Нижегородский государственный университет им. Н.И.Лобачевского

Зимняя школа по высокопроизводительным вычислениям ННГУ

при поддержке Интел

Мастер-класс по Intel Parallel Studio. Частотная фильтрация изображений (быстрое преобразование Фурье).

"Быстрое преобразование Фурье" - анализ, разработка, отладка, оптимизация, распараллеливание.

Сиднев А.А.
Кафедра математического обеспечения ЭВМ, факультет ВМК

Предварительный вариант

Преобразование Фурье. Назначение и примеры использования

Назначение ПФ

- В обработке сигналов и связанных областях преобразование Фурье обычно рассматривается как декомпозиция сигнала на частоты и амплитуды, то есть, *обратимый* переход от *временного пространства* (*time domain*) в *частотное пространство* (*frequency domain*).
- Далее рассмотрим примеры:
  - обработка изображений;
  - обработка звука.

Аудио

- В исходном WAV файле, хранится полная информация об исходном звуке, оцифрованном и проквантованном с частотой 44кГц.
- Этой информации абсолютно достаточно для воспроизведения всех частот исходного сигнала, меньших половины частоты квантования (по т. Котельникова).
- !!! Одна минута записи занимает около **15 МБ**.

Сжатие MP3

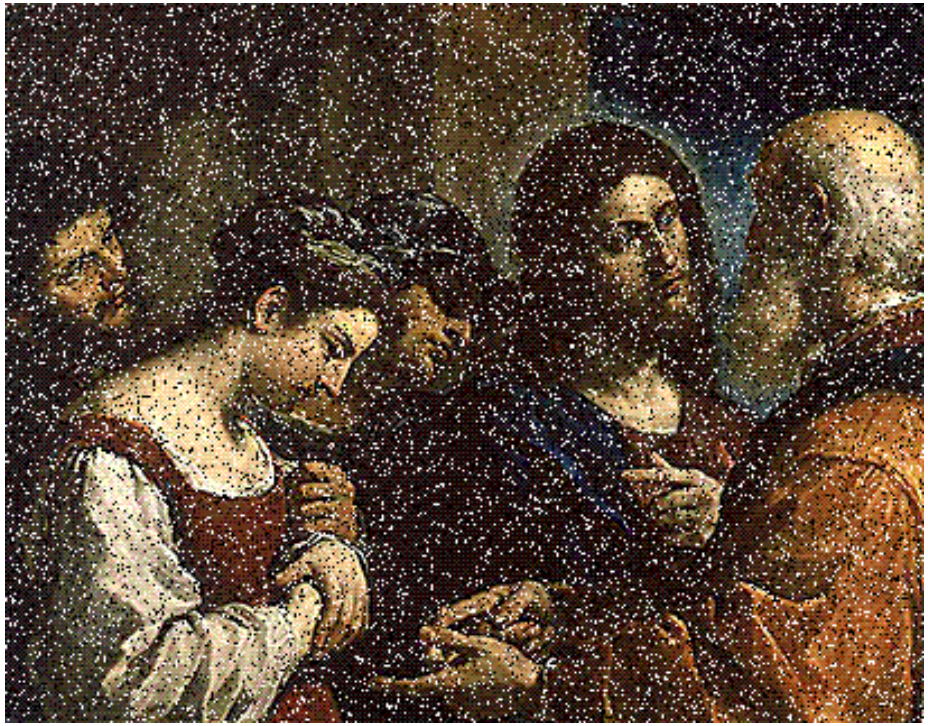
- MPEG Layer-3.
- Исходный звуковой файл режется на фрагменты, длительностью по 50 миллисекунд, каждый из которых анализируется отдельно.
- При анализе фрагмент раскладывается на гармоники по методу Фурье, из которых в соответствии с теорией восприятия звука человеческим ухом выбрасываются те гармоники, которые человек хуже воспринимает на фоне остальных:
  - более тихие гармоники на фоне более громких;
  - звуки, замаскированные вследствие инертности слуха.

Воспроизведение MP3

- При воспроизведении производится обратное преобразование, при котором оставшиеся гармоники вновь преобразуются в звуковую волну.
- Поскольку часть информации об исходном сигнале была выкинута, звук получается не совпадающий с исходным.
- Битрейт - это объем информации в единицу времени, как много информации можно потратить 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 выкидывать.

Фильтрация изображений

- Пример очистки зашумленного изображения медианным фильтром

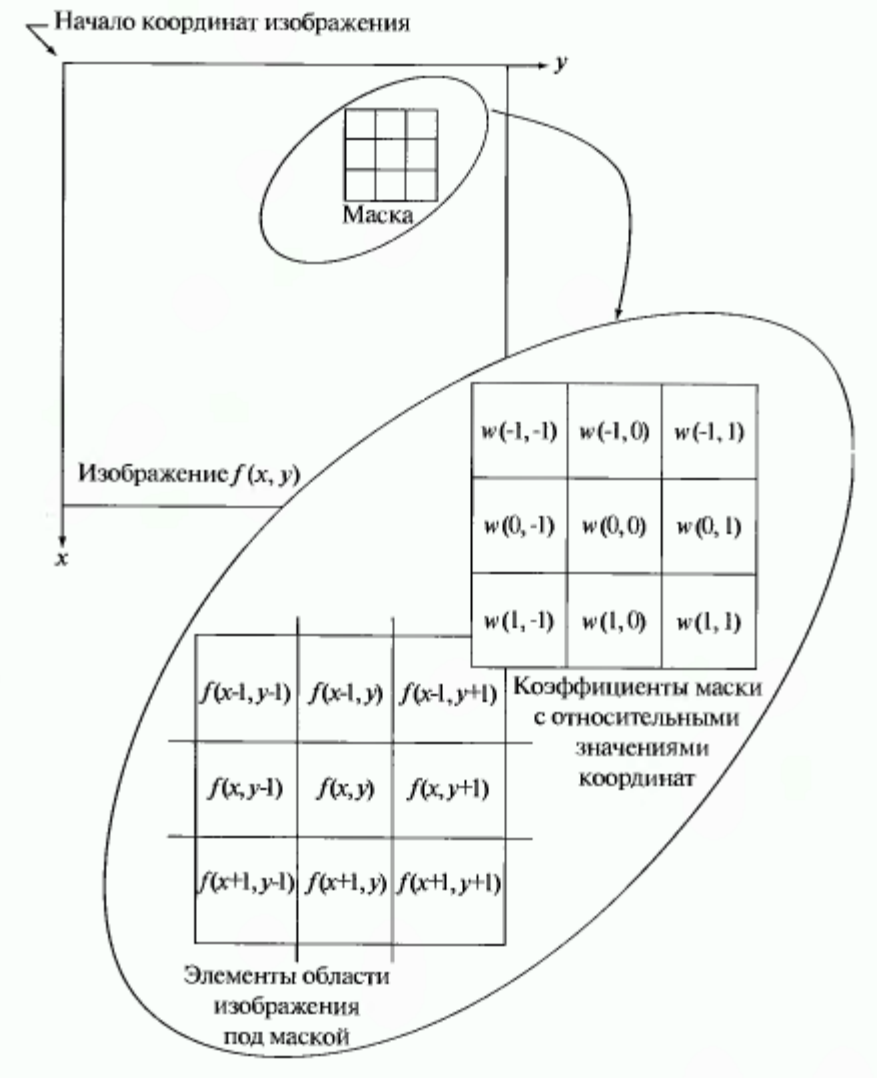


Цифровая обработка изображений

- Фильтрация
  - Пространственная
    - Линейный фильтр
    - Медианный фильтр
    - Изотропный фильтр
  - Частотная

Пространственная фильтрация

- Процесс фильтрации основан на простом перемещении маски фильтра от точки к точке изображения.
- В каждой точке отклик фильтра вычисляется с использованием предварительно заданных связей.
- Пространственные фильтры имеют **линейную** трудоёмкость.

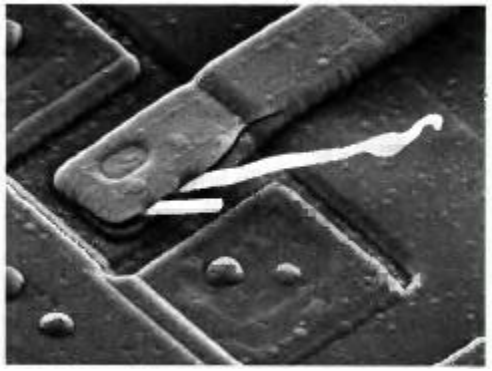


Частотная фильтрация

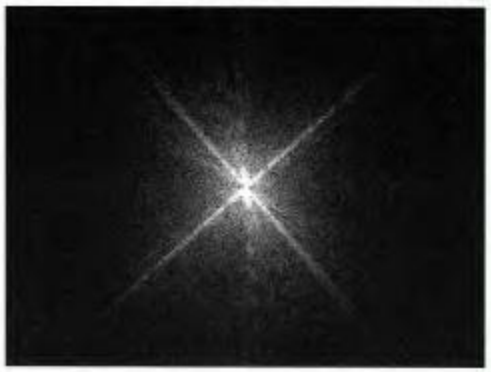
- Все пространственные фильтры можно реализовать с помощью частотных.
- Фильтрация происходит в частотной области изображения.
- С помощью преобразования Фурье происходит получение частотной области (спектра) изображения.
- Простая идея фильтрации:
  - низкие частоты – плавное изменение яркости изображения;
  - высокие частоты – быстрое изменение яркости изображения (границы объектов).

Пример частотного спектра

- Изображение повреждённой интегральной схемы.



- Фурье-спектр (яркое перекрестие отражает наличие дефекта в схеме).



0. Начнём с простого: кто знает язык C?

Разминка

Ошибки в программе

- **Откройте проект “.\0. filter (bugs)\filter.sln”.**
- **Найдите и исправьте ошибки!**
- **Заставьте программу работать!**
- Компилятор вам поможет ☺

Ошибки компиляции

- Отсутствует “using namespace std;” (filter.cpp:14, fft:h:6).
- Отсутствует “;” (filter.cpp:48).
- Отсутствует “}”, закрывающая тело цикла (filter.cpp:273).
- Опечатка в функции *p**n**intf* (filter.cpp:290).
- Неверная передача параметров в функцию *printf* (filter.cpp:339).
- Неверный аргумент функции *cvReleaseImage* - отсутствует “&” (filter.cpp:418).
- Не объявлена переменная, хранящая количество обработанных кадров (filter.cpp:231):

  int totFrames = 0;

1. Суслика видишь? Нет? А он есть!

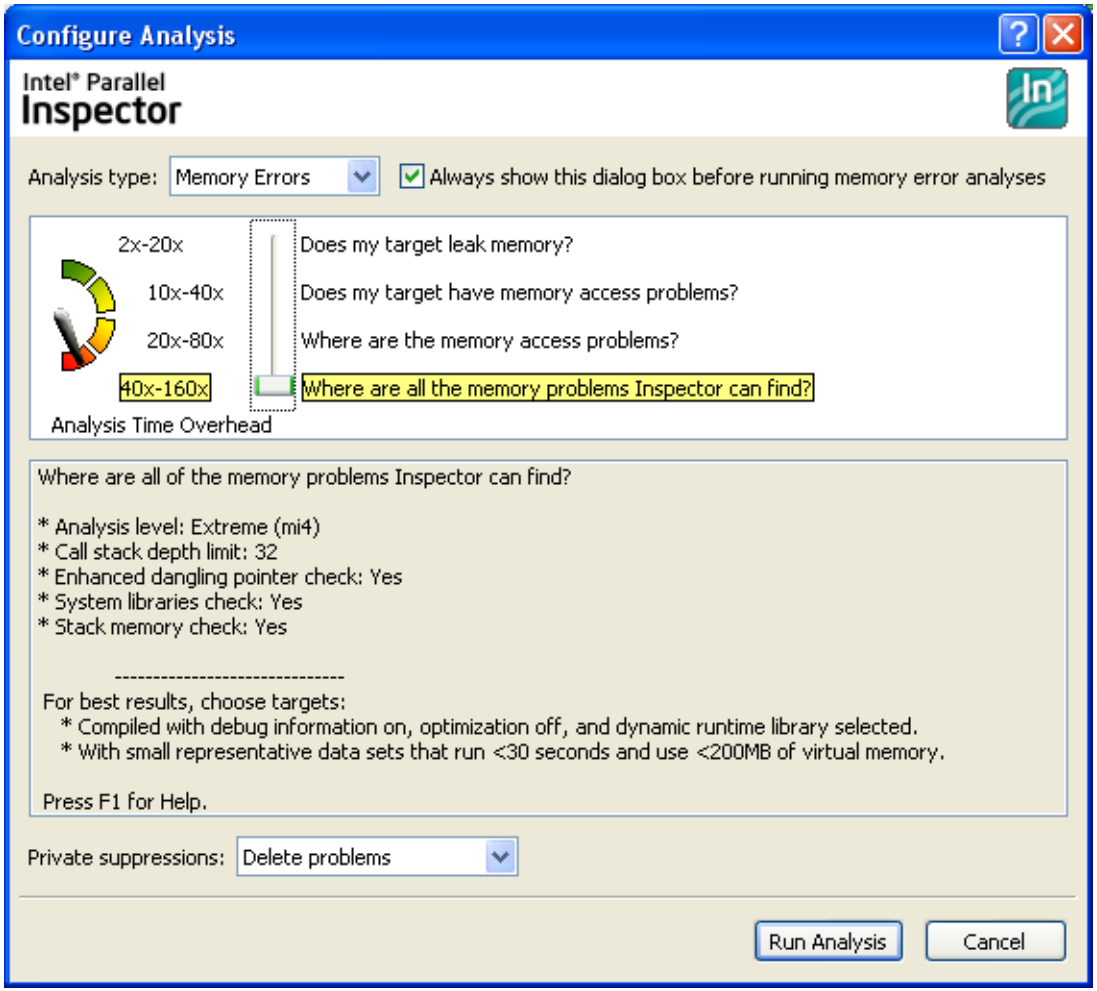
Ещё раз вспомним C

Ошибки в программе

- **Откройте проект “.\1. filter (errors)\filter.sln”.**
- Программа запустилась.
- Теперь необходимо заставить работать программу **правильно**!!!
- **Для поиска ошибок работы с памятью воспользуйтесь Intel Parallel Inspector (достаточно одного кадра):**
  - **“Debug” версия программы;**
  - **режим работы “Memory errors”;**
  - **режим анализа “Analysis level: Extreme”.**

Конфигурация Intel Parallel Inspector

- Пример конфигурации Intel Parallel Inspector для анализа:



Ошибки (1)

- Не освобождается ресурс “film” перед завершением работы программы (filter.cpp:425):

  cvReleaseCapture(&film);

- Не правильное вычитание (filter.cpp:382):

  (finish.QuadPart-start.QuadPart)

- Не выполняется передача параметра при рекурсивном вызове (fft.cpp:44):

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

- Не закрывается открытый файл (filter.cpp:285):

  fclose(f);

Ошибки (2)

- Не уничтожается одно из окон перед завершением работы программы (filter.cpp:215):

  cvDestroyWindow(fStr);

- Не освобождаются ресурсы картинки (filter.cpp:421): *cvReleaseImage(&filter);*

2. Заработало?! Тут случаем не первый Пень!? Ладно, главное, чтоб тачка не глючила.

Профилировка

Оценка эффективности

- **Откройте проект “.\2. filter (simple)\filter.sln”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

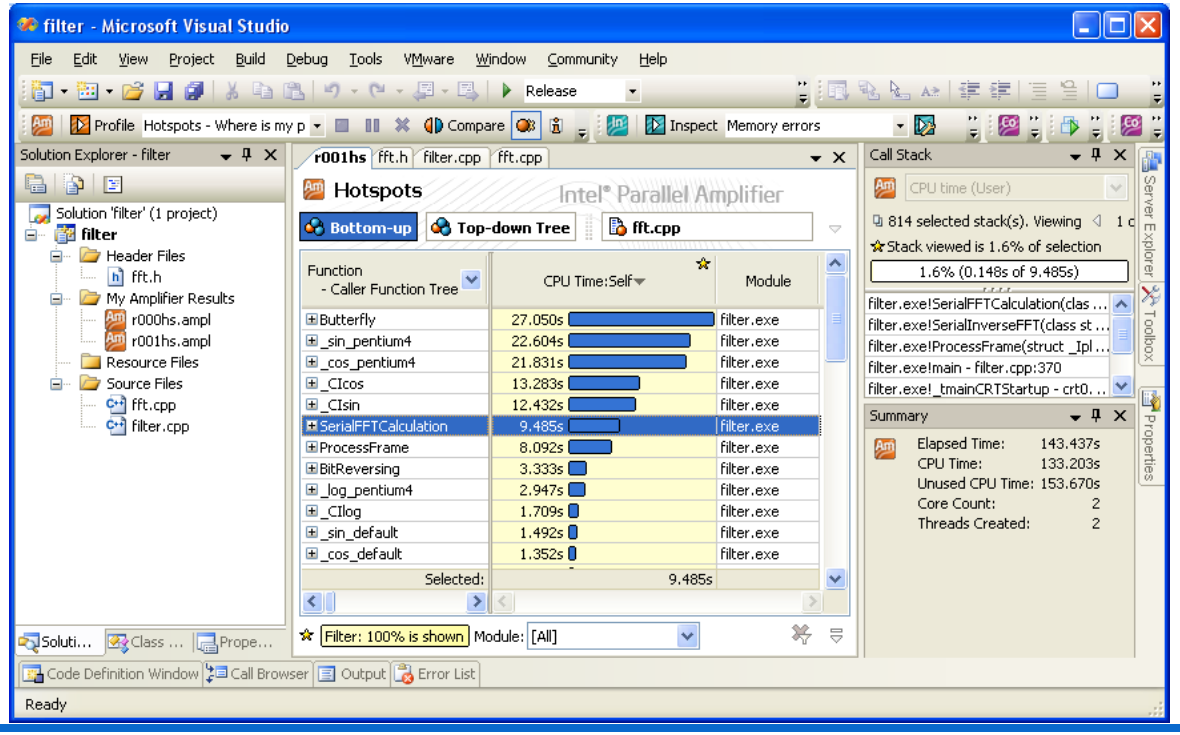
- Время, потраченное на обработку видео: 131.6 с

Профилировка

- Итак, программа работает очень медленно и нам предстоит с ней разобраться! Для этого необходимо определить наиболее “горячие” точки программы – локализовать задачу.
- Можно воспользоваться профилировщиком:
  - Intel VTune;
  - Intel Parallel Amplifier;
  - Profiler (VS 2008);
  - gprof.

Использование Intel Parallel Amplifier

- Для поиска горячих точек в программе воспользуйтесь Intel Parallel Amplifier:
  - “Release” версия программы;
  - режим работы “Hotspots”.



“Горячая” точка

- Самая “горячая” функция: *Butterfly*.

Butterfly	29.447s	filter.exe
_sin_pentium4	22.182s	filter.exe
_cos_pentium4	20.730s	filter.exe
_CIcos	12.731s	filter.exe
_CIsin	12.167s	filter.exe
SerialFFTCalculation	9.077s	filter.exe
ProcessFrame	5.318s	filter.exe
BitReversing	4.353s	filter.exe
_log_pentium4	2.844s	filter.exe
_CIlog	1.703s	filter.exe

- Если получится выполнить оптимизацию этой функции, то получим максимальный выигрыш!

Использование массивов вместо std:vector

- Очень медленно выполняется операция =.

```
29  void Butterfly(vector<complex<double>> &signal, complex<double> &
30  {                                                          1.687s
31    complex<double> tem = signal[offset +  butterflySize] * u;  1.843s
32
33    signal[offset + butterflySize] = signal[offset] - tem;     19.965s
34    signal[offset] += tem;                                     2.422s
35  }                                                          3.530s
```

- Это связано с тем, что при выполнении “копирования” выполняется вызов оператора = класса *vector*. Это делается очень медленно.
- Помимо этого, контейнеры передаются в функции по значению, что приводит к многократным лишним копированиям.
- Откажемся от использования *std:vector* – будем использовать обычные массивы.

3. STL реализации различных контейнеров далеко не самые быстрые. Их достоинство в унификации, а не в скорости.

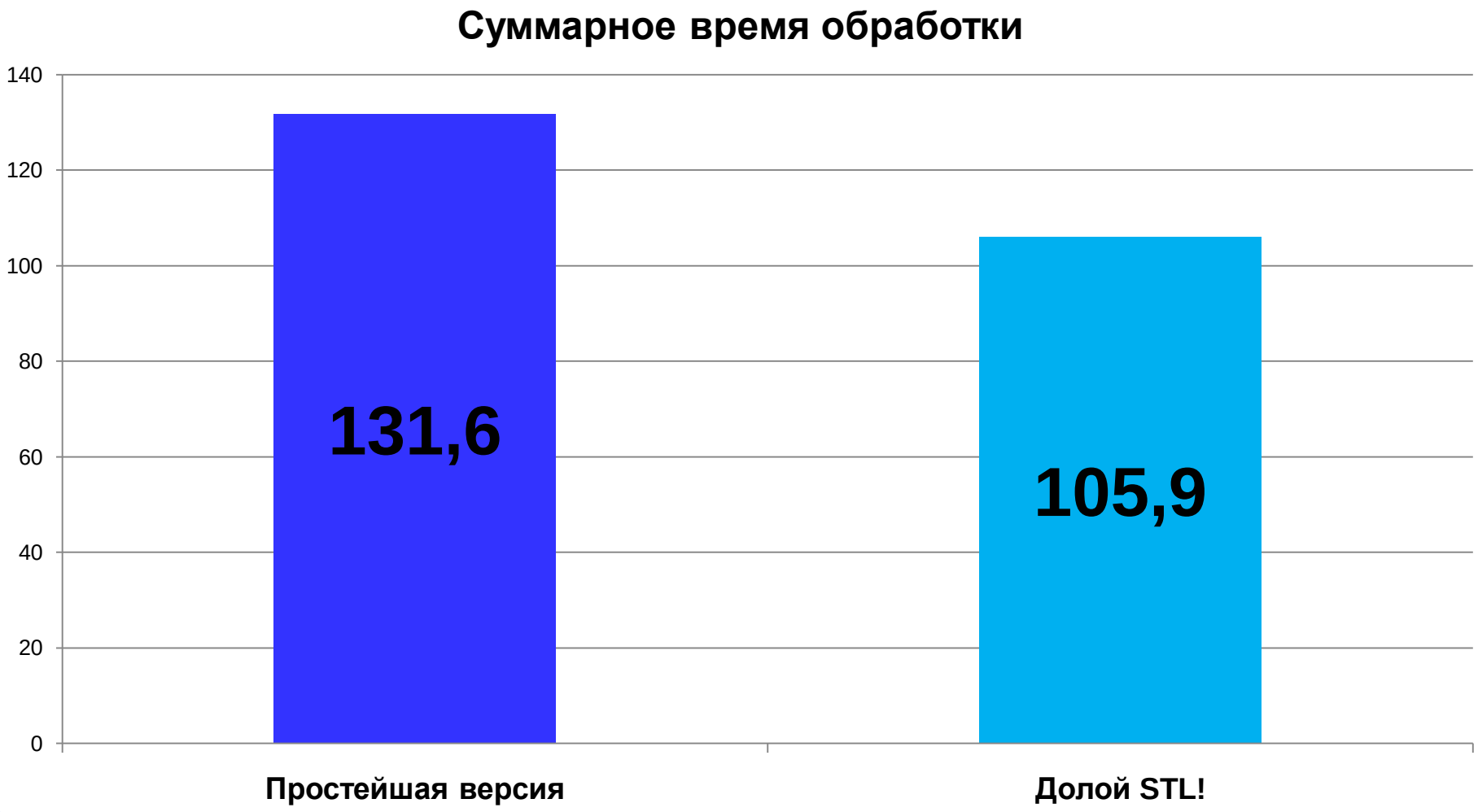
Долой STL! Долой!

Оценка эффективности

- **Откройте проект “.\3. filter (down with STL)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 105.9 с

Прогресс оптимизации



Ищем ошибки

- А программа то была с ошибками! ;-)
- **С помощью Intel Parallel Inspector найдите ошибки работы с памятью (достаточно одного кадра):**
  - **“Debug” версия программы;**
  - **режим работы “Memory errors”;**
  - **достаточно среднего режима анализа “Analysis level: Medium”.**

Ошибки

- Не освобождается память, выделенная под динамический массив (filter.cpp:171):

  delete[] outR;

- Освобождение памяти не соответствует выделению (filter.cpp:173):

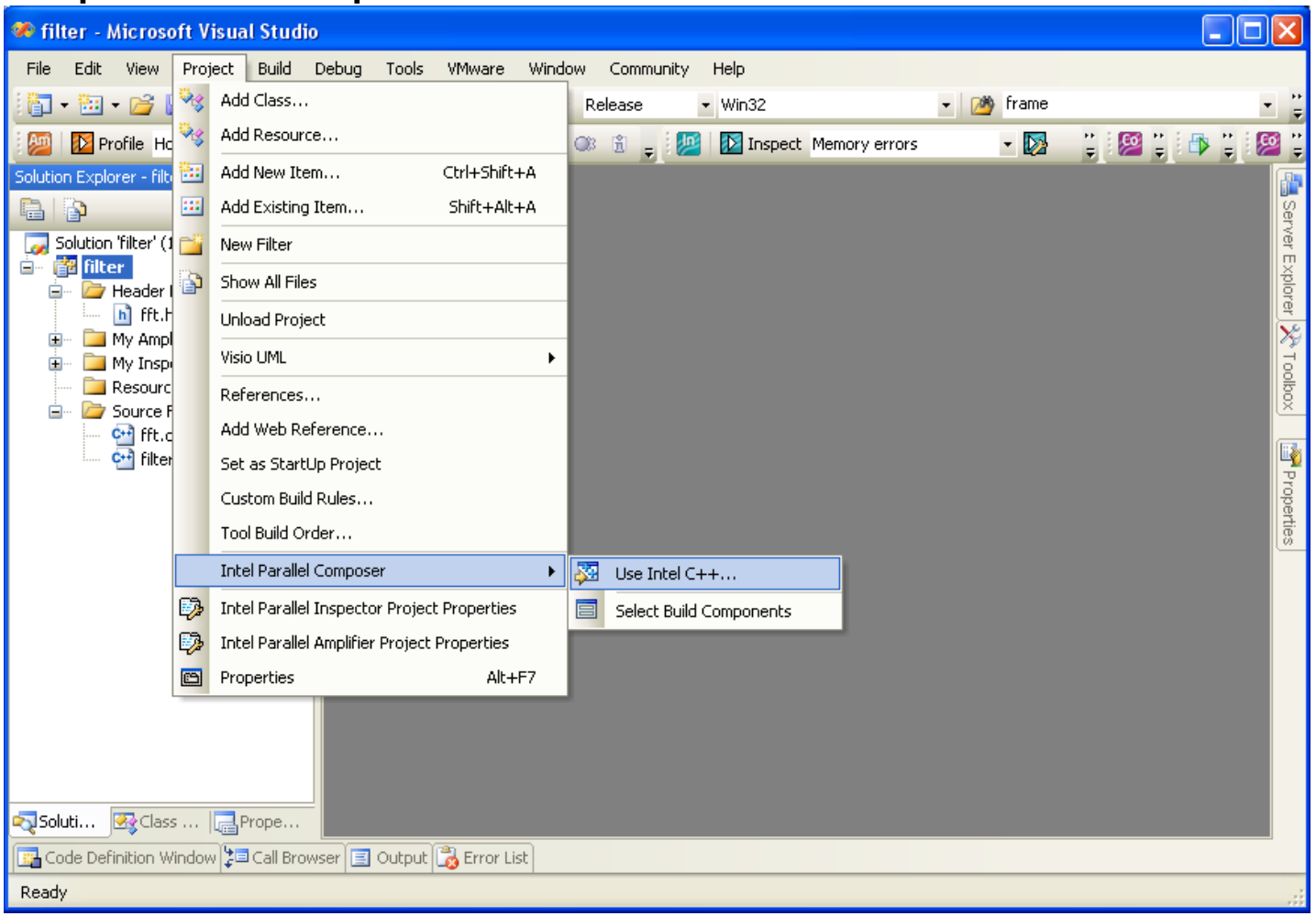
  delete[] f;

Intel Compiler

- Далее поступим самым простым образом (давно надо было так сделать ☺): воспользуемся оптимизирующим компилятором Intel.

Использование Intel Compiler

- Конвертирование проекта:



4. “Тяжесть — это хорошо. Тяжесть — это надёжно. Даже если не выстрелит, таким всегда можно врезать...” (о револьвере)

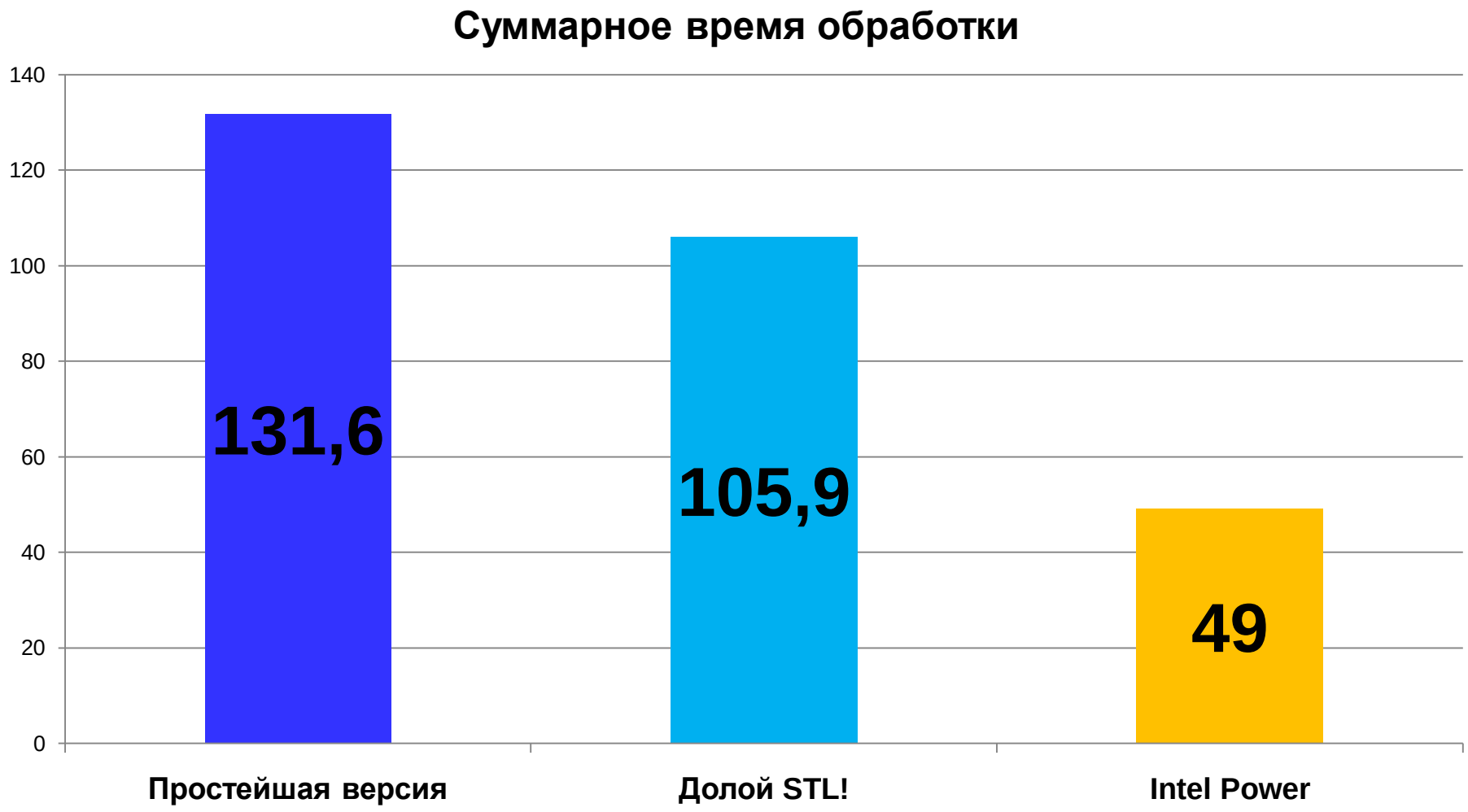
Intel Power

Оценка эффективности

- **Откройте проект “.\4. filter (Intel power)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 49 с

Прогресс оптимизации



Ищём следующее проблемное место

“– Что дальше?

– Снова профилировка и поиск “горячих” точек!”

- **Для поиска горячих точек в программе воспользуйтесь Intel Parallel Amplifier:**
  - **“Release” версия программы;**
  - **режим работы “Hotspots”.**

Самая “медленная” функция

- Большую часть времени работает функция SerialFFTCalculation:

SerialFFTCalculation	31.322s	filter.exe
ProcessFrame	15.510s	filter.exe
SerialInverseFFT	1.861s	filter.exe
SerialFFT	0.047s	filter.exe
KiFastSystemCallRet	0.031s	ntdll.dll
cvWaitKey	0.016s	highgui110.dll
LdrDisableThreadCalloutsForDll	0.012s	ntdll.dll
CvCaptureAVI_VFW::grabFrame	0.009s	highgui110.dll
cvSetTrackbarPos	0.001s	highgui110.dll
icvUpdateTrackbar	0.000s	highgui110.dll

“Горячая” точка

```
37  void SerialFFTCalculation(complex<double> *signal, int first, int size,
38  {                                                                  0.547s
39    if(size==1)                                                      0.078s
40      return;
41
42    double const coeff=2.0*PI/size;                                  0.313s
43
44    SerialFFTCalculation( signal, first, size/2, forward);           0.047s
45    SerialFFTCalculation( signal, first + size/2, size/2, forward);  0.031s
46
47    for (int  j=first; j<first+size/2; j++ )                         0.328s
48      if(forward)                                                    0.515s
49        Butterfly(signal, complex<double>(cos(-j*coeff), sin(-j*coeff)), j  11.187s
50      else
51        Butterfly(signal, complex<double>(cos(j*coeff), sin(j*coeff)), j  17.651s
52  }                                                                  0.625s
```

- Очень долго происходит вычисление функций sin/cos.

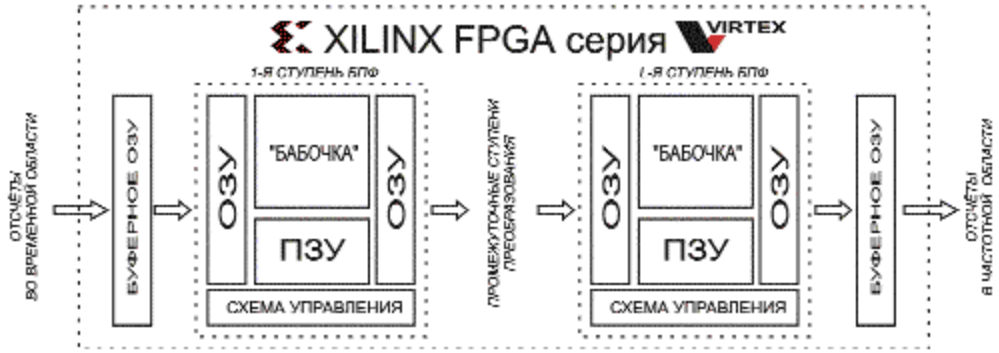
Преобразование Фурье

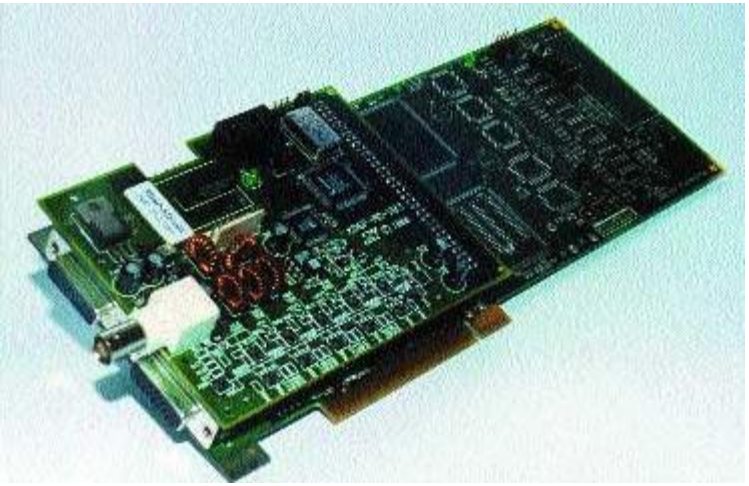
Преобразование Фурье…

- ПФ широко применяется в задачах анализа спектра сигнала, цифровой обработки сигналов и др.
- Существуют специализированные процессоры для вычисления ДПФ (DSP).

Аппаратная реализация БПФ

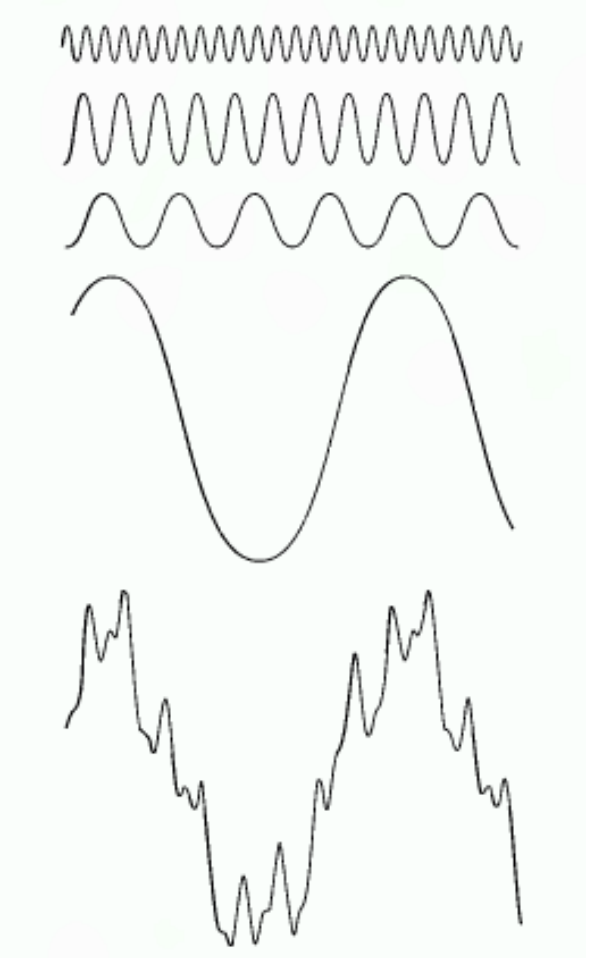
- **Однокристальная реализация алгоритма БПФ на ПЛИС фирмы Xilinx**





Преобразование Фурье

- Любая функция, периодически воспроизводящая свои значения, может быть представлена в виде суммы синусов или косинусов различных частот, умноженных на некоторые коэффициенты.



“Долгое” преобразование Фурье

- Дискретное преобразование Фурье (ДПФ, DFT) – это разложение дискретной функций на гармонические составляющие.
- Вычисление ДПФ чаще всего происходит в виде быстрого преобразования Фурье (БПФ, FFT).

Комплексные числа

Комплексное число

$z = x + iy; \quad x, y$ - вещественные числа; $\quad i^2 = -1$

Тригонометрическая форма записи

$z = r(\cos\varphi + i\sin\varphi); \quad r = |z|$

$$\cos\varphi = \frac{x}{|z|}, \sin\varphi = \frac{y}{|z|}$$

Показательная форма записи

$z = re^{i\varphi} = r(\cos\varphi + i\sin\varphi)$

Дискретное преобразование Фурье

$f(x)$ - комплексная функция действительного аргумента

$$x_k = f(k), k = \overline{0, n-1}$$

$$y_p = \sum_{k=0}^{n-1} x_k e^{-\frac{kp}{n} 2\pi i}, \; p = \overline{0, n-1}$$ - дискретное преобразование Фурье (ДПФ)

$$y_p = \sum_{k=0}^{n-1} x_k \left(\cos\left(\frac{kp}{n} 2\pi\right) - i \sin\left(\frac{kp}{n} 2\pi\right)\right), \; p = \overline{0, n-1}$$

Трудоёмкость:

Дискретное преобразование Фурье

$f(x)$ - комплексная функция действительного аргумента

$$x_k = f(k), k = \overline{0, n-1}$$

$$y_p = \sum_{k=0}^{n-1} x_k e^{-\frac{kp}{n} 2\pi i}, \; p = \overline{0, n-1}$$ - дискретное преобразование Фурье (ДПФ)

$$y_p = \sum_{k=0}^{n-1} x_k (\cos(\frac{kp}{n} 2\pi) - i \sin(\frac{kp}{n} 2\pi)), \; p = \overline{0, n-1}$$

Трудоёмкость: $O(n^2)$

Прямое и обратное преобразование Фурье

- По заданному фурье-образу можно восстановить исходную функцию с помощью обратного ДПФ

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

- прямое дискретное преобразование Фурье (ДПФ)

Трудоёмкость: $O(n^2)$

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

- обратное дискретное преобразование Фурье (ДПФ)

Трудоёмкость: $O(n^2)$

Быстрое преобразование Фурье…

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

Пусть n – чётное, тогда $y_p = \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{2kp}{n}2\pi i}}_{\text{Чётные слагаемые}} + \underbrace{\sum_{k=0}^{n/2-1} x_{2k+1} e^{-\frac{(2k+1)p}{n}2\pi i}}_{\text{Нечётные слагаемые}} =$

$$= \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2}2\pi i}}_{\text{ДПФ над чётными слагаемыми}} + \underbrace{\sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kp}{n/2}2\pi i})}_{\text{ДПФ над нечётными слагаемыми}} \cdot \underbrace{e^{-\frac{p}{n}2\pi i}}_{\text{Коэффициент поворота}}$$

Быстрое преобразование Фурье…

Пусть $t = n/2 + p,\ p = \overline{0, \frac{n}{2} - 1}$

$$y_t = \sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kt}{n/2} 2\pi i} + \sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kt}{n/2} 2\pi i}) \cdot e^{-\frac{t}{n} 2\pi i} =$$

$$= \sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2} 2\pi i} \underbrace{e^{-kp2\pi i}}_{1} + \sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kp}{n/2} 2\pi i} \underbrace{e^{-kp2\pi i}}_{1}) \cdot e^{-\frac{p}{n} 2\pi i} \underbrace{\cdot e^{-\pi i}}_{-1} =$$

$$= \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2} 2\pi i}}_{a_p} - \underbrace{\sum_{k=0}^{n/2-1} (x_{2k+1} e^{-\frac{kp}{n/2} 2\pi i})}_{b_p} \cdot e^{-\frac{p}{n} 2\pi i}$$

Быстрое преобразование Фурье

- В результате получили:

$$y_p = \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2}2\pi i}}_{a_p} + \underbrace{\sum_{k=0}^{n/2-1} \left(x_{2k+1} e^{-\frac{kp}{n/2}2\pi i}\right)}_{b_p} \cdot e^{-\frac{p}{n}2\pi i}$$

$$t = n/2 + p$$

$$y_t = \underbrace{\sum_{k=0}^{n/2-1} x_{2k} e^{-\frac{kp}{n/2}2\pi i}}_{a_p} - \underbrace{\sum_{k=0}^{n/2-1} \left(x_{2k+1} e^{-\frac{kp}{n/2}2\pi i}\right)}_{b_p} \cdot e^{-\frac{p}{n}2\pi i}$$

БПФ. «Бабочка»

Рекуррентное соотношение «бабочка»

$$y_p = a_p + b_p \cdot e^{-\frac{p}{n}2\pi i}$$
$$y_{n/2+p} = a_p - b_p \cdot e^{-\frac{p}{n}2\pi i} \qquad p = \overline{0, \frac{n}{2} - 1}$$

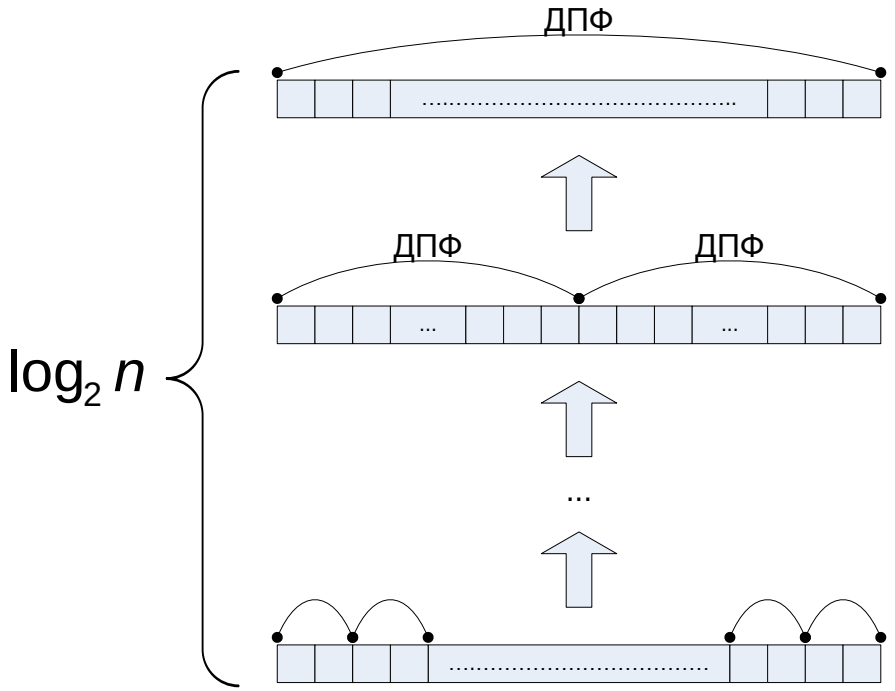
$a_0, a_1, \ldots, a_p,\ p = \overline{0, \frac{n}{2} - 1}$ - коэффициенты ПФ для четных элементов

$b_0, b_1, \ldots, b_p,\ p = \overline{0, \frac{n}{2} - 1}$ - коэффициенты ПФ для нечетных элементов

Для вычисления ДПФ над n элементами по соотношению «бабочки» необходимо вычислить два ДПФ размерности n/2.

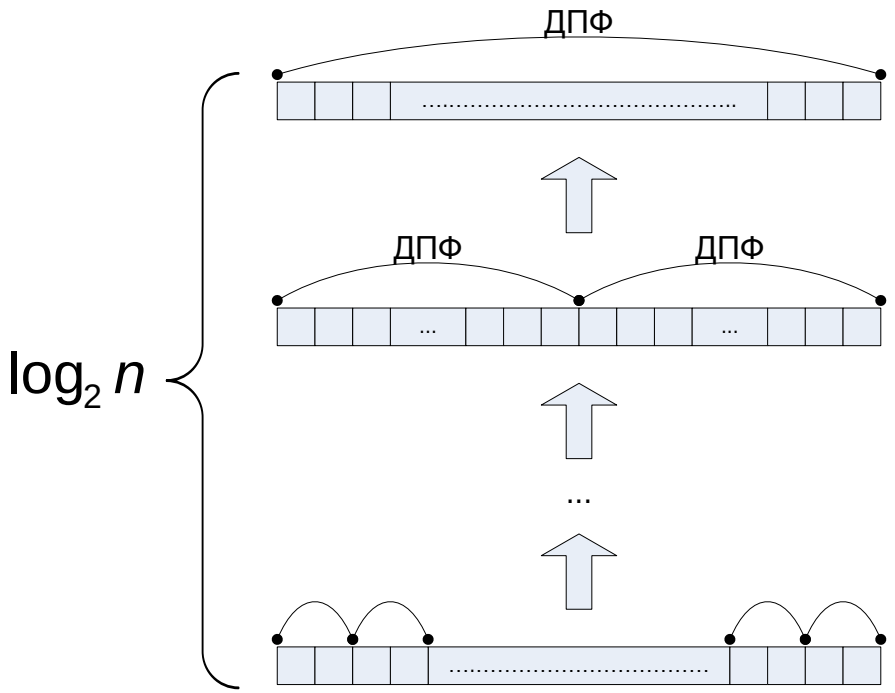
Быстрое преобразование Фурье

- Алгоритм Кули и Таки (Cooley-Tukey)
- Основан на рекуррентном соотношении
- Глубина рекурсии равна $\log_2 n$
- Трудоёмкость:



Быстрое преобразование Фурье

- Алгоритм Кули и Таки (Cooley-Tukey)
- Основан на рекуррентном соотношении
- Глубина рекурсии равна $\log_2 n$
- Трудоёмкость: $O(n \log n)$



Пример вычисления 4-точечного БПФ…

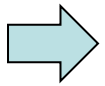
- Рассмотрим алгоритм работы БПФ на примере сигнала из 4-х точек, т.е. n=4
- К массиву из 4 комплексных чисел применим «бабочку»

x_0	x_1	x_2	x_3

$$y_p = a_p + b_p \cdot e^{-\frac{p}{2}\pi i}$$
$$y_{p+1} = a_p - b_p \cdot e^{-\frac{p}{2}\pi i} \qquad p = \overline{0,1}$$

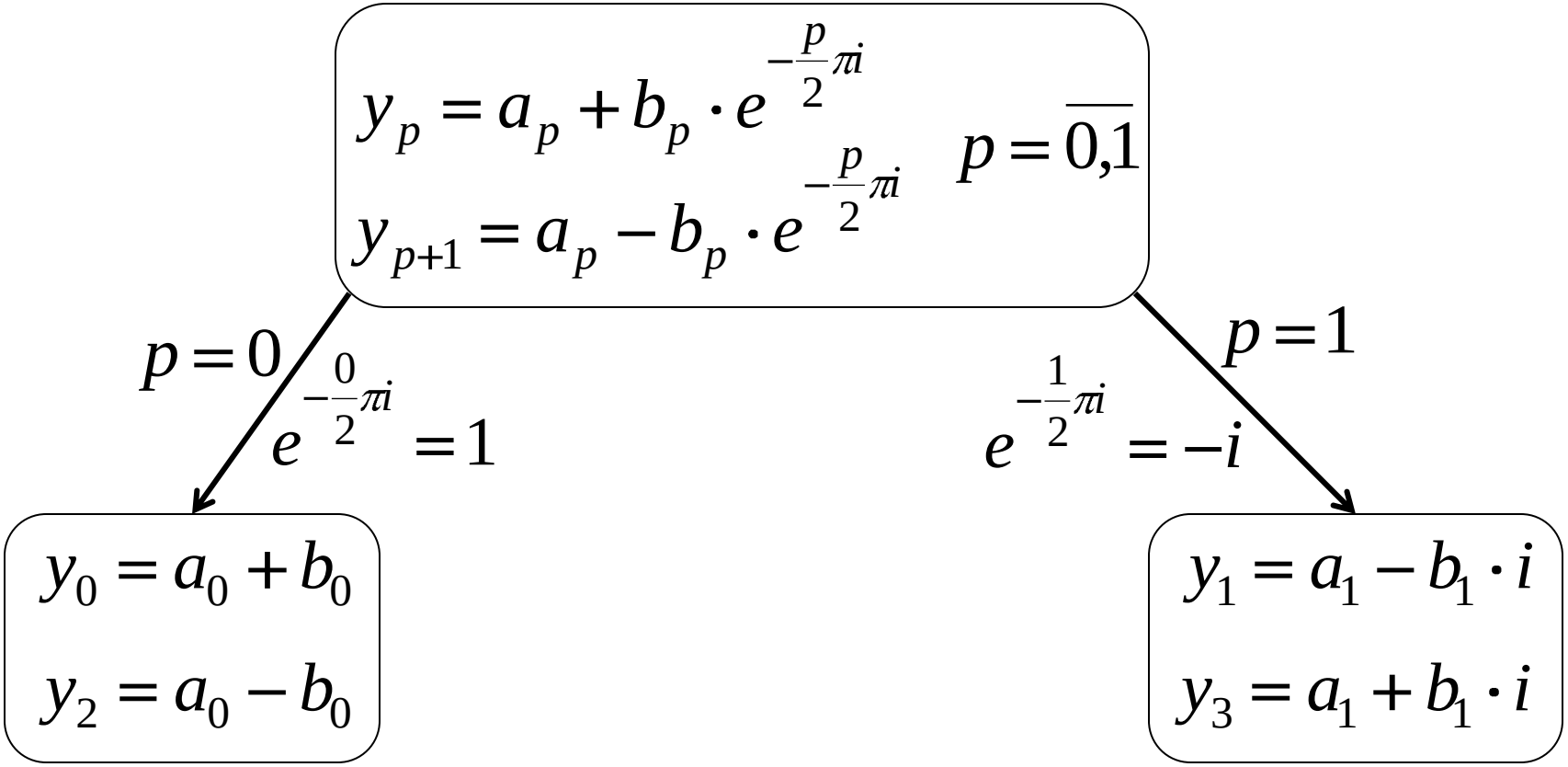
- Для простоты в дальнейших вычислениях выполним перестановку элементов массива (бит-реверсирование)

x_0	x_1	x_2	x_3



x_0	x_2	x_1	x_3

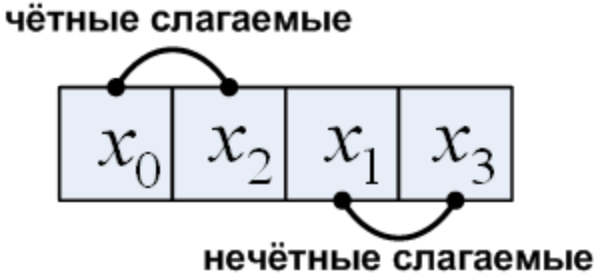
Пример вычисления 4-точечного БПФ…



- Чтобы посчитать коэффициенты Фурье y_0, y_1, y_2, y_3 необходимо вычислить коэффициенты a_0, a_1, b_0, b_1

Пример вычисления 4-точечного БПФ…

- Коэффициенты a_0, a_1 и b_0, b_1 являются коэффициентами преобразования Фурье над сигналами из 2-х точек (чётные и нечётные слагаемые)



- Для вычисления указанных коэффициентов снова воспользуемся соотношением «бабочки»

Пример вычисления 4-точечного БПФ…

- Для 2-точечного сигнала «бабочка»

$$y_p = a_p + b_p \cdot e^{-\frac{p}{n}2\pi i}$$
$$y_{n/2+p} = a_p - b_p \cdot e^{-\frac{p}{n}2\pi i} \qquad p = \overline{0, \frac{n}{2} - 1}$$

примет следующий вид для чётных слагаемых:

$$a_0 = a'_0 + b'_0$$
$$a_1 = a'_0 - b'_0 \qquad p = 0$$

Пример вычисления 4-точечного БПФ…

- Коэффициенты a'_0, b'_0 являются коэффициентами Фурье для сигнала, состоящего из одной точки:
  - a'_0 для x_0;
  - b'_0 для x_2.
- Коэффициент Фурье для множества из одной точки является самой точкой, поэтому:
  - $a'_0 = x_0$;
  - $b'_0 = x_2$.
- a''_0, b''_0 - коэффициенты Фурье для сигнала, состоящего из одной точки (нечетные слагаемые):
  - $a''_0 = x_0$;
  - $b''_0 = x_2$.

a'_0	b'_0	a''_0	b''_0

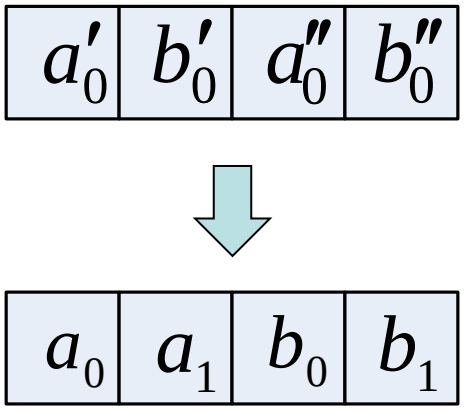
Пример вычисления 4-точечного БПФ…

- Вычислим коэффициенты 2-точечного БПФ:

$$a_0 = a'_0 + b'_0 \qquad a_1 = a'_0 - b'_0$$

$$b_0 = a''_0 + b''_0 \qquad b_1 = a''_0 - b''_0$$

- Результат запишем в исходный массив:



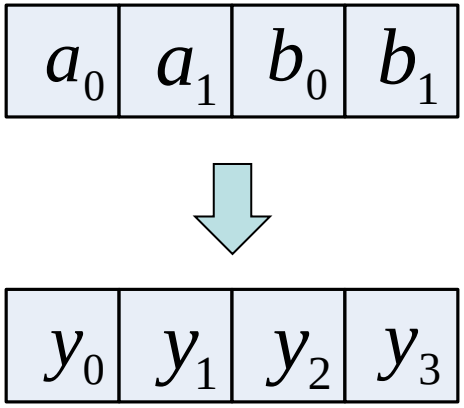
Пример вычисления 4-точечного БПФ…

- Вычислим искомые коэффициенты (4-точечного БПФ):

$$y_0 = a_0 + b_0 \qquad y_1 = a_1 - b_1 \cdot i$$

$$y_2 = a_0 - b_0 \qquad y_3 = a_1 + b_1 \cdot i$$

- Результат запишем в исходный массив:



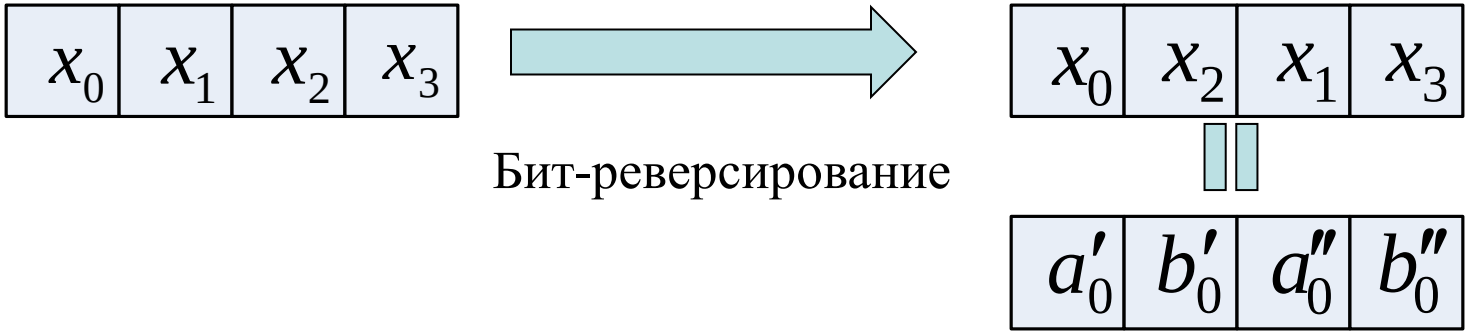
Пример вычисления 4-точечного БПФ…

- 1. Перестановка элементов массива

Бит-реверсирование

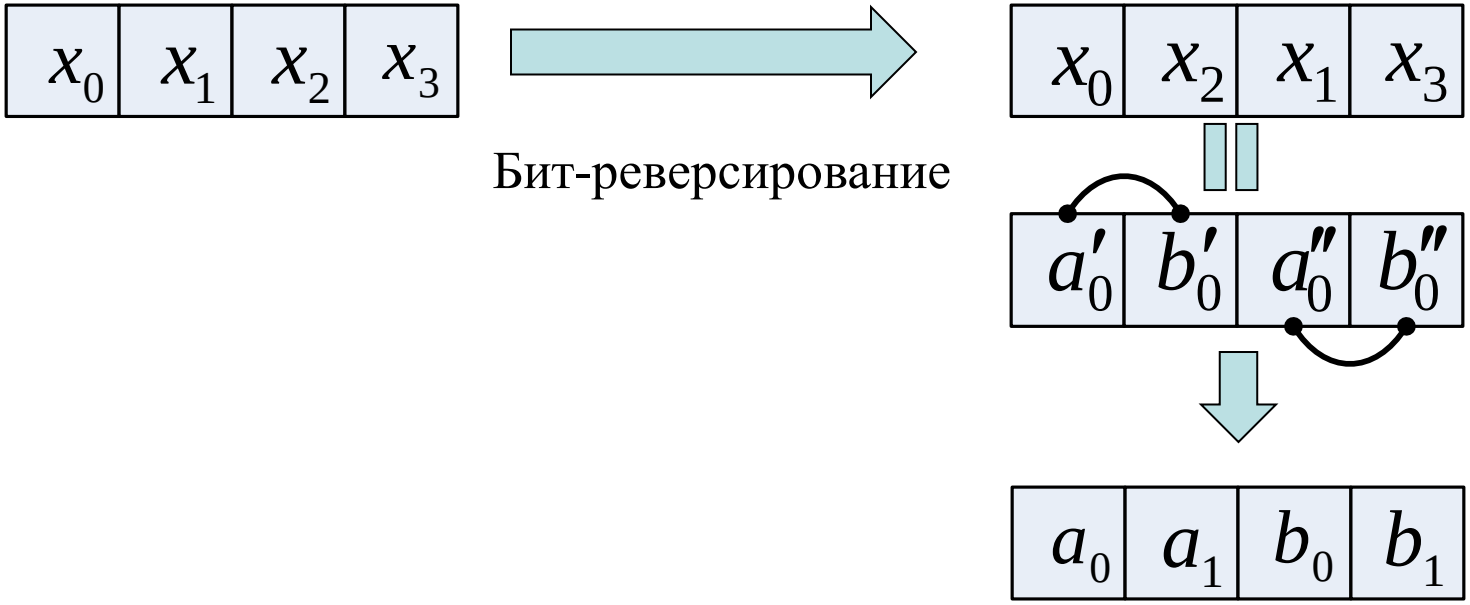
Пример вычисления 4-точечного БПФ…

- 2. Выполнение 1-точечного ПФ



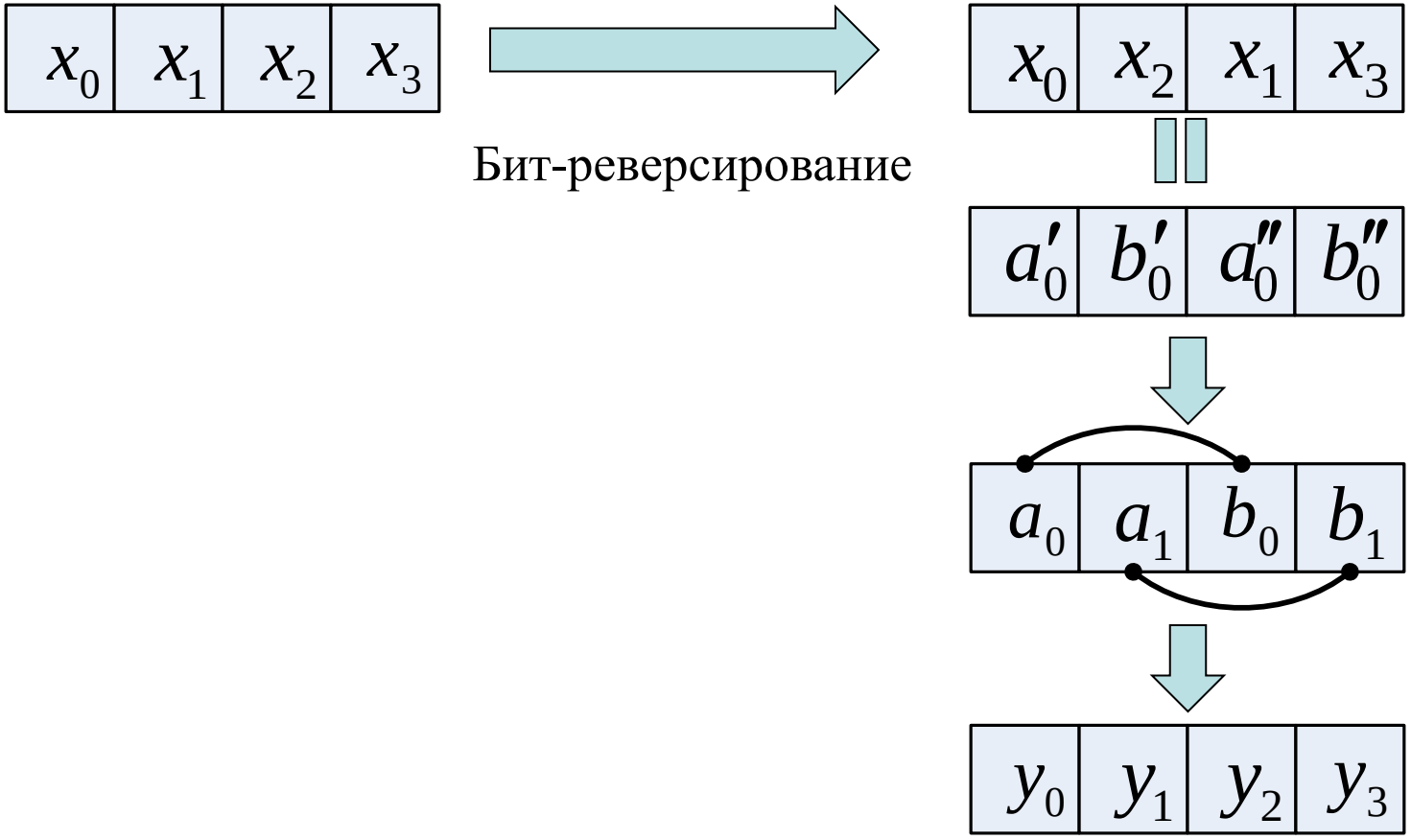
Пример вычисления 4-точечного БПФ…

- 3. Выполнение 2-точечного ПФ

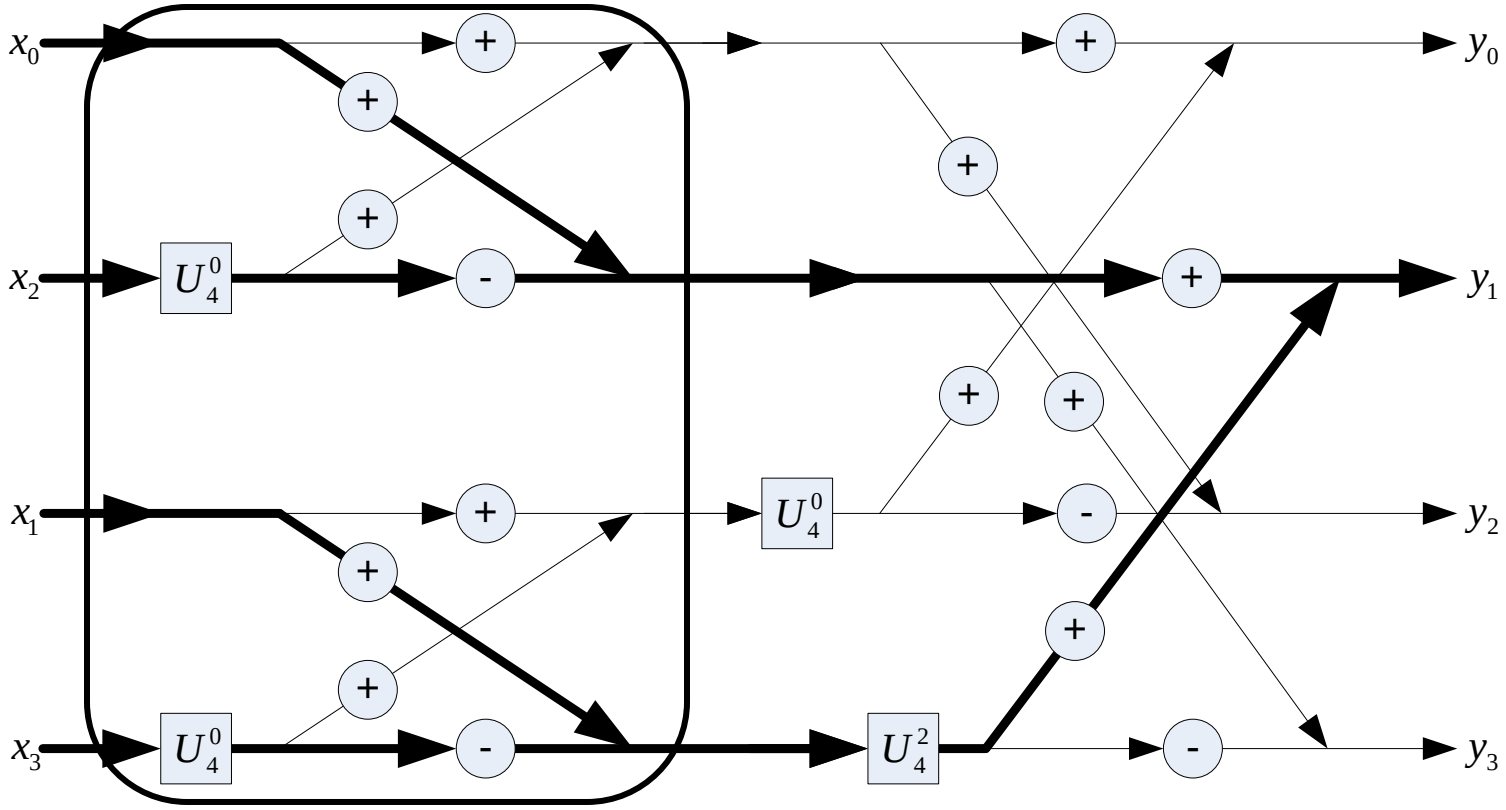


Пример вычисления 4-точечного БПФ…

- 4. Выполнение 4-точечного ПФ



Демонстрация вычисления 4-точечного БПФ

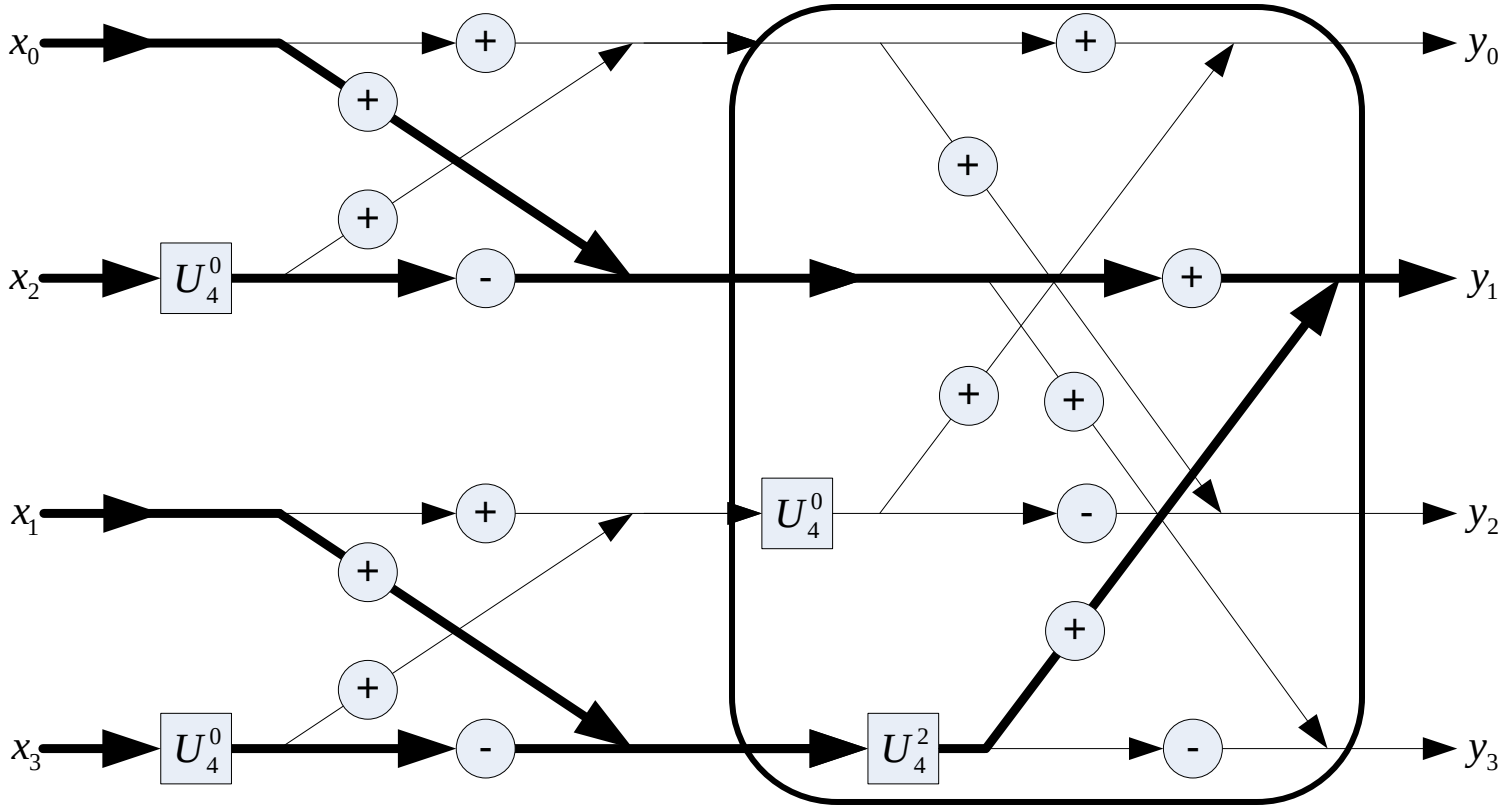


$$U_n^k = e^{-\frac{k}{n}\pi i}$$

$$U_4^0 = e^0 = 1$$

$$U_4^2 = e^{-\frac{\pi}{2}i} = -i$$

Демонстрация вычисления 4-точечного БПФ



1-ый этап

2-ой этап

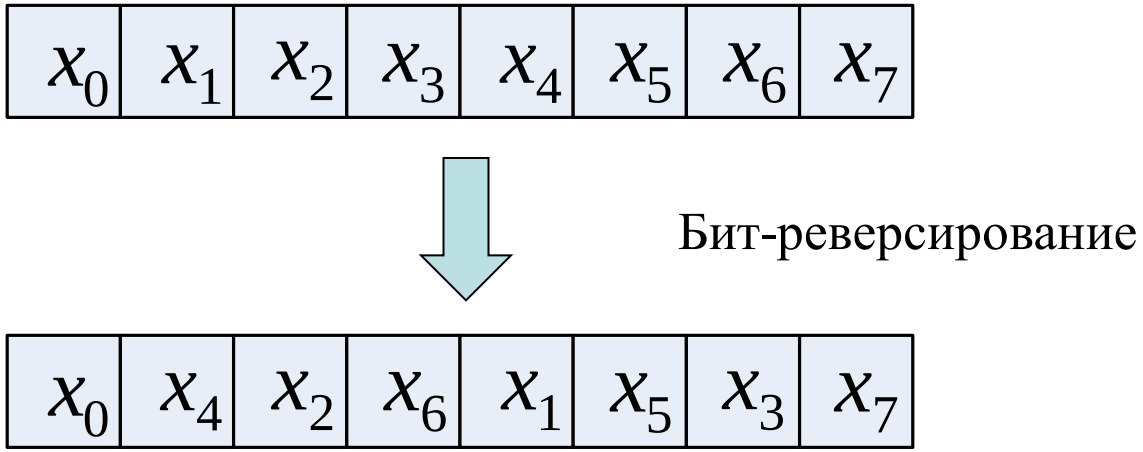
$$U_n^k = e^{-\frac{k}{n}\pi i}$$

$$U_4^0 = e^0 = 1$$

$$U_4^2 = e^{-\frac{\pi}{2}i} = -i$$

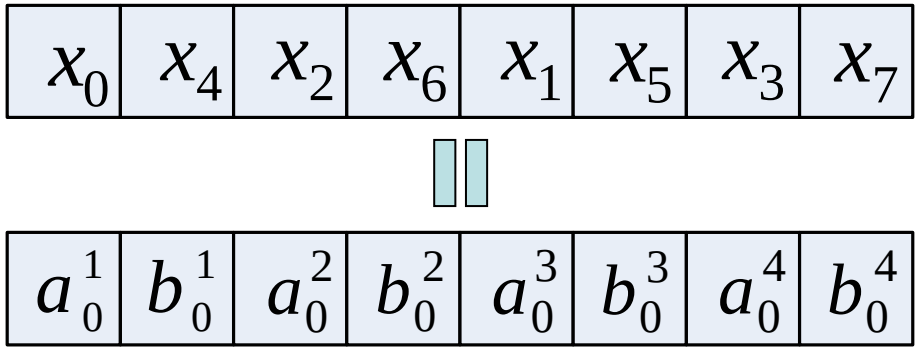
Пример вычисления 8-точечного БПФ…

- 1. Перестановка элементов массива



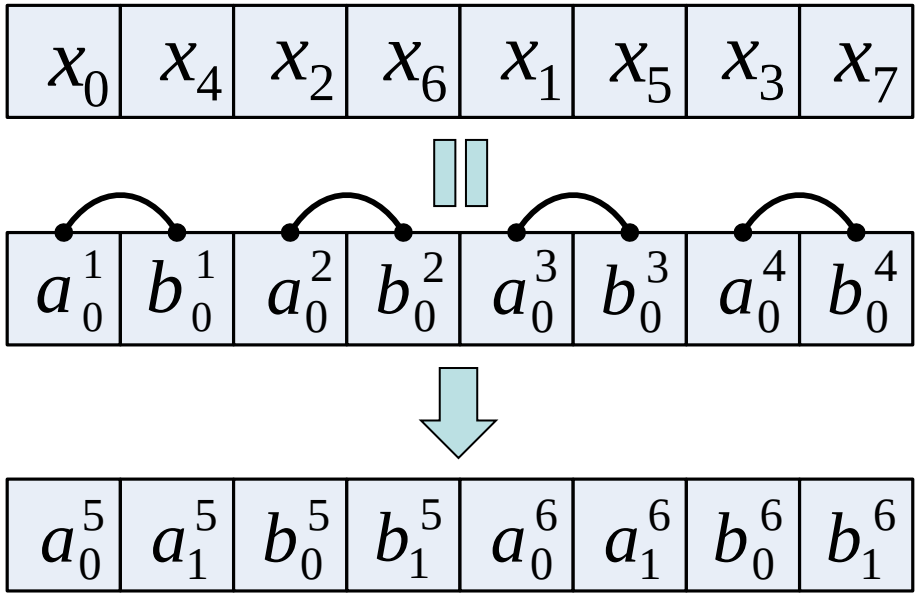
Пример вычисления 8-точечного БПФ…

- 2. Выполнение 1-точечного ПФ



Пример вычисления 8-точечного БПФ…

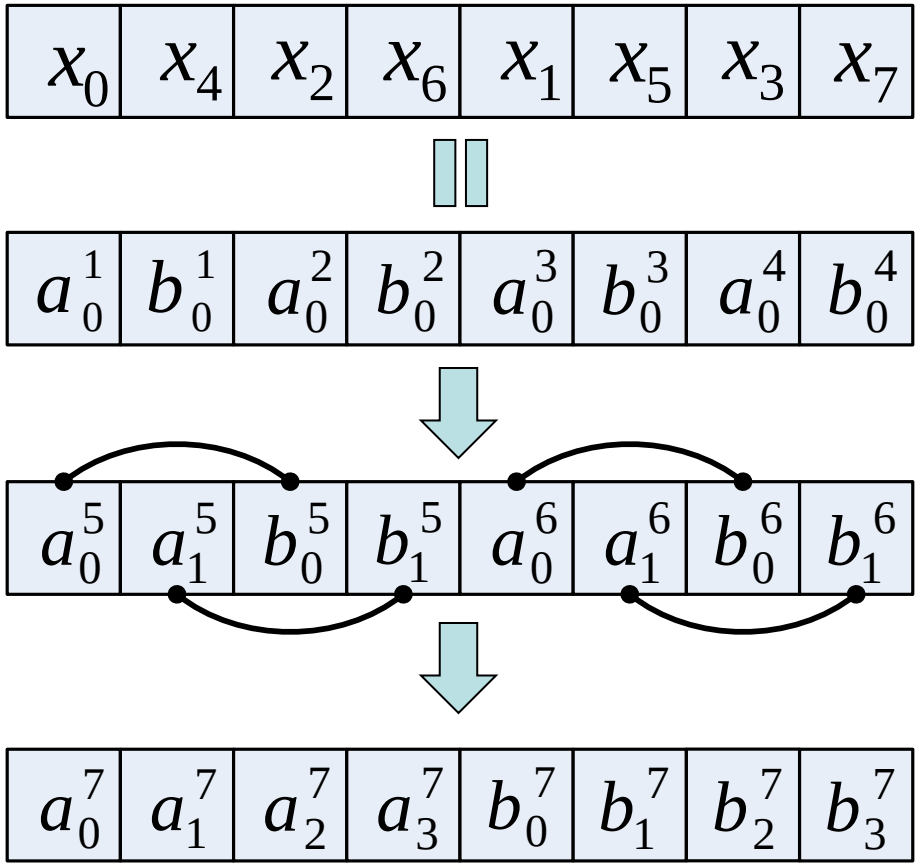
- 3. Выполнение 2-точечного ПФ



$$a_0^5 = a_0^1 + b_0^1$$
$$a_1^5 = a_0^1 - b_0^1$$
$$b_0^5 = a_0^2 + b_0^2$$
$$b_1^5 = a_0^2 - b_0^2$$
$$\dots$$

Пример вычисления 8-точечного БПФ…

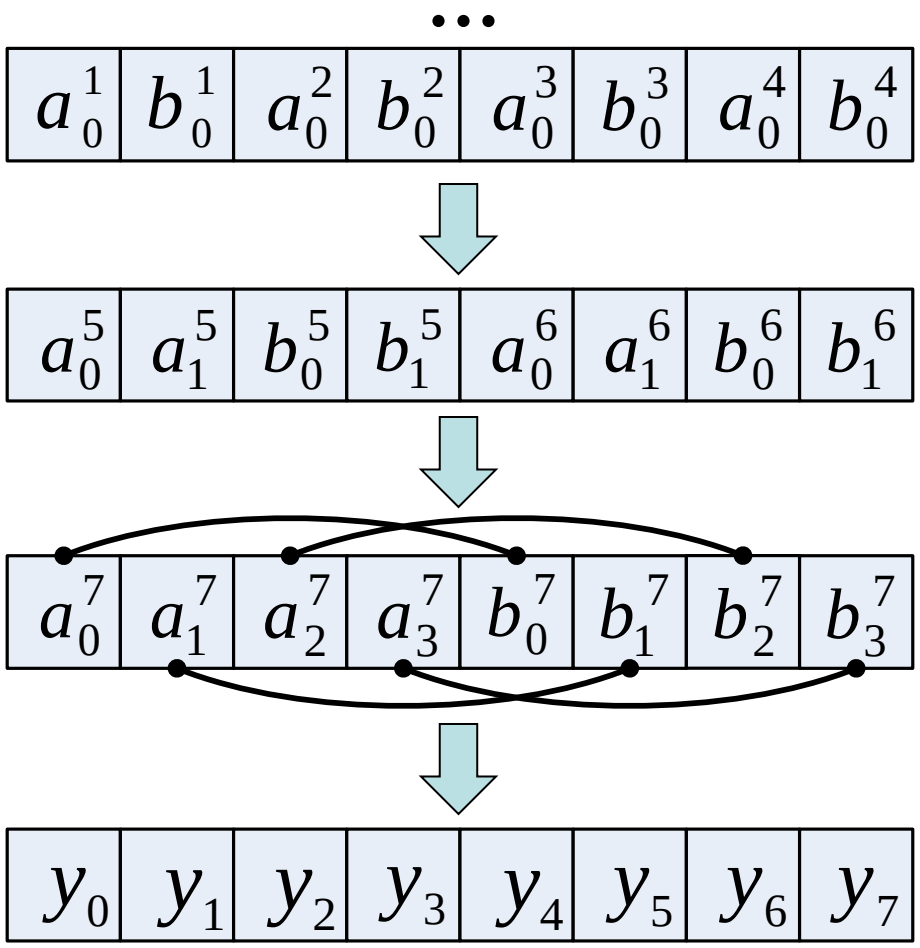
- 4. Выполнение 4-точечного ПФ



$$a_0^7 = a_0^5 + b_0^5$$
$$a_2^7 = a_0^5 - b_0^5$$
$$a_1^7 = a_1^5 - b_1^5 \cdot i$$
$$a_3^7 = a_1^5 + b_1^5 \cdot i$$
$$b_0^7 = a_0^6 + b_0^6$$
$$b_2^7 = a_0^6 - b_0^6$$
$$b_1^7 = a_1^6 - b_1^6 \cdot i$$
$$b_3^7 = a_1^6 + b_1^6 \cdot i$$

Пример вычисления 8-точечного БПФ…

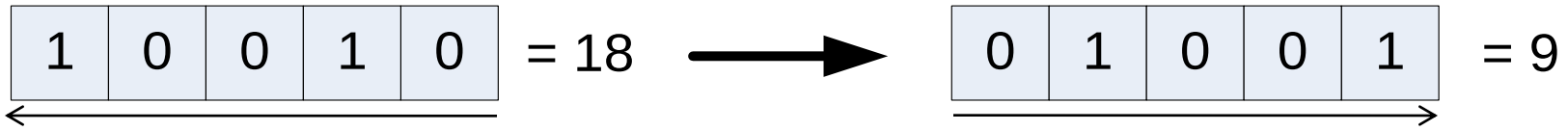
- 5. Выполнение 8-точечного ПФ



$$y_0 = a_0^7 + b_0^7$$
$$y_4 = a_0^7 - b_0^7$$
$$y_1 = a_1^7 + b_1^7 \cdot (\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2} i)$$
$$y_5 = a_1^7 - b_1^7 \cdot (\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2} i)$$
$$y_2 = a_2^7 - b_2^7 \cdot i$$
$$y_6 = a_2^7 + b_2^7 \cdot i$$
$$y_3 = a_3^7 + b_3^7 \cdot (-\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2} i)$$
$$y_7 = a_3^7 - b_3^7 \cdot (-\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2} i)$$

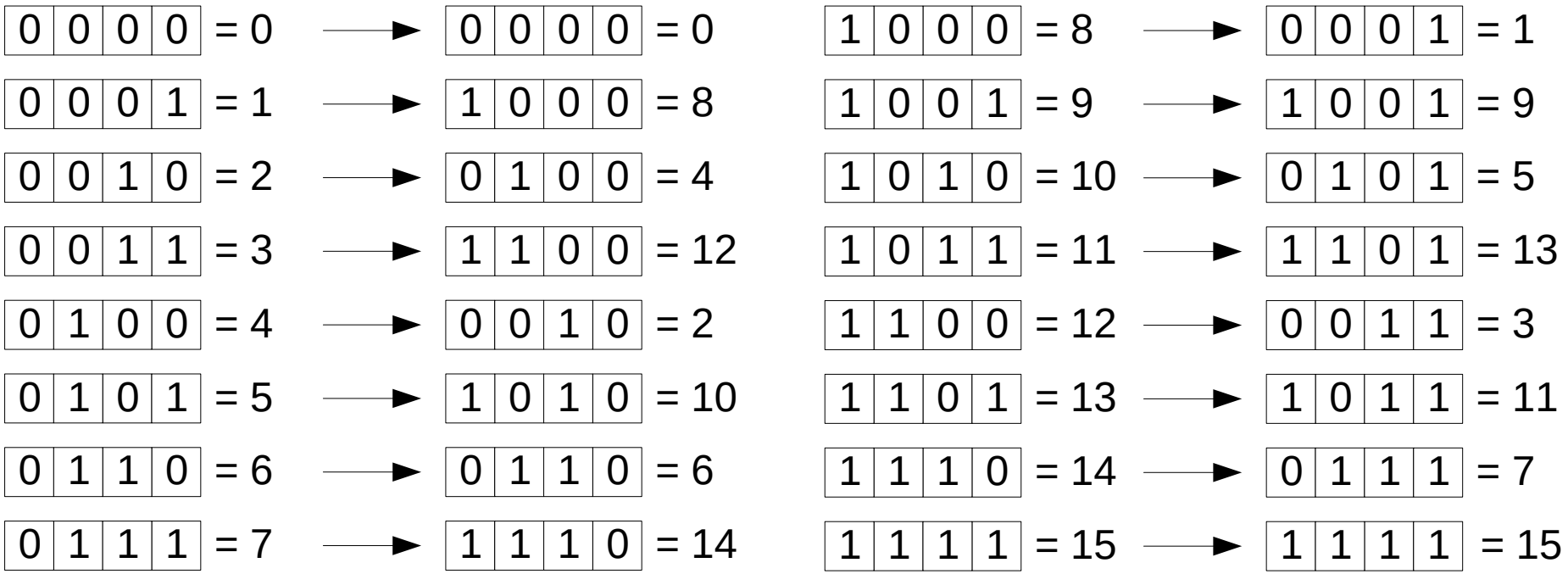
Бит-реверсирование

- Переупорядочивание элементов массива
- Бит-реверсирование — это преобразование двоичного числа заключающееся в изменении порядка следования бит на противоположный



Бит-реверсирование. Пример

- Пример изменения индексов элементов при выполнении бит-реверсирования над массивом из 16 элементов



Бит-реверсирование. Реализация

```
//in:  ind - обычный порядок следования бит
//out: revInd - бит-реверсивный порядок следования бит
//size = 1 << bitsCount;

int mask = 1 << (bitsCount - 1);
revInd = 0;

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

Первая итерация (1)

ind	1	0	0	1	0
mask	1	0	0	0	0
val	0	0	0	0	**1**
revInd	0	0	0	0	**1**

Бит-реверсирование. Реализация

```
//in:  ind - обычный порядок следования бит
//out: revInd - бит-реверсивный порядок следования бит
//size = 1 << bitsCount;

int mask = 1 << (bitsCount - 1);
revInd = 0;

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

Вторая итерация (2)

ind	1	0	0	1	0
mask	0	1	0	0	0
val	0	0	0	0	**0**
revInd	0	0	0	**0**	1

Бит-реверсирование. Реализация

```
//in:  ind - обычный порядок следования бит
//out: revInd - бит-реверсивный порядок следования бит
//size = 1 << bitsCount;

int mask = 1 << (bitsCount - 1);
revInd = 0;

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

Третья итерация (3)

ind	1	0	0	1	0
mask	0	0	1	0	0
val	0	0	0	0	**0**
revInd	0	0	**0**	0	1

Бит-реверсирование. Реализация

```
//in:  ind - обычный порядок следования бит
//out: revInd - бит-реверсивный порядок следования бит
//size = 1 << bitsCount;

int mask = 1 << (bitsCount - 1);
revInd = 0;

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

Четвёртая итерация (4)

ind	1	0	0	1	0
mask	0	0	0	1	0
val	0	0	0	0	**1**
revInd	0	**1**	0	0	1

Бит-реверсирование. Реализация

```
//in:  ind - обычный порядок следования бит
//out: revInd - бит-реверсивный порядок следования бит
//size = 1 << bitsCount;

int mask = 1 << (bitsCount - 1);
revInd = 0;

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

Пятая итерация (5)

ind	1	0	0	1	0
mask	0	0	0	0	1
val	0	0	0	0	**0**
revInd	**0**	1	0	0	1

БПФ. Алгоритм

- Бит-реверсирование
- Рекурсия (БПФ)
  - Вычисление БПФ над первой половиной массива
  - Вычисление БПФ над второй половиной массива
  - Применение «бабочки» ко всем парам элементов

$$y_p = a_p + b_p \cdot e^{-\frac{p}{n}2\pi i}$$

$$y_{n/2+p} = a_p - b_p \cdot e^{-\frac{p}{n}2\pi i} \qquad p = \overline{0, \frac{n}{2} - 1}$$

Реализация БПФ

- FFTW (Fastest Fourier Transform in the West)
  - библиотека для вычисления ДПФ;
  - GNU GPL.
- Intel MKL (Math Kernel Library)
  - содержит реализации большого количества алгоритмов;
  - оптимизирована для процессоров Intel;
  - имеет реализацию интерфейсов FFTW.

5. “Не пей эту гадость. Она... не синхронизирована с эволюцией!”
(про молоко)

Шустрый синус

High five…
...low five...
...down low...
...too slow.

“Горячая” точка

- Итак, мы воспользовались Intel Parallel Amplifier и обнаружили, что очень долго происходит вычисление функций sin/cos.

```
37  void SerialFFTCalculation(complex<double> *signal, int first, int size,
38  {                                                                  0.547s
39    if(size==1)                                                      0.078s
40      return;
41
42    double const coeff=2.0*PI/size;                                  0.313s
43
44    SerialFFTCalculation( signal, first, size/2, forward);           0.047s
45    SerialFFTCalculation( signal, first + size/2, size/2, forward);  0.031s
46
47    for (int  j=first; j<first+size/2; j++ )                         0.328s
48      if(forward)                                                    0.515s
49        Butterfly(signal, complex<double>(cos(-j*coeff), sin(-j*coeff)), j  11.187s
50      else
51        Butterfly(signal, complex<double>(cos(j*coeff), sin(j*coeff)), j  17.651s
52  }                                                                  0.625s
```

- Чтобы ускорить работу указанного участка кода, необходимо вспомним тригонометрические тождества.

Вспомним математику

- Представим экспоненту в тригонометрической форме:

$$e^{-\frac{p}{n}2\pi i} = \cos(\frac{p}{n}2\pi) - i\sin(\frac{p}{n}2\pi) \qquad p = 0, \overline{\frac{n}{2} - 1}$$

- Вспомним тригонометрические тождества:

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

- Для фиксированного n получаем:

$$\sin(p\frac{2\pi}{n}) = \sin((p-1)\frac{2\pi}{n})\cos(\frac{2\pi}{n}) + \cos((p-1)\frac{2\pi}{n})\sin(\frac{2\pi}{n})$$

$$\cos(p\frac{2\pi}{n}) = \cos((p-1)\frac{2\pi}{n})\cos(\frac{2\pi}{n}) - \sin((p-1)\frac{2\pi}{n})\sin(\frac{2\pi}{n})$$

Оптимизация вычисления тригонометрических функций

- Чтобы посчитать sin и cos, необходимо выполнить 4 операции умножения и по одной операции сложения и вычитания:

$$\sin(p\frac{2\pi}{n}) = \sin((p-1)\frac{2\pi}{n})\cos(\frac{2\pi}{n}) + \cos((p-1)\frac{2\pi}{n})\sin(\frac{2\pi}{n})$$

$$\cos(p\frac{2\pi}{n}) = \cos((p-1)\frac{2\pi}{n})\cos(\frac{2\pi}{n}) - \sin((p-1)\frac{2\pi}{n})\sin(\frac{2\pi}{n})$$

- Т.к. n фиксированное, то коэффициенты $\cos(\frac{2\pi}{n})$ и $\sin(\frac{2\pi}{n})$ достаточно посчитать один раз.

Применение оптимизации

- В соответствие с указанными соотношениями выполним оптимизацию вычисления функций *sin* и *cos*.
- В цикле по *j* от *first* до *first+size/2* необходимо посчитать все значения *sin(j*coeff)* и *cos(j*coeff)*.

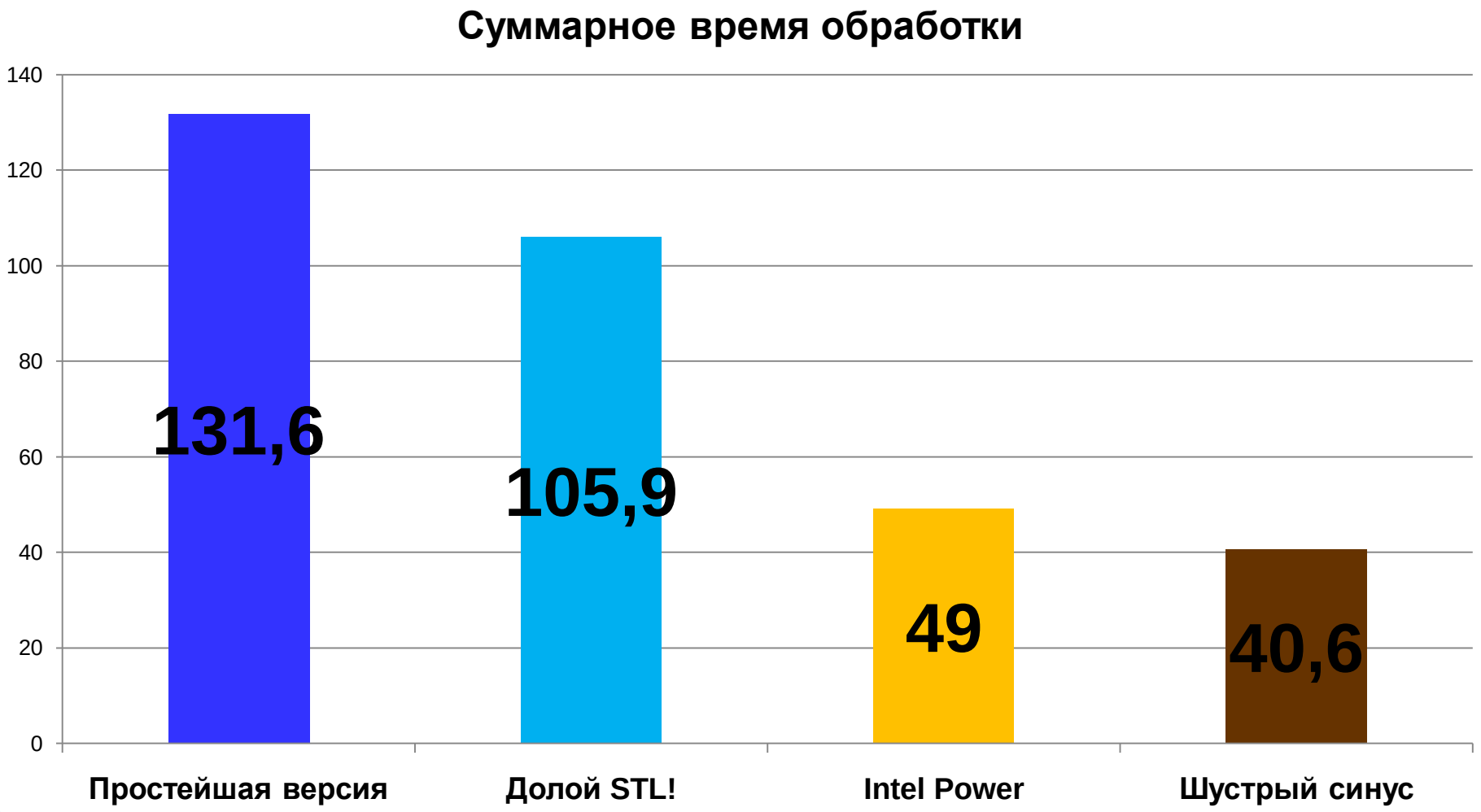
1. Посчитаем первые *sin* и *cos*: *sin(first*coeff)* и *cos(first*coeff)*.
2. Посчитаем множители: *sin(coeff)* и *cos(coeff)*.
3. Далее в цикле по *j* для вычисления *sin(j*coeff)* и *cos(j*coeff)* будем использовать предыдущие значения *sin* и *cos* для *j-1* и множители в соответствии с ранее выведенными соотношениями.

Оценка эффективности

- **Откройте проект “.\5. filter (fastest sin)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 40.6 с

Прогресс оптимизации



Ищём следующее проблемное место

- Снова ищем “горячие точки”. Воспользуйтесь Intel Parallel Amplifier:
  - “Release” версия программы;
  - режим работы “Hotspots”.

Наиболее медленные функции

- Снова большую часть времени работает функция SerialFFTCalculation:

SerialFFTCalculation	24.447s	filter.exe
ProcessFrame	16.033s	filter.exe
SerialInverseFFT	0.094s	filter.exe
free	0.047s	filter.exe
SerialFFT	0.016s	filter.exe
[Unknown frame(s)]	0.016s	[Unknown]
MainWindowProc	0.016s	highgui110.dll
GrFmtFilterFactory::CheckFile	0.016s	highgui110.dll
CvCaptureAVI_VFW::open	0.016s	highgui110.dll
cvSetTrackbarPos	0.002s	highgui110.dll
main	0.001s	filter.exe
icvUpdateTrackbar	0.000s	highgui110.dll
icvFindWindowByName	0.000s	highgui110.dll

“Горячая” точка

```
59
60    SerialFFTCalculation( signal, first, size/2, forward);                    0.063s
61    SerialFFTCalculation( signal, first + size/2, size/2, forward);           0.063s
62
63    for (int  j=first; j<first+size/2; j++ )                                  0.156s
64      if(forward)                                                             0.094s
65      {
66        Butterfly(signal, complex<double>(uRf, uIf), j , size/2 );            3.515s
67        uRtmp=uRf;                                                            0.140s
68        uRf=uRf*wRf-uIf*wIf;                                                  0.203s
69        uIf=uIf*wRf+uRtmp*wIf;                                                0.094s
70      }
71      else
72      {
73        Butterfly(signal, complex<double>(uR, uI), j , size/2 );              3.076s
74        uRtmp=uR;                                                             0.156s
75        uR=uR*wR-uI*wI;                                                       0.063s
76        uI=uI*wR+uRtmp*wI;                                                    0.141s
77      }
78  }                                                                           0.703s
```

- Функция *Butterfly* занимает много времени на вычисления, т.к. это наиболее часто вызываемая функция.
- Попробуем избавиться от лишних ветвлений в цикле.

6. “Никогда не стоит недооценивать предсказуемость тупизны!”

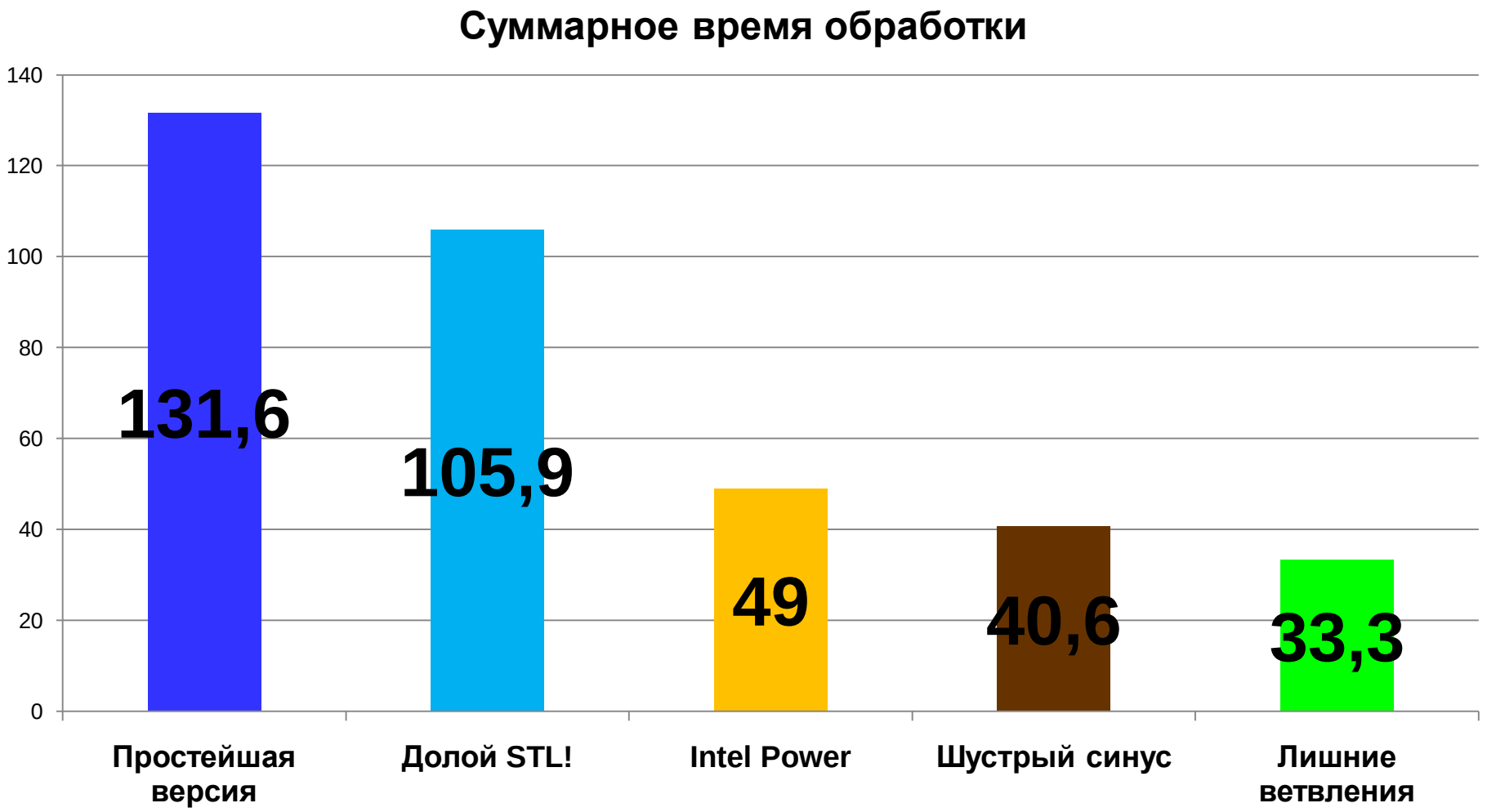
Лишние ветвления

Оценка эффективности

- **Откройте проект “.\6. filter (excess branch)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 33.3 с

Прогресс оптимизации



Дальнейшая оптимизация

- Снова выполняем профилирование. Воспользуйтесь Intel Parallel Amplifier:
  - “Release” версия программы;
  - режим работы “Hotspots”.

Удаление “лишних” вызовов

- Функция *SerialFFTCalculation* всё ещё остаётся горячей точкой программы, поэтому продолжим её оптимизацию.

Теорема об удалении рекурсии.

По каждому алгоритму можно построить эквивалентный алгоритм без рекурсии.

- Избавимся от рекурсии.
- Избавимся от “лишних” вызовов функций (сделаем явный inline функций).

7. Сколько веревочке не виться…

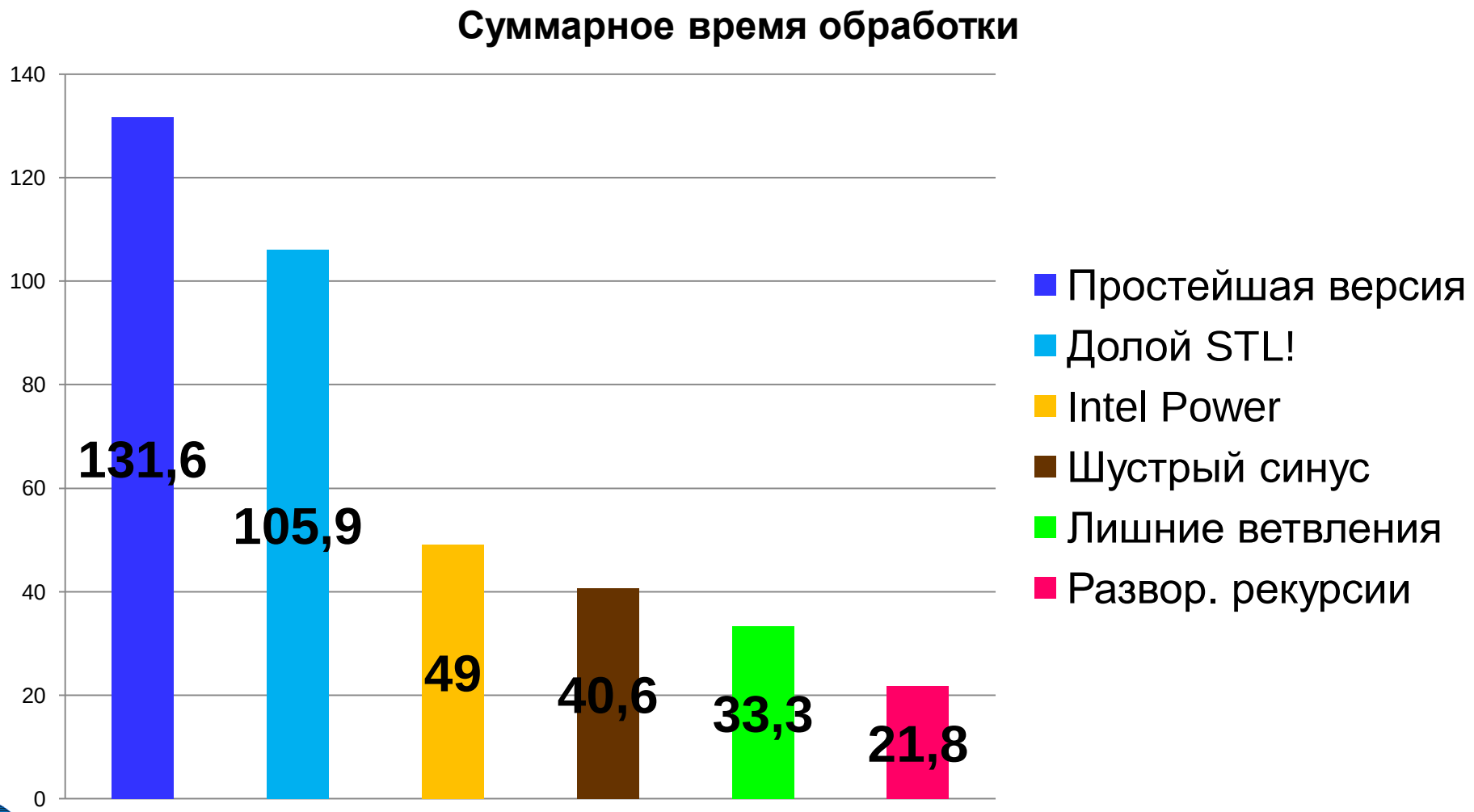
Развёртка рекурсии

Оценка эффективности

- **Откройте проект “.\7. filter (kill recursion)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 21.8 с

Прогресс оптимизации



Снова профилировка

- С помощью Intel Parallel Amplifier выполните профилировку программы:
  - “Release” версия программы;
  - режим работы “Hotspots”.

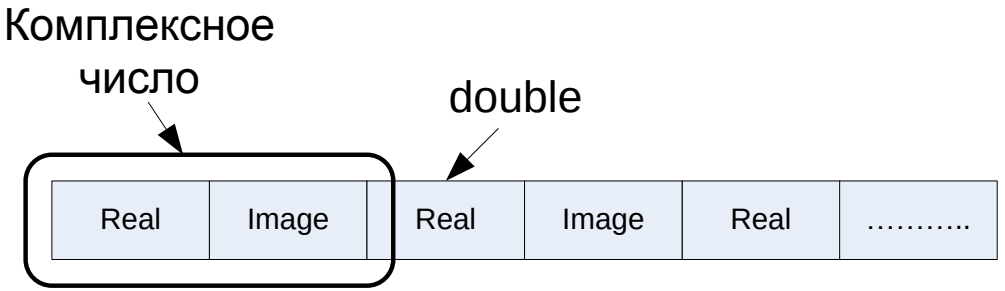
“Горячие точки”

- Достаточно предсказуемо выглядит результат работы профилировщика. Функции выполняющие БПФ работают наибольшее время:

⊞ SerialInverseFFT	8.044s	filter.exe
⊞ SerialFFT	4.062s	filter.exe
⊞ ProcessFrame	3.680s	filter.exe
⊞ log	2.264s	filter.exe
⊞ exp	2.095s	filter.exe
⊞ log	0.078s	filter.exe
⊞ exp	0.047s	filter.exe
⊞ LdrDisableThreadCalloutsForDll	0.031s	ntdll.dll
⊞ [Unknown frame(s)]	0.012s	[Unknown]
⊞ CvCaptureAVI_VFW::grabFrame	0.007s	highgui110.dll
⊞ cvSetTrackbarPos	0.002s	highgui110.dll
⊞ main	0.001s	filter.exe
⊞ icvFindWindowByName	0.000s	highgui110.dll
⊞ cvInitSystem	0.000s	highgui110.dll

Перейдём к native типам 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:vector*.
- Будем представлять комплексное число в виде последовательности из двух элементов типа *double*.



8.

“— Алле!

— Кажется я не туда попал?!

— Цельс лучше!”

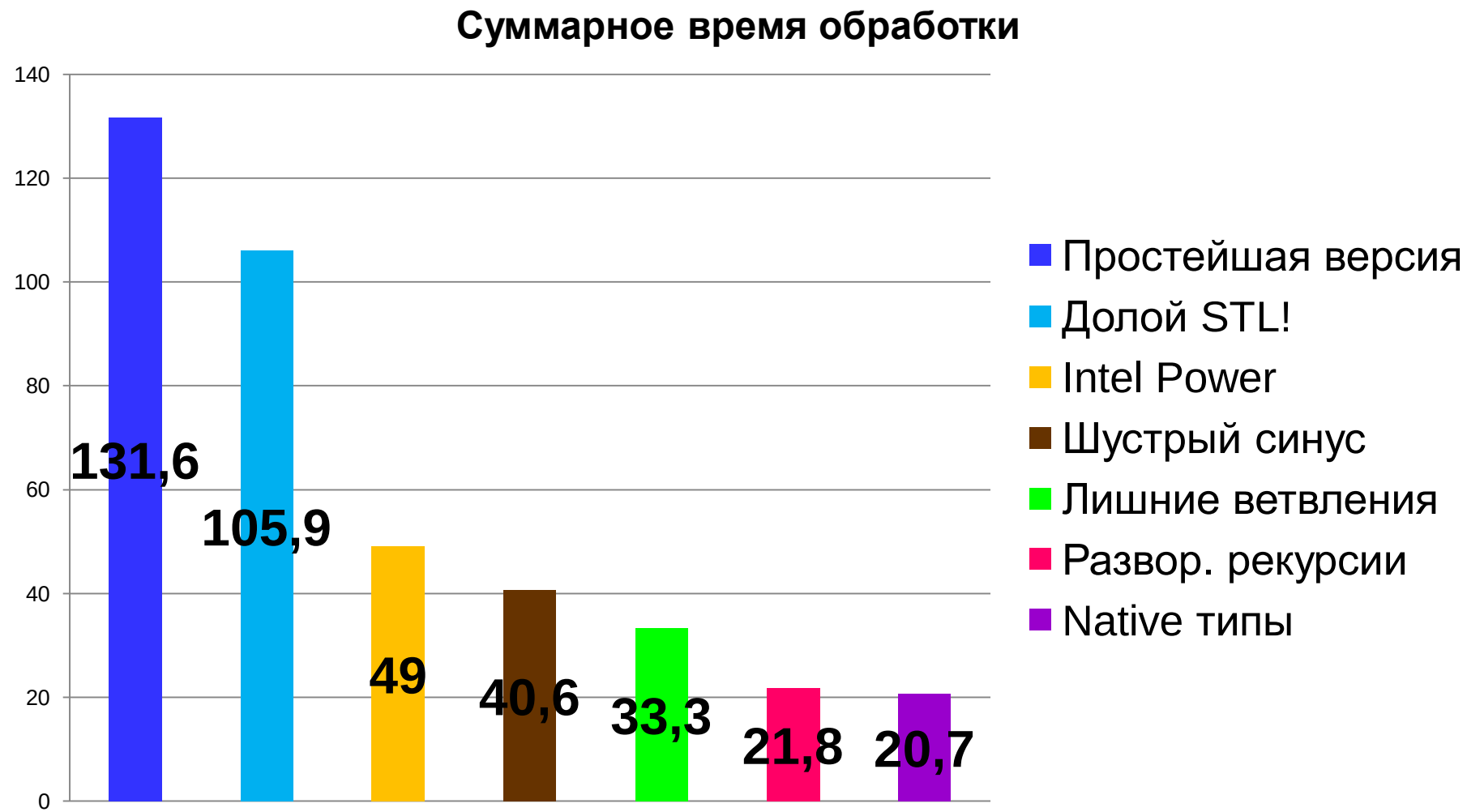
Native типы данных

Оценка эффективности

- **Откройте проект “.\8. filter (eliminate complex)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 20.7 с

Прогресс оптимизации

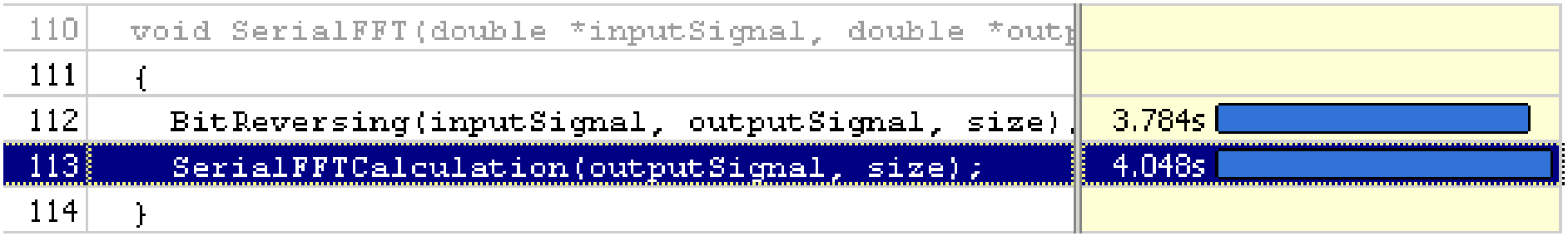


Снова профилировка

- Оценим “вклад” бит-реверсирования в вычисление БПФ.
- **С помощью Intel Parallel Amplifier выполните профилировку программы:**
  - **“Release” версия программы;**
  - **режим работы “Hotspots”.**

Анализ результатов профилирования

- Функция, выполняющая перестановку элементов массива с помощью бит-реверсирования, занимает достаточно много времени, чтобы заняться её оптимизацией.

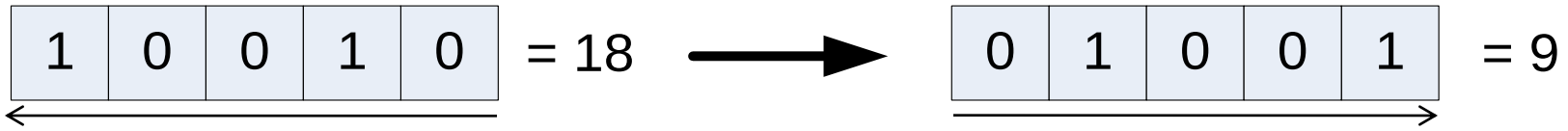


9. “Ну и какой у тебя план, Пендальф? Да-а-а, план у тебя - чистый термояд…”

Делайте всё заранее

Бит-реверсирование

- Переупорядочивание элементов массива
- Бит-реверсирование — это преобразование двоичного числа заключающееся в изменении порядка следования бит на противоположный



Бит-реверсирование

```
//in:  ind - обычный порядок следования бит
//out: revInd - бит-реверсивный порядок следования бит
//size = 1 << bitsCount;

int mask = 1 << (bitsCount - 1);
revInd = 0;

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

Бит-реверсирование

```
//in:  ind - обычный порядок следования бит
//out: revInd - бит-реверсивный порядок следования бит
//size = 1 << bitsCount;

int mask = 1 << (bitsCount - 1);
revInd = 0;

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

Для пересчёта каждого индекса необходимо выполнить цикл!

Оптимизация бит-реверсирования

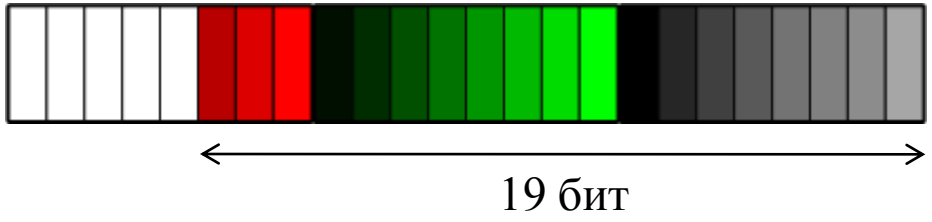
- Метод Васи Пупкина
  - Давайте заранее всё посчитаем и во время реверсирования будем подставлять посчитанные значения!

Оптимизация бит-реверсирования

- Создадим таблицу в которой будут содержаться все однобайтовые величины и их реверсивные аналоги (256 значений).
- Исходное число, которое необходимо реверсировать, разделим на байты.
- Каждый байт заменим на соответствующий аналог из таблицы.
- Выполним сдвиг вправо полученного результата, если это необходимо.

Пример бит-реверсирования

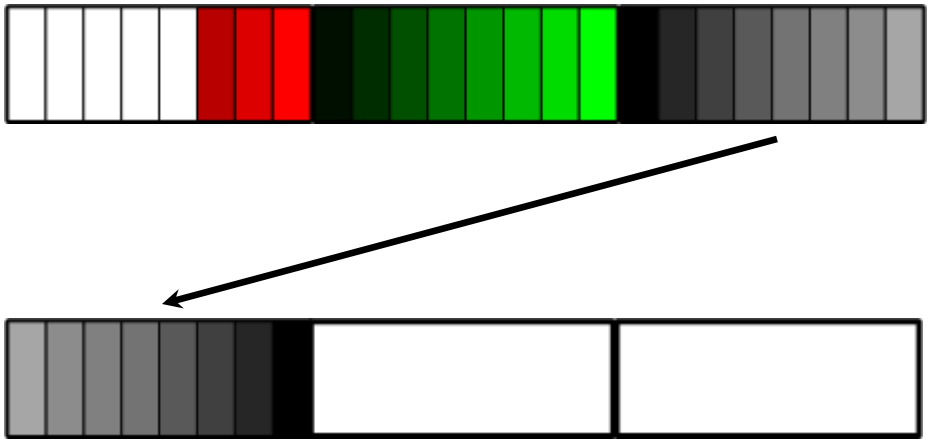
- Рассмотрим пример, в котором необходимо реверсировать число содержащее 19 значимых бит
- Исходное число



- Результат должен быть записан так же в 3 байта

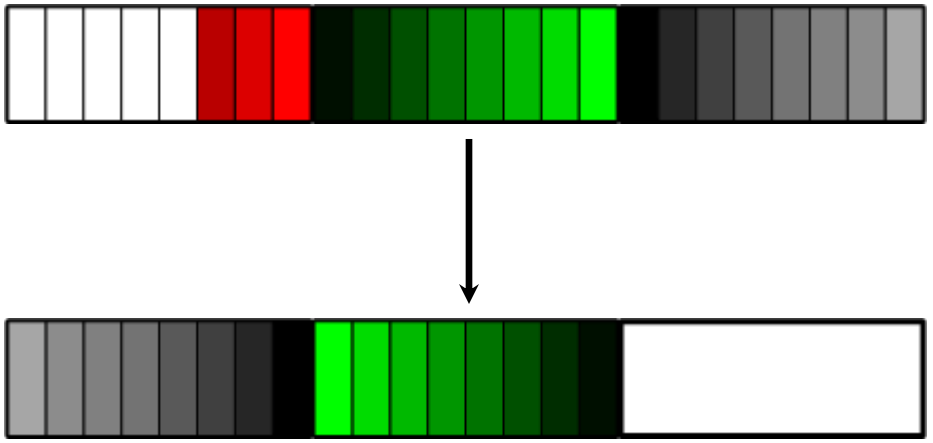
Пример бит-реверсирования

- Записываем первый байт в инверсивном порядке (1)



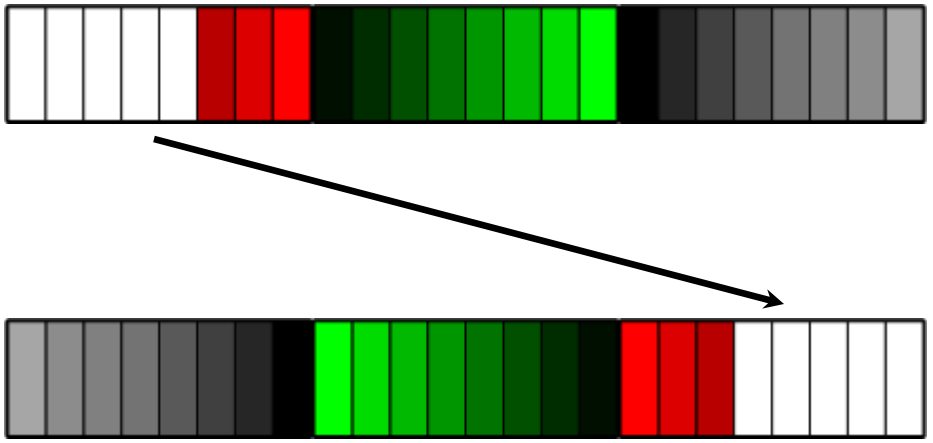
Пример бит-реверсирования

- Записываем второй байт в инверсивном порядке (2)



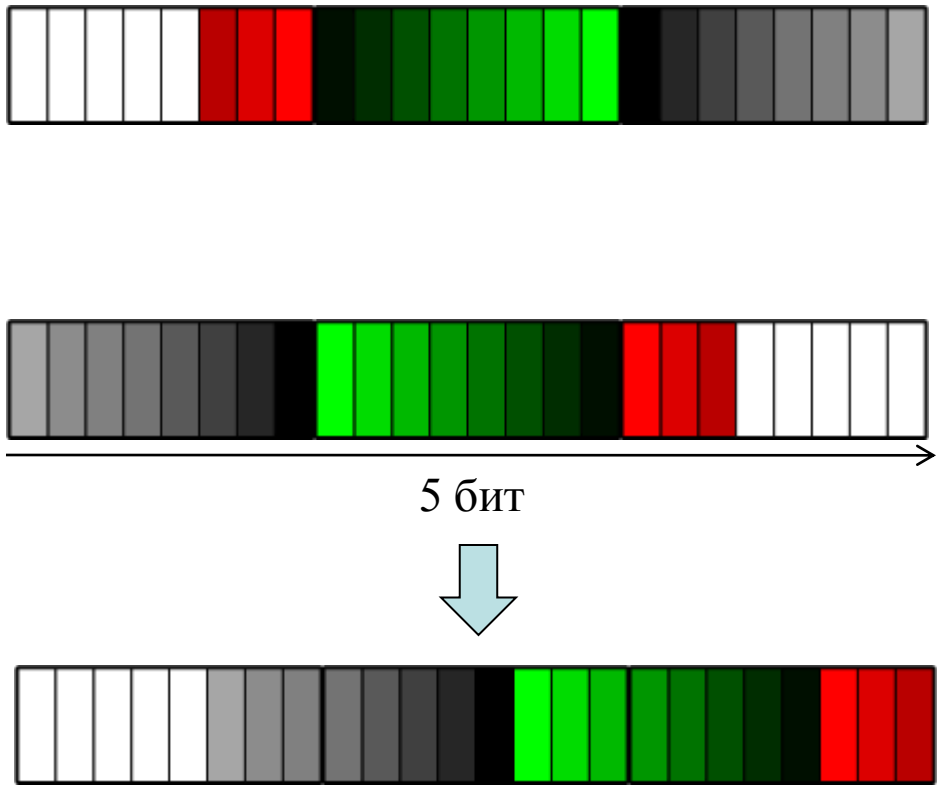
Пример бит-реверсирования

- Записываем третий байт в инверсивном порядке (3)



Пример бит-реверсирования

- Выполняем сдвиг на 5 бит вправо (4)

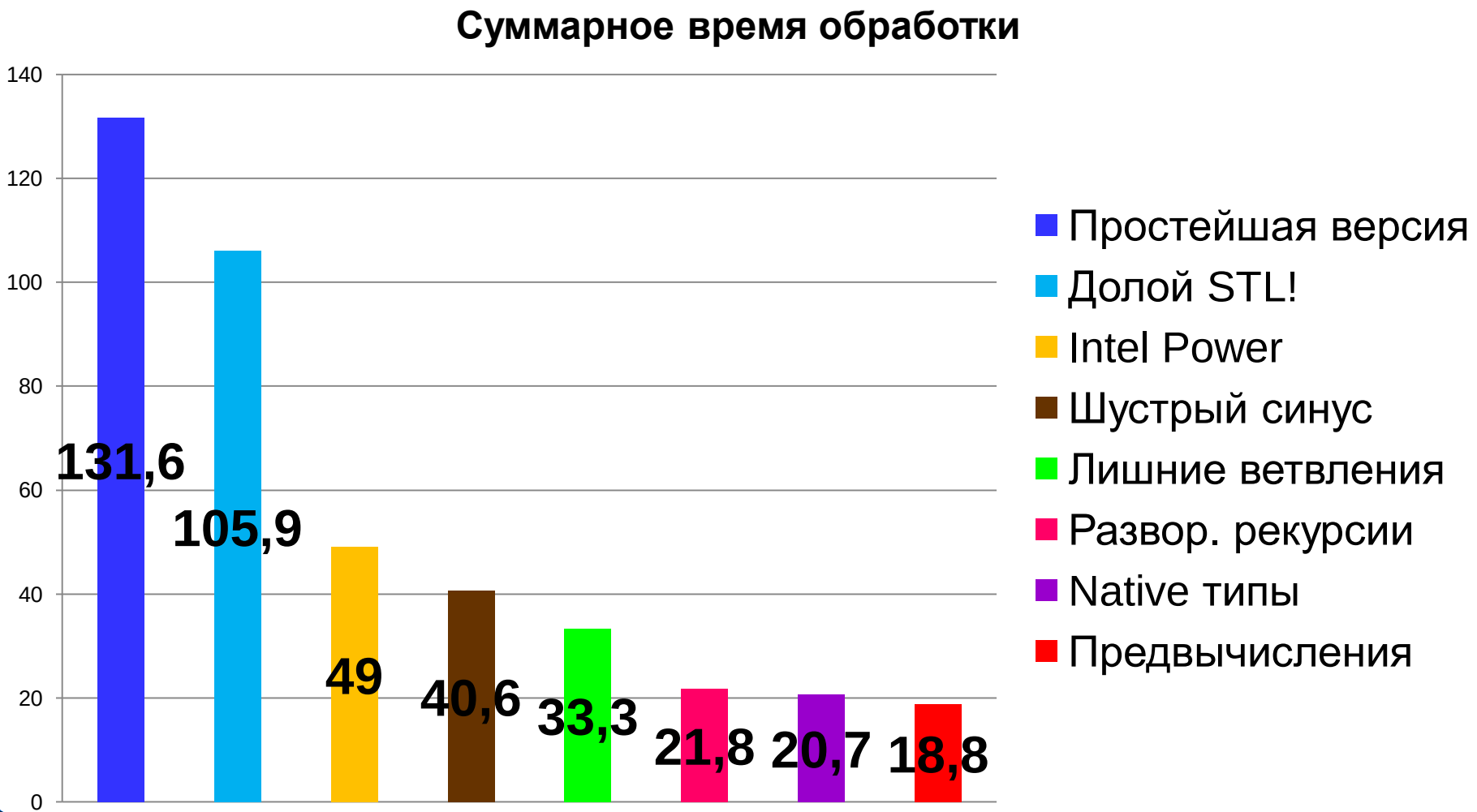


Оценка эффективности

- **Откройте проект “.\9. filter (precalculate)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

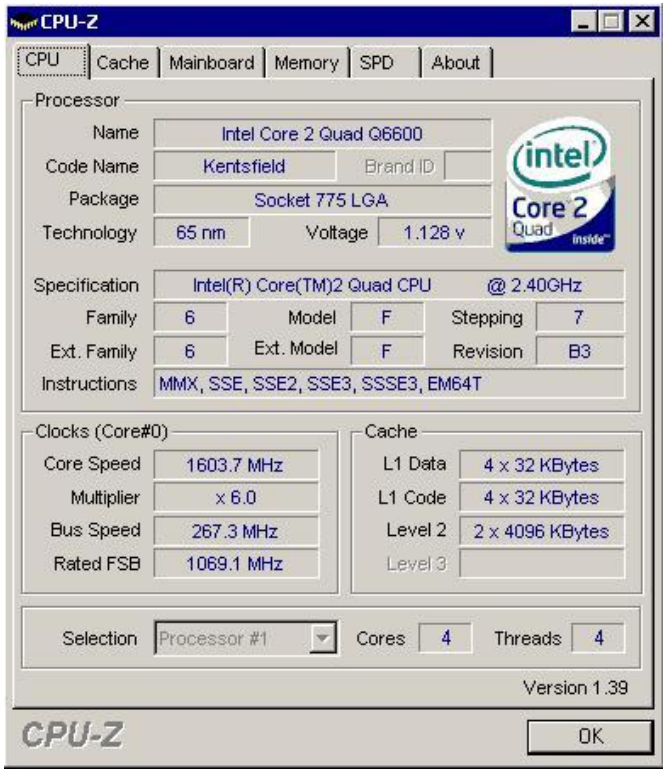
- Время, потраченное на обработку видео: 18.8 с

Прогресс оптимизации



Следующий шаг

- Многие (и наши в том числе) вычислительные системы имеют несколько ядер/процессоров (hyper threading на худой конец).



- Выполним распараллеливание функции, выполняющей вычисление БПФ, с использованием OpenMP.

10.

“— Мы бодры, веселы...

— Стоп, стоп! "Бодры" надо говорить бодрее. А "веселы"? ... “

Многоголовый монстр

Оценка эффективности

- **Откройте проект “.\10. filter (parallel)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

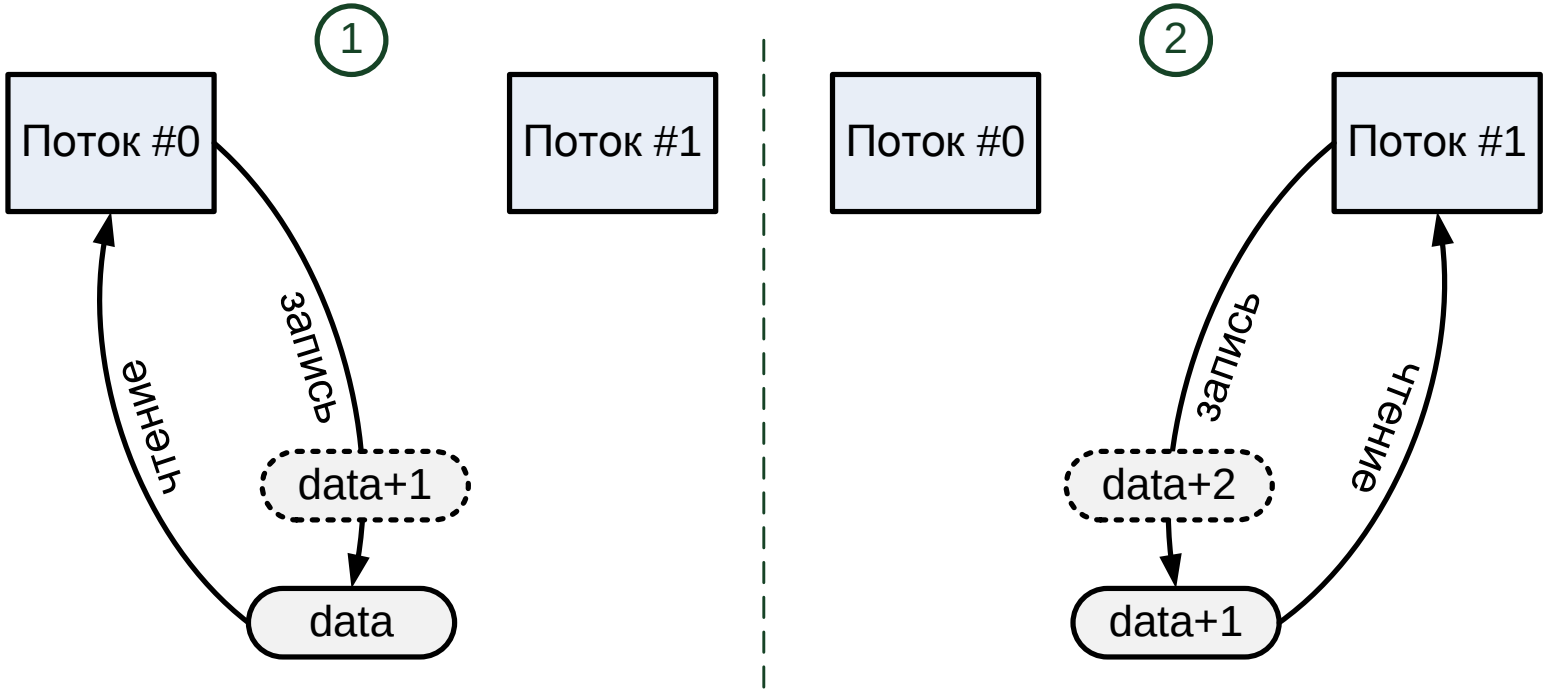
- С картинкой что-то не то!

“Гонки данных”

- Рассмотрим типичную ситуацию, в которой необходима синхронизация, 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 (оперативную память).

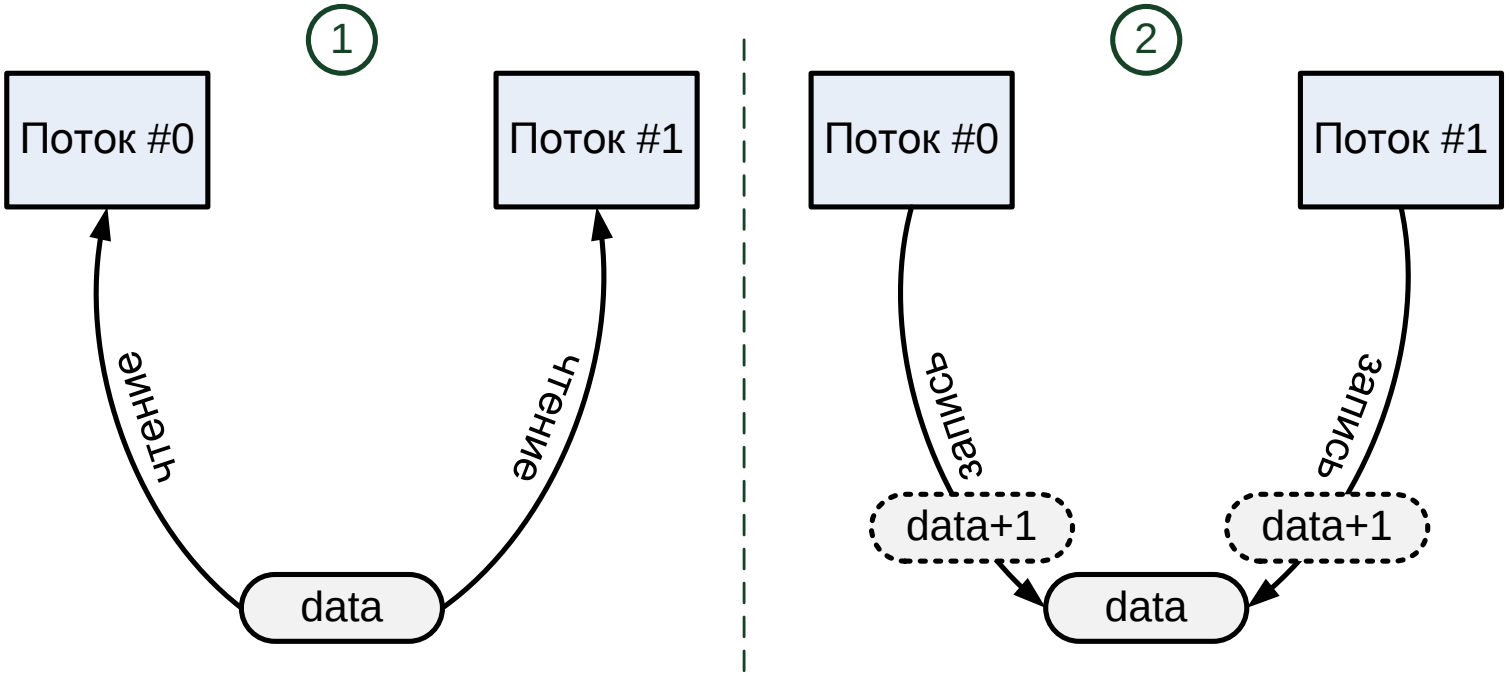
“Гонки данных”. Ожидаемая реализация

- 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**.



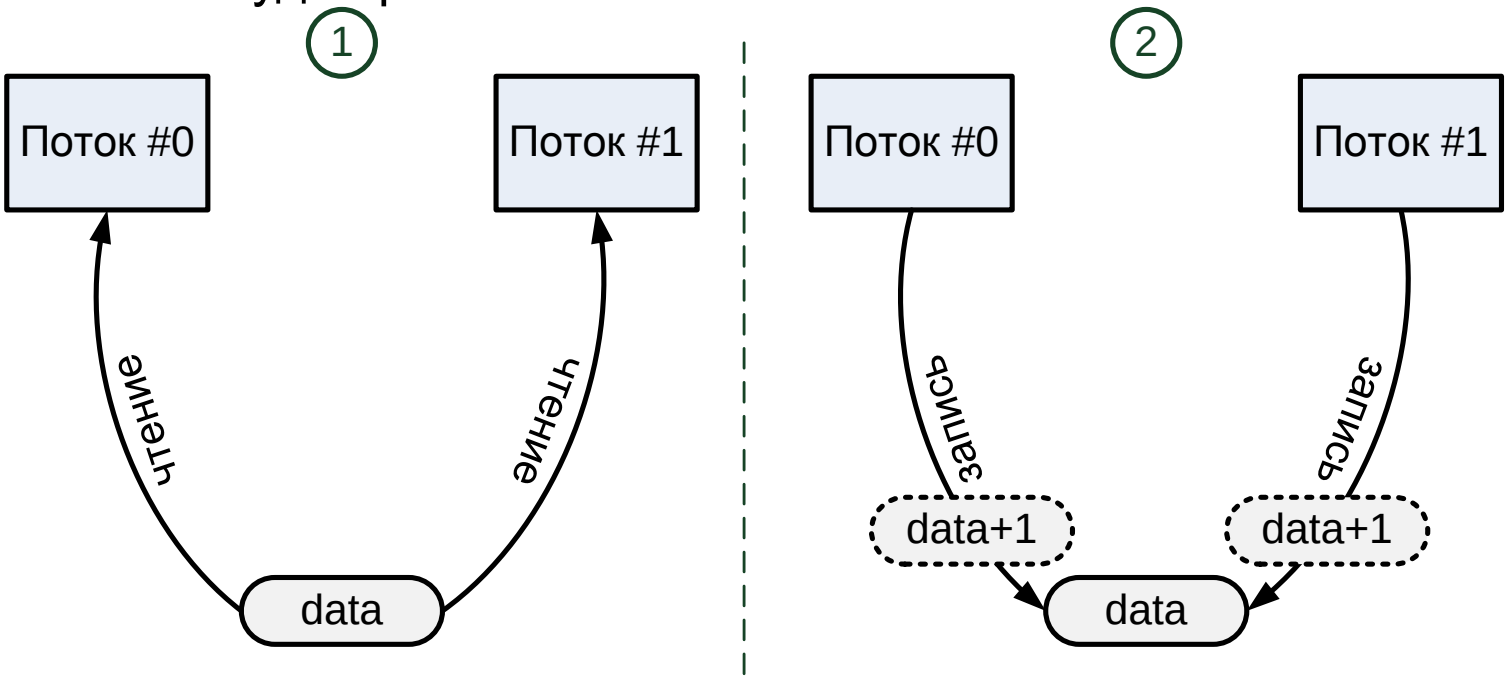
“Гонки данных”. Менее ожидаемая реализация (1)

- Поток 0 выполняет чтение значения переменной в регистр и инкремент этого регистра, и в этот же момент времени поток 1 выполняет чтение переменной **data**.
- Т.к. для каждого потока имеется свой набор регистров, то поток 0 продолжит выполнение и запишет в переменную значение **data+1**.



“Гонки данных”. Менее ожидаемая реализация (2)

- Поток 1 также выполнит инкремент регистра (значение переменной **data** было прочитано из оперативной памяти ранее 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**.



“Гонки данных”

- Обе реализации могут наблюдаться как на многоядерной (многопроцессорной) системе, так и на однопроцессорной, одноядерной.
- Таким образом, в зависимости от порядка выполнения команд результат работы приложения может меняться.

Ищем ошибки

- Вы когда-нибудь видели гонки данных? ☺
- **С помощью Intel Parallel Inspector попробуйте найти одну из типичных ошибок в параллельной программе (достаточно одного кадра):**
  - **“Debug” версия программы;**
  - **режим работы “Threading errors”;**
  - **режима анализа “Analysis level: Extreme”.**

Ошибки

- Гонка данных: потоки используют переменную *temI* параллельно для записи и чтения (fft.cpp:67, 70).
- Гонка данных: потоки используют переменную *uRtmp* параллельно для записи и чтения (fft.cpp:75, 77).

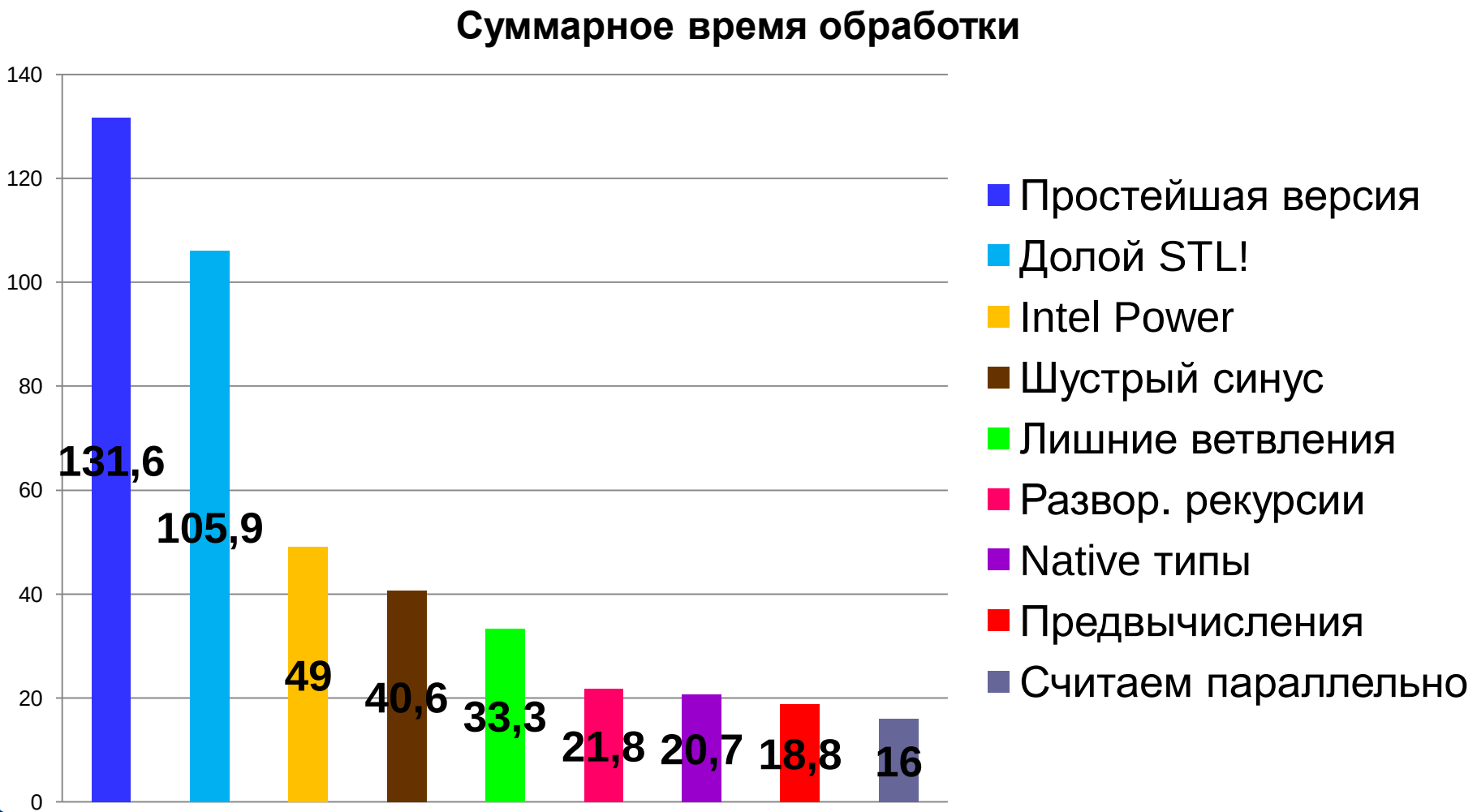
- Чтобы исправить ошибки необходимо:
  - либо воспользоваться директивой private:
    #pragma omp parallel for private(temI, uRtmp)
  - либо объявить переменные локально.

Оценка эффективности

- **Откройте проект “.\10. filter (parallel)”.**
- **Исправьте ошибки.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 16 с

Прогресс оптимизации

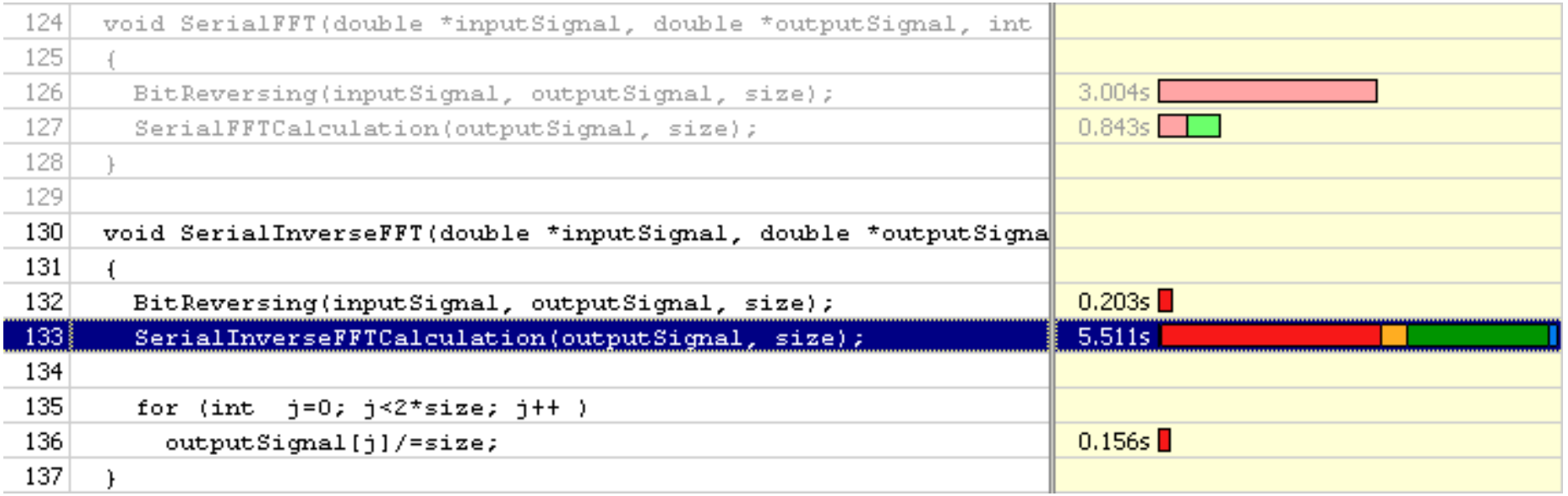


Анализ эффективности

- Оцените эффективность распараллеливания.
- **С помощью Intel Parallel Amplifier выполните оценку эффективности распараллеливания программы:**
  - **“Release” версия программы;**
  - **режим работы “Concurrency”.**

Эффективность параллельного кода

- Функции, вычисляющие БПФ, имеют не лучшую степень параллелизма:



Полная параллелизация

- Выполним полное распараллеливание функции выполняющей БПФ (на данный момент последняя итерация выполняется последовательно) и функции *ProcessFrame*.

11.

“Не бойся ложки, бойся вилки,

Один удар - 4 дырки”

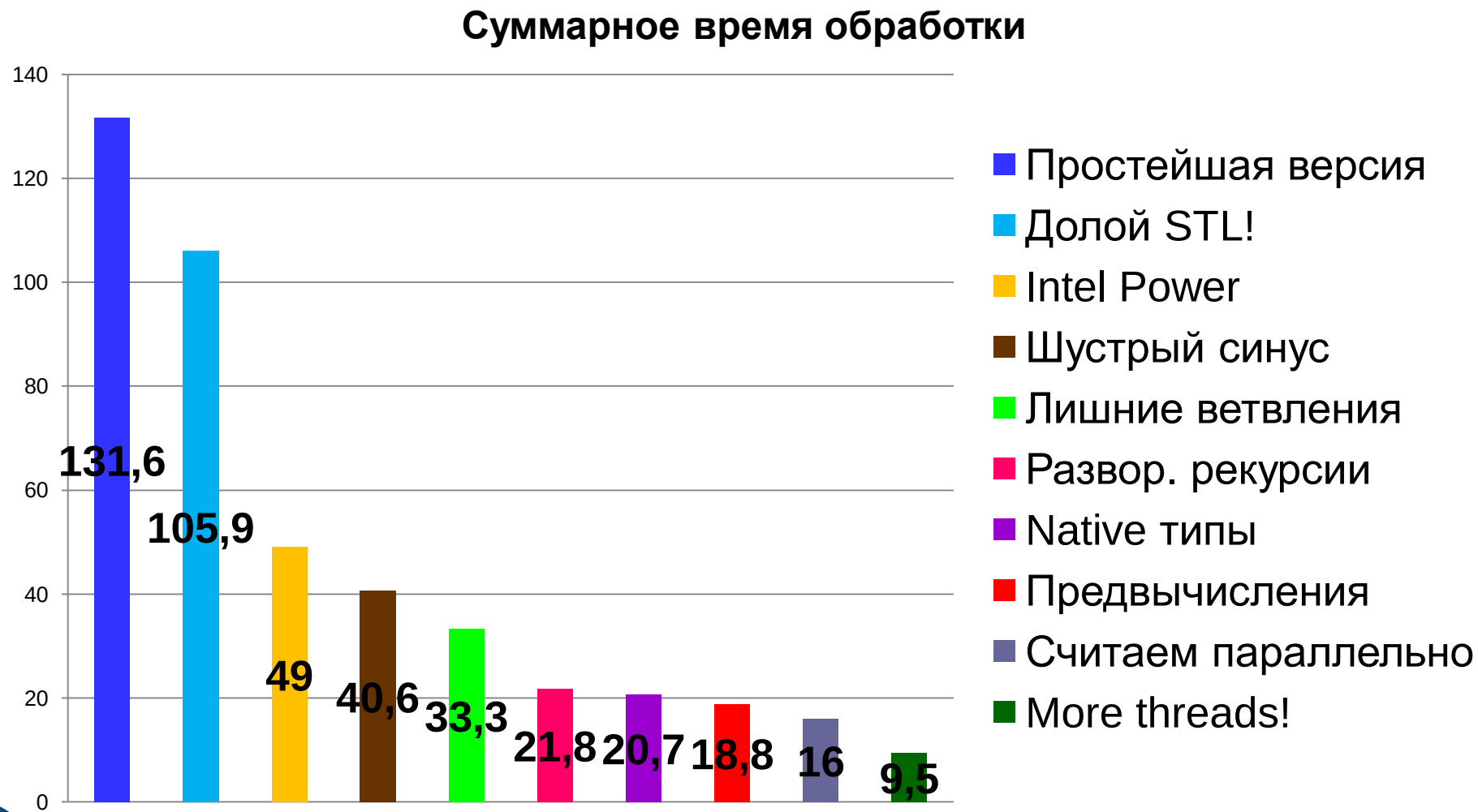
More! More threads!

Оценка эффективности

- **Откройте проект “.\11. filter (more threads!)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 9.5 с

Прогресс оптимизации



Дальнейшая оптимизация

- Далее выполним простейшие оптимизации в функции *ProcessFrame*:
  - заменим вычисление функции *pow* на соответствующее умножение;
  - вынесем *switch* из тела цикла;
  - выполним объединение циклов, выполняющих похожие действия;
  - будем выделять/освобождать динамическую память один раз.

12. Почему пингвины не летают!?

“Летающие” пингвины

Почему пингвины не летают

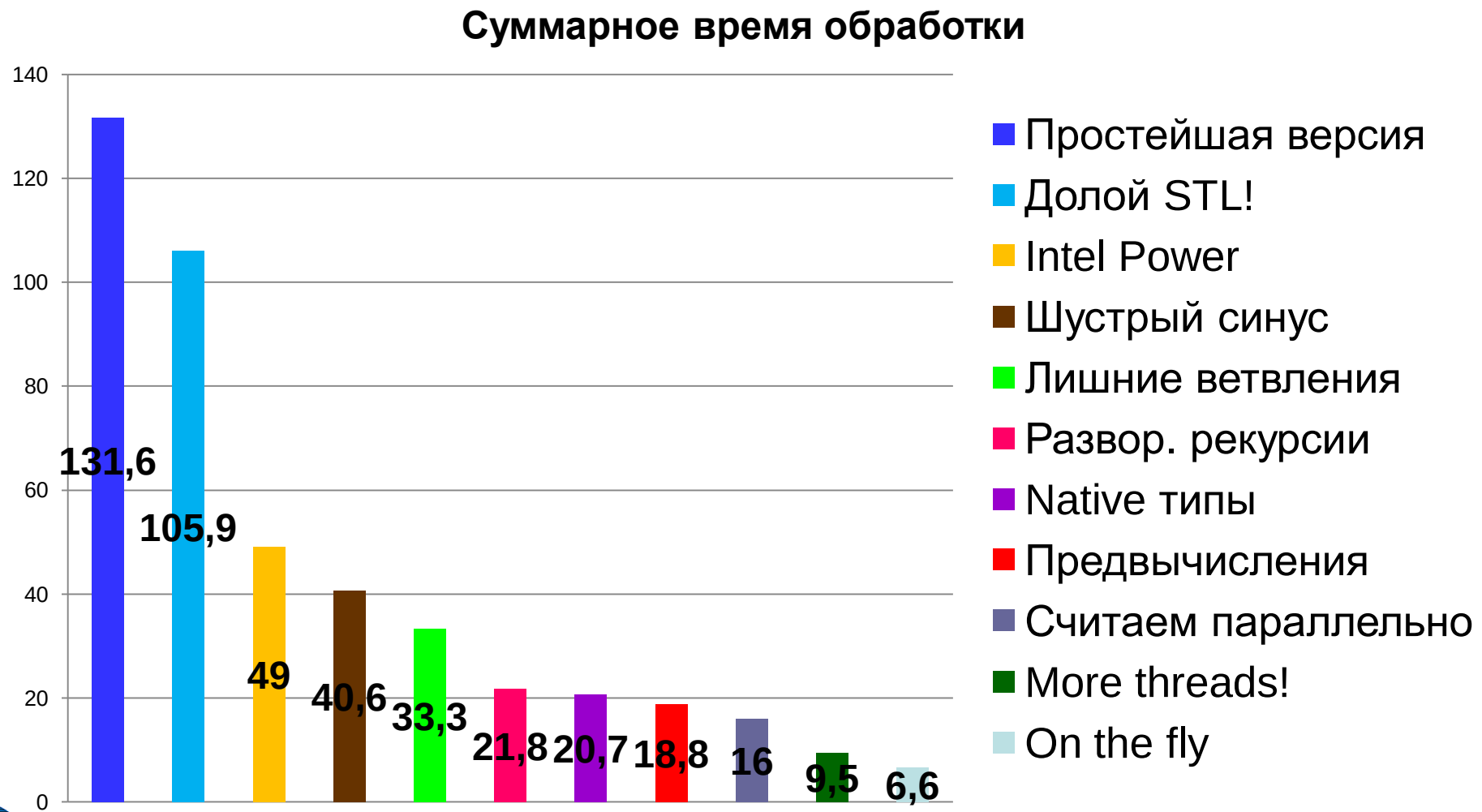
- Началось все очень давно. В те времена летали все, даже пингвины. Однажды маленький пингвинчик учился летать вместе с другими птенцами. Он подошел к обрыву и прыгнул. "Маши, маши!"-кричали все. Пингвинчик махал крылышками, но напрасно. Он все падал и падал. Даже когда оказался в воде, он продолжал махать крыльями (ему не очень хотелось потонуть). Он махал, махал и...ПОПЛЫЛ!
- С тех пор пингвины не летают, но очень хорошо плавают.

Оценка эффективности

- **Откройте проект “.\12. filter (on the fly)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Penguins_Short.avi –s –gh”

- Время, потраченное на обработку видео: 6.6 с

Прогресс оптимизации



Тестовые данные

- До настоящего момента все эксперименты проводились над одним и тем же видео файлом, поэтому вся оптимизация проводилась по сути для этого файла (размера картинки).
- Проведём эксперимент над файлом с большим размером картинки.

Оценка эффективности

- **Откройте проект “.\12. filter (on the fly)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Avatar2_High.avi –s –gh”

- Время, потраченное на обработку видео: 36.9 с

13. Читаемость образуемого кода граничит с безумием…

Размер имеет значение

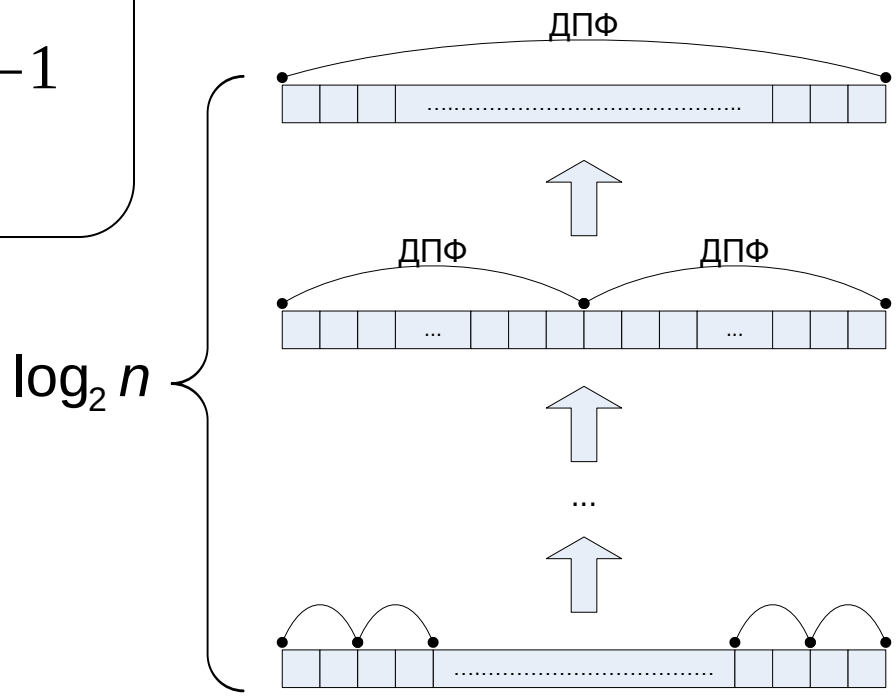
Быстрое преобразование Фурье

- Алгоритм основан на применении «бабочки» на каждом уровне рекурсии

$$y_p = a_p + b_p \cdot e^{-\frac{p}{n}2\pi i}$$

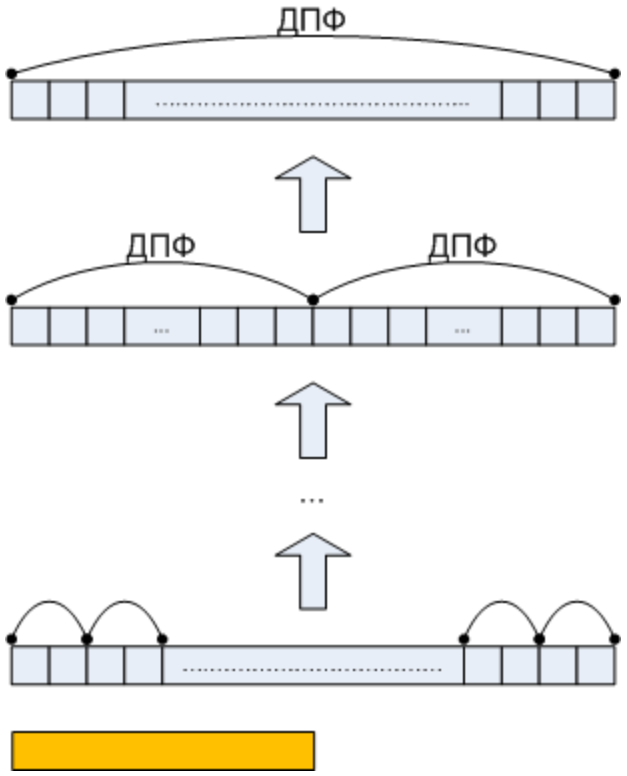
$$y_{n/2+p} = a_p - b_p \cdot e^{-\frac{p}{n}2\pi i}$$

$$p = \overline{0, \frac{n}{2} - 1}$$



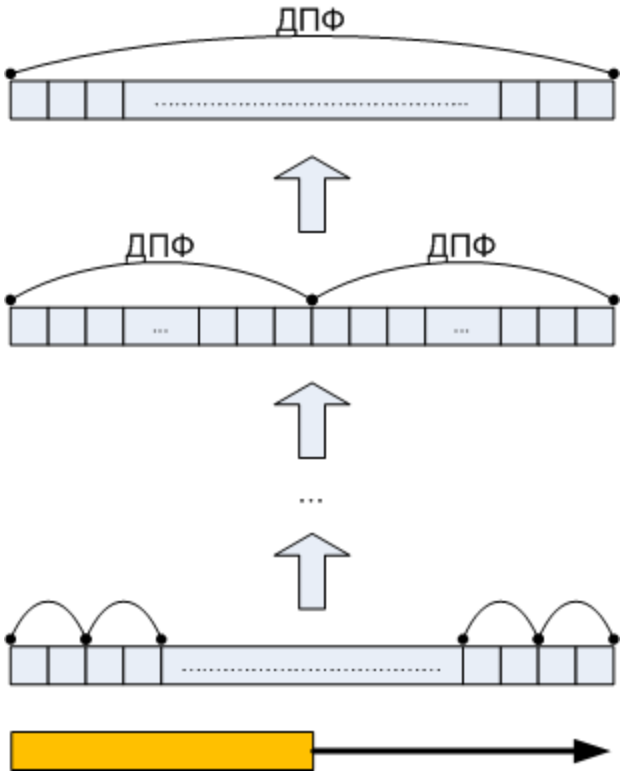
Эффективное использование памяти

- Пусть в процессоре используется кеш-память, позволяющая хранить половину обрабатываемых данных:



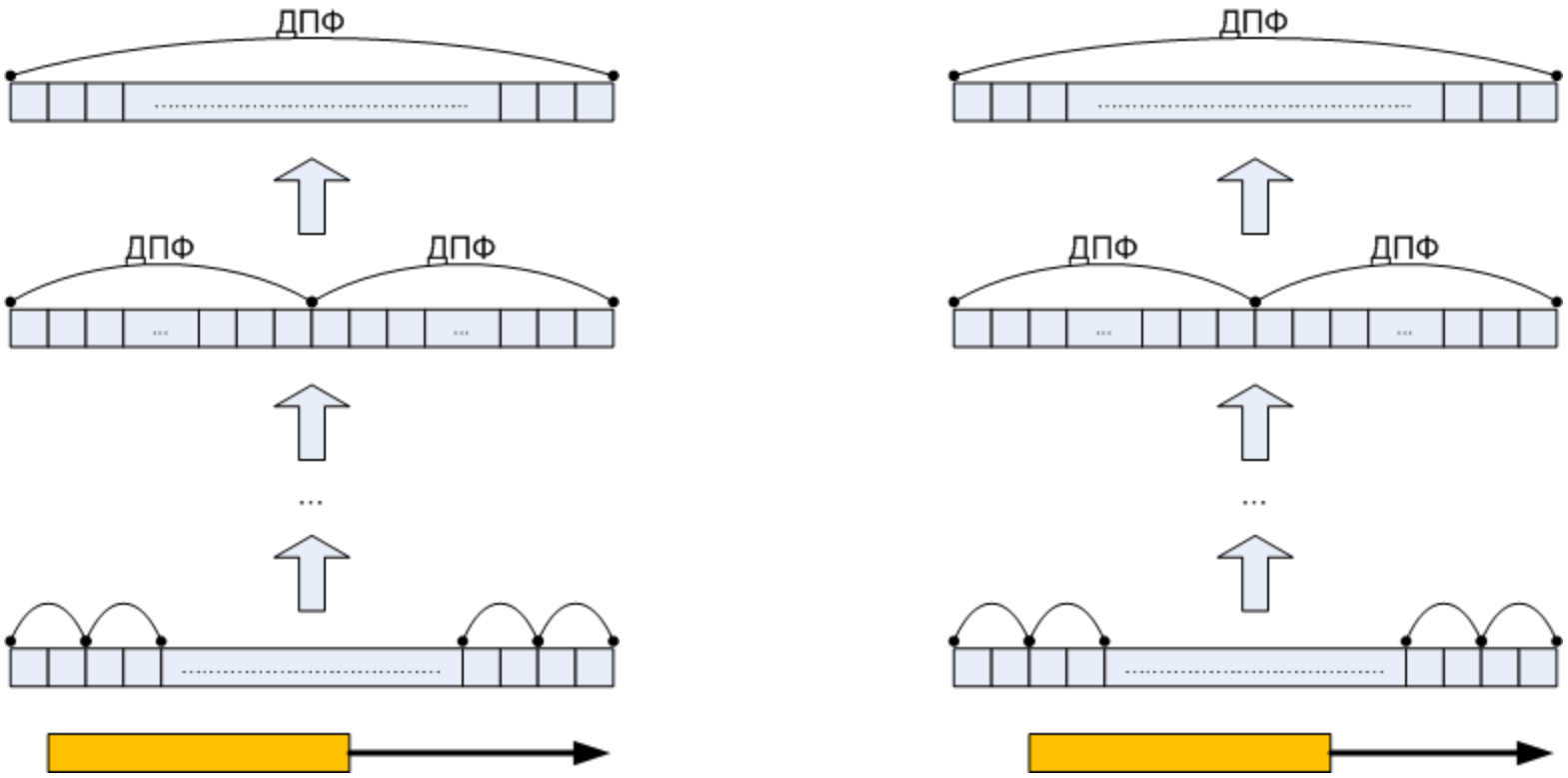
Эффективное использование памяти

- Сначала выполняется загрузка всех данных в кеш-память (1)



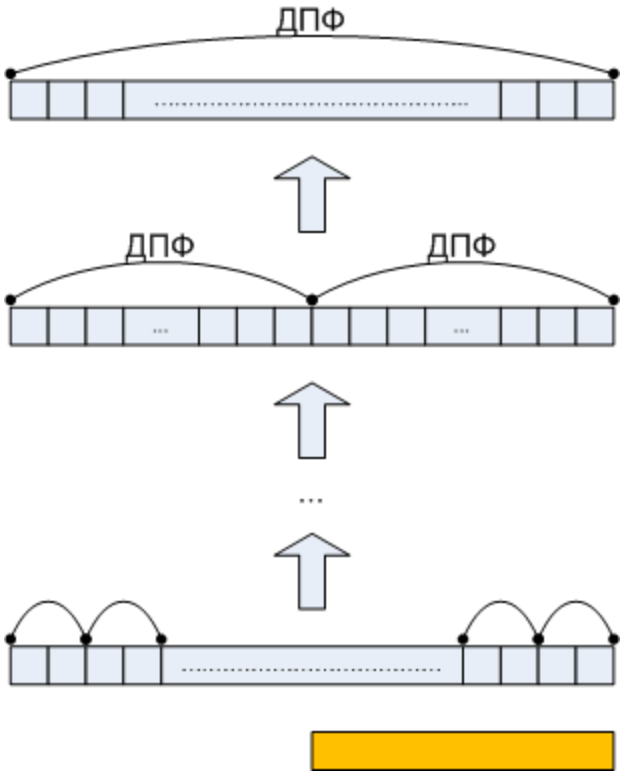
Эффективное использование памяти

- Далее происходит смещение загруженных данных (2)



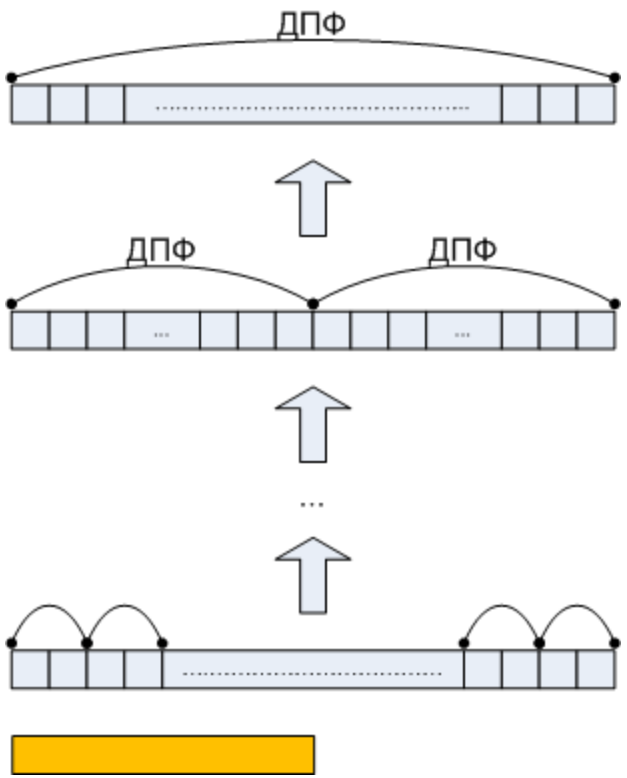
Эффективное использование памяти

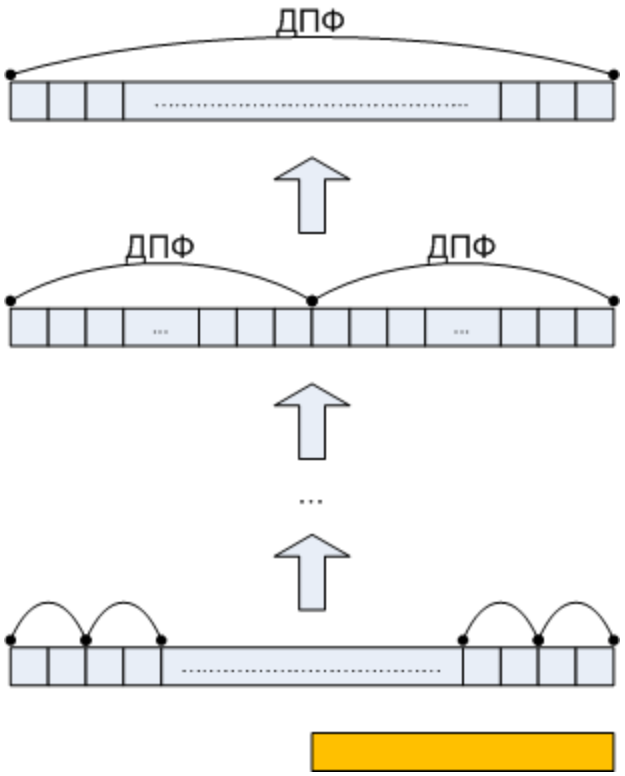
- В конце концов в кеш-памяти оказывается вторая часть данных (3)



Эффективное использование памяти

- Для более эффективного использования кеш-памяти необходимо повторно использовать данные, находящиеся в ней:





Оптимизация использования кеш-памяти

- В соответствии с описанной идей выполним оптимизацию, которая заключается в выполнении нескольких итераций БПФ над частью массива.

Оценка эффективности

- **Откройте проект “.\13. filter (big image)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Avatar2_High.avi –s –gh”
- **Подберите оптимальное значение макроса *BITS_CACHE***

- Время, потраченное на обработку видео: 36.3 с

Оценка эффективности

- **Откройте проект “.\12. filter (on the fly)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Avatar_High.avi –s –gh”

- Время, потраченное на обработку видео: 851.9 с

Оценка эффективности

- **Откройте проект “.\13. filter (big image)”.**
- **Соберите Release версию проекта.**
- **Запустите Release приложение:**
  “filter.exe ..\Videos\Avatar_High.avi –s –gh”

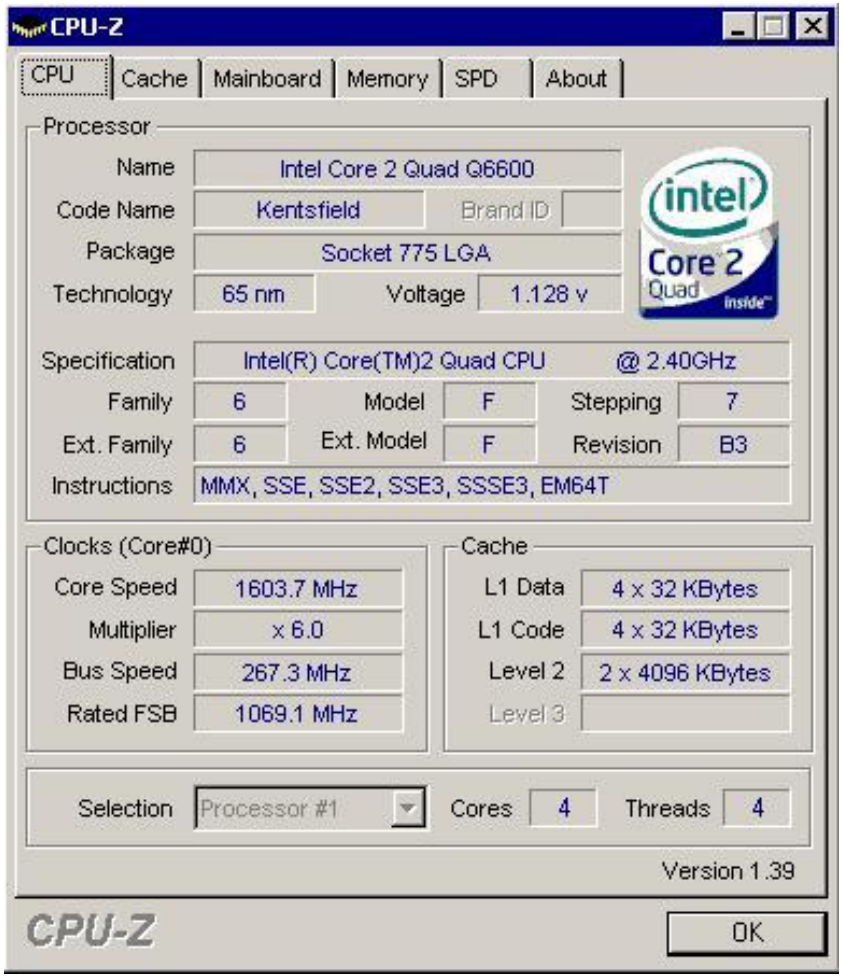
- Время, потраченное на обработку видео: 767.5 с

14. Пришло время узнать правду

Тестовая система

Аппаратная часть

- 4-хядерный процессор Intel Core 2



Программное обеспечение

- Microsoft Visual Studio 2005 SP1
  - Version 8.0.50727.762
  - Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.762 for 80x86
- Intel Parallel Studio
  - Intel Parallel Composer
    - Intel C++ Compiler for applications running on IA-32, Version 11.1.061
  - Intel Parallel Amplifier Update 2 (build 76442)
  - Intel Parallel Inspector Update 2 (build 75522)

Вот и сказочке конец

Вот и сказочке конец,

а кто слушал – молодец!