
    #pragma omp for nowait
    for (int i=0; i<size; i++)
    {
      double t =
         sqrt(outR[2*i]*outR[2*i]+outR[2*i+1]*outR[2*i+1]);
      double delR = maxR - minR;

      if (t != 0)
        freq->imageData[3*i+0] = (char) (255 * ( log(t) -
minR ) / delR );
      else
        freq->imageData[3*i+0] = 0;
    }

    #pragma omp for nowait
    for (int i=0; i<size; i++)
    {
      double t =
         sqrt(outG[2*i]*outG[2*i]+outG[2*i+1]*outG[2*i+1]);
      double delG = maxG - minG;
```

```
      if (t != 0)
        freq->imageData[3*i+1] =
           (char) (255 * ( log(t) - minG ) / delG ) ;
      else
        freq->imageData[3*i+1] = 0;
    }

    #pragma omp for nowait
    for (int i=0; i<size; i++)
    {
      double t =
         sqrt(outB[2*i]*outB[2*i]+outB[2*i+1]*outB[2*i+1]);
      double delB = maxB - minB;

      if (t != 0)
        freq->imageData[3*i+2] =
           (char) (255 * ( log(t) - minB ) / delB ) ;
      else
        freq->imageData[3*i+2] = 0;
    }
  }

  //Convert to frequency area
  ParallelInverseFFT2D(outR, inpR, w, h);
  ParallelInverseFFT2D(outG, inpG, w, h);
  ParallelInverseFFT2D(outB, inpB, w, h);

  #pragma omp parallel for private(a)
  for (int i=0; i<h; i++)
  {
    a = pow(-1.0, i);
    int b = a;
    for (int j=0; j<w; j++)
    {
      outFrame->imageData[3*(i*w + j)+0] =
         (char) (b * inpB[2*(i*w + j)]);
      outFrame->imageData[3*(i*w + j)+1] =
         (char) (b * inpG[2*(i*w + j)]);
      outFrame->imageData[3*(i*w + j)+2] =
         (char) (b * inpR[2*(i*w + j)]);

      b *= -1;
    }
    a *= -1;
  }

}

void OnWindowClick(int ev, int x, int y, int flags,
```

```
                    void* param)
{
  if(ev == CV_EVENT_LBUTTONDOWN)
  {
    if(*(bool*)param)
      printf("Resume!\n");
    else
      printf("Pause!!!\n");

    *(bool*)param = !(*(bool*)param);
  }
}

DWORD WINAPI DrawingFunction( LPVOID lpParam )
{
  cvNamedWindow("Output",CV_WINDOW_AUTOSIZE);
  cvShowImage("Output", empty);
  cvCreateTrackbar( "Frame", "Output", &frame, frames,
                    NULL );
  cvMoveWindow("Output", 50, 200);

  char fStr[100];
  sprintf(fStr, "Filter: %s", filters[fType]);
  cvNamedWindow(fStr,CV_WINDOW_AUTOSIZE);
  cvShowImage(fStr, empty);
  cvCreateTrackbar( "Radius", fStr, &r,
                    cvGetSize(empty).width, 0 );
  cvMoveWindow(fStr, 400, 200);

  cvNamedWindow("Frequency area",CV_WINDOW_AUTOSIZE);
  cvMoveWindow("Frequency area", 750, 200);

  cvSetMouseCallback("Output", OnWindowClick, &pause);
  cvSetMouseCallback(fStr, OnWindowClick, &pause);
  cvSetMouseCallback("Frequency area", OnWindowClick,
                     &pause);

  ReleaseSemaphore(threadInitMutex, 1, NULL);

  while(frame != frames)
  {
    cvShowImage("Output", outFrame);
    cvShowImage(fStr, filter);
     cvShowImage("Frequency area", freq);
    chW = cvWaitKey(_WAIT_TIME);

    if( chW == 27 || chC == 27)  //Esc
      break;
  }
```

```
  cvDestroyAllWindows();

  return 0;
}

int main(int argc, char** argv)
{
  const char* fileName =
     argc>=2 ? argv[1] : "Videos\\TestVideo1.avi";
  CvCapture* film = NULL;
  int w, h, s;
  int msec;
  LARGE_INTEGER start, finish, totStart;
  LARGE_INTEGER fr;
  double wrkTime = 0.0;
  CvSize size;
  HANDLE threadH;
  DWORD threadId;
  char codec[5];
  bool silent = false;
  FILE *file;
  int totFrames = 0;

  //1. Parse each string in "argv" array
  for(int i=2; i < argc; i++)
  {
    //Silent mode
    if( string(argv[i]) == "-s" )
    {
      silent = true;
      continue;
    }

    //Gauss lowpass filter
    if( string(argv[i]) == "-gl" )
    {
      fType = GaussLow;
      continue;
    }

    //Gauss highpass filter
    if( string(argv[i]) == "-gh" )
    {
      fType = GaussHigh;
      continue;
    }

    //Battervort lowpass filter
    if( string(argv[i]) == "-bl" )
```

```
    {
      fType = BattervortLow;
      continue;
    }

    //Battervort highpass filter
    if( string(argv[i]) == "-bh" )
    {
      fType = BattervortHigh;
      continue;
    }

    printf("%s\n", helpString);
    exit(1);
  }

  //2.1. Video file is present?
  printf("Video file: %s\n", fileName);
  file = fopen(fileName,"r");
  if (file == NULL)
  {
    printf("Error: File not found!\n");
    printf("%s\n", helpString);
    return -1;
  }
  fclose(file);

  //2.2. Load video file
  film=cvCreateFileCapture(fileName);
  if(!film)
  {
    printf("Error: Unsupported video codec!\n");
    printf("%s\n", helpString);
    return -1;
  }

  QueryPerformanceFrequency(&fr);

  //3. Get video info
  frames = cvGetCaptureProperty(film,
                                CV_CAP_PROP_FRAME_COUNT);
  int val = (int) cvGetCaptureProperty(film,
                                       CV_CAP_PROP_FOURCC);
  memcpy(codec, &val, 4);
  codec[4]=0;

  w = (int) cvGetCaptureProperty(film,
                                 CV_CAP_PROP_FRAME_WIDTH);
  h = (int) cvGetCaptureProperty(film,
```

```
                                        CV_CAP_PROP_FRAME_HEIGHT);
s = w * h;
size.height = h;
size.width = w;

//4.1. Check up height and width for correct value
int bits = 0;
for( int tmp = h; tmp > 1; tmp/=2, bits++);
if( h != 1<<bits )
{
  printf("Error: Height should be power of 2!\n");
  cvReleaseCapture(&film);
  return -1;
}

bits = 0;
for( int tmp = w; tmp > 1; tmp/=2, bits++);
if( w != 1<<bits )
{
  printf("Error: Width should be power of 2!\n");
  cvReleaseCapture(&film);
  return -1;
}

//4.2. Create mutex for thread init
threadInitMutex = CreateSemaphore(NULL, 0, 1, NULL);
if(threadInitMutex==NULL)
{
  printf("Error: CreateMutex error: %d\n",
         GetLastError());
  return -1;
}

//5. Print info
printf("Loaded!\n");
printf("Codec: \t%s\n", codec);
printf("Total frames: %i\n", frames);
printf("Picture size: %i x %i\n", w, h);
printf("Filter: %s\n", filters[fType]);
printf("\n");
printf("Click on window to \"Pause\"\n");
printf("Press ESC to \"Exit\"\n");
printf("\n");

//6.1. Create images
empty = cvCreateImage(size, IPL_DEPTH_8U, 1);
outFrame = cvCreateImage(size, IPL_DEPTH_8U, 3);
filter = cvCreateImage(size, IPL_DEPTH_8U, 1);
freq = cvCreateImage(size, IPL_DEPTH_8U, 3);
```

```
 //6.2. Create arrays
  f=new double[s];
  inpR = new double[2*s],
  inpG = new double[2*s],
  inpB = new double[2*s],
  outR = new double[2*s],
  outG = new double[2*s],
  outB = new double[2*s];

  //7. Create and wait thread for drawing
  if (!silent)
  {
    threadH = CreateThread(NULL, 0, DrawingFunction, NULL,
                           0, &threadId);
    WaitForSingleObject(threadInitMutex, INFINITE);
  }

  printf("Log:\n");

  //8. Variable of start time is installed
  QueryPerformanceCounter(&totStart);

  //9. Process frames
  while(frame != frames)
  {
    //9.1. Get next frame
    cvSetCaptureProperty(   film,    CV_CAP_PROP_POS_FRAMES,
frame);
    msec = (int)cvGetCaptureProperty(film,
                                     CV_CAP_PROP_POS_MSEC);
    inpFrame = cvQueryFrame(film);

    //9.2. Process current frames
    QueryPerformanceCounter(&start);
    ProcessFrame(inpFrame, outFrame, filter, freq, fType,
                 &r);
    QueryPerformanceCounter(&finish);

    //9.3. Increase count of total frames
    totFrames++;

    //9.4. Compute and print time
    printf("Frame: %5i, position: %6i ms, processing time:
           %f \n", frame, msec,
           (finish.QuadPart-start.QuadPart) /
           (double)fr.QuadPart);
    wrkTime += (finish.QuadPart-start.QuadPart) /
               (double)fr.QuadPart;
```

```
    //9.5. "ESC" is pressed?
    if(_kbhit())
      chC = _getch();

    //9.6. The thread is completed?
    if( WaitForSingleObject(threadH, 1) == WAIT_OBJECT_0)
      break;

     //9.7. Break if ESC pressed
    if (silent)
      if( chC == 27 )  //Esc
        break;

    //9.8. Compute next frame and move trackbar
    if(!pause)
    {
      frame++;
      cvSetTrackbarPos( "Frame", "Output", frame);
    }
  }

  QueryPerformanceCounter(&finish);

  //10. Print summary
  printf("Done!\n");
  printf("\n");
  printf("Total time: %f\n",
         (finish.QuadPart-totStart.QuadPart) /
         (double)fr.QuadPart);
  printf("Total frames: %i\n", totFrames);
  printf("Total processing time: %f\n", wrkTime);
  if(totFrames)
    printf("Average processing time per frame: %f\n",
           wrkTime/totFrames);

  //11. Release resources
  if (!silent)
    CloseHandle(threadH);

  delete[] inpB;
  delete[] inpG;
  delete[] inpR;
  delete[] outB;
  delete[] outG;
  delete[] outR;
  delete[] f;

  cvReleaseImage(&freq);
  cvReleaseImage(&filter);
  cvReleaseImage(&outFrame);
```

```
  cvReleaseImage(&empty);

  CloseHandle(threadInitMutex);

  cvReleaseCapture(&film);

  return 0;
}
```