Final thesis for S-15., Michal Dobroczynski

2D FFT in Image Processing: measurements, implementation, parallelization and computer architecture

Keywords: Fourier Programming Linux

Synopsis:
Report describes implementation of two-dimensional Fast Fourier Transform in the PACO project. Apart from mathematical equations and sophisticated algorithms, the report tries to introduce and explain in details all other factors, that are usually skipped and may influence overall efficiency of the implementation. Since the PACO project is developed under Linux, the thesis contains important notes about Linux kernel and describes in details project organization including notes about GNU building process/tools. From the hardware point of view, this paper provides a comprehensive description of currently available technology and possible pitfalls associated with it – mostly in terms of real-time, large throughput Linux oriented software.
As the final conclusion, the report states that the best solution is not to implement 2D FFT routines from scratch, since complexity of the task and number of possible hardware combinations will outnumber every programmer.

I accept that the report is available at the library of the department.

Student: Michal Dobroczynski sign.
Supervisor: Norbert Krüger sign.
Company: Aalborg University Copenhagen
Co-ordinator: Henning Haugaard
Ext. examiner Stig Skelboe sign.
# Table of Contents

Preface.............................................................................................................................5

Part I - “In mathematics you don't understand things. You just get used to them.” ..........................................................................................................................7

Théorie analytique de la chaleur – where everything begins..................................7
Jean Baptiste Joseph Fourier.........................................................................................7
What Fourier Transform does – human language definition.................................7
Fourier transform: mathematical approach...............................................................9
  Conditions.........................................................................................................................9
  Dirac delta: impulse function......................................................................................10
  Periodic function.........................................................................................................11
  Exponential signal.......................................................................................................11
  Unit step.........................................................................................................................11
  Constant signal.............................................................................................................11
  Gate function................................................................................................................11

Properties of Fourier transformation..........................................................................11
DFT - Discrete Fourier Transform...............................................................................13
  Sampling theory.........................................................................................................13
  Quantization................................................................................................................14
  DFT: special case of Fourier Transform..................................................................14
  Properties of DFT........................................................................................................17

Convolution....................................................................................................................17
Two-dimensional Fourier Transform...........................................................................18
  Introduction..................................................................................................................18
  Definition of two-dimensional Fourier Transform..................................................19
  Inverse two-dimensional Fourier Transform............................................................19
  Discrete two-dimensional Fourier Transform............................................................20
  DFT – computational bottleneck, FFT – efficiency winner .......................................20

Fast Fourier Transform algorithm..............................................................................21
  Radix-2 Butterfly..........................................................................................................21
  DIT.................................................................................................................................21
  DIF.................................................................................................................................24
  Bit reversed...................................................................................................................24

Implementing radix-2 DIF.............................................................................................25
  Twiddle factors............................................................................................................25
  Radix-2 DIF FFT..........................................................................................................26
  Testing radix-2 DIF FFT..............................................................................................29
  Radix-2 in numbers: cost.............................................................................................30

Two-dimensional FFT: adapting 1D FFT.....................................................................30
  2D – computational madness.....................................................................................30

Programming considerations and overview of the method........................................31
  Transposition...............................................................................................................31
  Parallelization and 2D FFT.........................................................................................32
  Implementation of 2D FFT: reusing 1D FFT...............................................................33
  2D FFT: test..................................................................................................................35

Computations vs. methods............................................................................................36
  Algorithms for real data..............................................................................................37

Part II - “Not everything that can be counted counts, and not everything that
counts can be counted.”...............................................................................................38
Linux as destination operating system .............................................. 38
Introduction to Linux environment .................................................. 38
Operating system – organization ...................................................... 39
Linux means kernel ........................................................................ 40
Kernel – organization ...................................................................... 40
Scheduler and task management ...................................................... 41
Scheduler types .............................................................................. 42
I/O and CPU bound processes ........................................................ 43
Priorities – influencing scheduler's queue ........................................ 44
Quick note about I/O schedulers ...................................................... 45
Available I/O schedulers .................................................................. 45
Multi-threaded applications ............................................................. 46
Classic vs. modern processes .......................................................... 46
Why threads are so important? ......................................................... 48
SMP – more CPUs, more problems .................................................. 48
Dual-core technology ....................................................................... 49
Northbridge problem ....................................................................... 49
64 bits to happiness ....................................................................... 51
Why 64 is better than 32? ................................................................. 51
64 bits of problems ......................................................................... 52
64-bit computational research box .................................................... 53
Inside and close to CPU ................................................................. 54
Importance of this chapter ............................................................... 54
Stage 1: pipelining ........................................................................... 54
Number of pipeline stages .............................................................. 55
Floating-point unit .......................................................................... 56
Many processors – one floating point standard (IEEE 754). .............. 57
x87 ................................................................................................. 58
(Marketing) Extensions .................................................................... 59
Inside the memory hierarchy ........................................................... 61
Method ............................................................................................ 62
Sizes, levels, organization ............................................................... 63
One bus, different banks .................................................................. 65
Alternative solution – GPGPU .......................................................... 65
Introduction to GPU world ............................................................. 66
Streams ............................................................................................ 66
Specialization vs. optimization - datapath ....................................... 66
FFT on GPU ................................................................................... 67
Part III - “If you torture the data enough, it will confess.” .................. 68
Introduction .................................................................................... 68
Project formulation ......................................................................... 68
Milestone plan ................................................................................ 68
Requirements ................................................................................ 69
Problem solution ........................................................................... 70
BenchFFT ....................................................................................... 71
Florian ............................................................................................. 71
mic ................................................................................................. 74
qtux ............................................................................................... 77
volks ............................................................................................... 79
Results from FFTW's webpage ...................................................... 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project structure</td>
<td>85</td>
</tr>
<tr>
<td>How does FFTW work</td>
<td>86</td>
</tr>
<tr>
<td>Portability and adaptivity – let's plan!</td>
<td>86</td>
</tr>
<tr>
<td>Calculating DFT: overview</td>
<td>86</td>
</tr>
<tr>
<td>Implementation notes for multi-threaded environments</td>
<td>87</td>
</tr>
<tr>
<td>Wisdom: using memoization</td>
<td>87</td>
</tr>
<tr>
<td>Conclusions</td>
<td>87</td>
</tr>
<tr>
<td>MFFT</td>
<td>88</td>
</tr>
<tr>
<td>Development method</td>
<td>88</td>
</tr>
<tr>
<td>Configure.in</td>
<td>88</td>
</tr>
<tr>
<td>Makefile.am</td>
<td>90</td>
</tr>
<tr>
<td>Iterations and milestones</td>
<td>90</td>
</tr>
<tr>
<td>Source code documentation</td>
<td>91</td>
</tr>
<tr>
<td>Idea of mfft and key features</td>
<td>92</td>
</tr>
<tr>
<td>Wisdom manager</td>
<td>92</td>
</tr>
<tr>
<td>Time engines</td>
<td>93</td>
</tr>
<tr>
<td>Configuration file and runtime settings</td>
<td>93</td>
</tr>
<tr>
<td>In action: how to use the library</td>
<td>97</td>
</tr>
<tr>
<td>Running mfftest: mfft output</td>
<td>98</td>
</tr>
<tr>
<td>Integration into MoInS</td>
<td>101</td>
</tr>
<tr>
<td>Measurements</td>
<td>102</td>
</tr>
<tr>
<td>MoInS</td>
<td>103</td>
</tr>
<tr>
<td>MFFTEST</td>
<td>105</td>
</tr>
<tr>
<td>Florian</td>
<td>105</td>
</tr>
<tr>
<td>Mic</td>
<td>105</td>
</tr>
<tr>
<td>Qtux</td>
<td>106</td>
</tr>
<tr>
<td>Volks</td>
<td>107</td>
</tr>
<tr>
<td>In-place algorithms</td>
<td>107</td>
</tr>
<tr>
<td>In-place, out-of-place with fully preemptible kernel</td>
<td>107</td>
</tr>
<tr>
<td>2D FFT</td>
<td>108</td>
</tr>
<tr>
<td>SciMark</td>
<td>108</td>
</tr>
<tr>
<td>Florian</td>
<td>109</td>
</tr>
<tr>
<td>mic</td>
<td>109</td>
</tr>
<tr>
<td>qtux</td>
<td>109</td>
</tr>
<tr>
<td>volks</td>
<td>109</td>
</tr>
<tr>
<td>natalie</td>
<td>109</td>
</tr>
<tr>
<td>Cache benchmark</td>
<td>110</td>
</tr>
<tr>
<td>Florian</td>
<td>110</td>
</tr>
<tr>
<td>Mic</td>
<td>111</td>
</tr>
<tr>
<td>Qtux</td>
<td>111</td>
</tr>
<tr>
<td>Volks</td>
<td>112</td>
</tr>
<tr>
<td>FFT on GPU</td>
<td>112</td>
</tr>
<tr>
<td>Final conclusions</td>
<td>113</td>
</tr>
<tr>
<td>Appendices</td>
<td>115</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>115</td>
</tr>
<tr>
<td>References</td>
<td>116</td>
</tr>
</tbody>
</table>
Preface

Two years ago real-time filtering in frequency domain of images being larger than 1Megapixel was from the practical point of view impossible. Today it is, but it does not mean it is neat and easy. Especially now, when technique reaches its physical boundaries programmers even more have to focus on solving tasks in optimal and efficient way. The other thing is, that accessing all power lying in contemporary machines becomes more and more complex.

This document was divided intro three sections – each one corresponds to different field of science that is relevant to the “FFT problem”, as it will be called in later chapters. The first part introduces Fourier transform, its definition, applications and slowly derives other forms of the transform. Last chapters of the first part concentrate on the FFT algorithm. “Example isn't another way to teach, it is the only way to teach” (A. Einstein) – so that discussion about FFT calculations ends up in implementing both one- and two-dimensional routines in C. But good mathematical approach is just one of the ways to the success – one out of many.

If programming was only about implementing algorithms or mathematical formulas as they are, then it would be a really nice and easy job for programmers. Of course – it is not like that. Second part of the report tries to answer several questions that usually are not even mentioned in most books about FFT algorithms, but are key factors to accessing maximum performance. Please, have in mind that the second chapter does not provide ready answers, as each problem requires detailed analysis and several tests. Instead it gives a clue about things that happen inside CPUs, memories and so on. It also pin-points important facts about hardware and Linux organization and should definitely help in creating further development plans and organizing test cases. It also touches alternative solution – FFT on a GPU, which is a brand new approach that should be carefully tested.

Third part tries to make use of all the knowledge introduced in previous parts. It shows step by step, how the “FFT problem” was solved, including details about the FFTW library, which was the one picked up after benchmarking process.

Why then writing your own FFT routines? Why bother about all these facts, since anyway “we are not going to write/use even a single piece of our own FFT code”? Knowledge is power, and power can be used only by those who know how to master it.
The TUX

TUX's magnitude spectrum
Part I - “In mathematics you don't understand things. You just get used to them.”

Johann von Neumann

Théorie analytique de la chaleur – where everything begins

Jean Baptiste Joseph Fourier

Fourier was born in Auxerre, France on March 21, 1768 and is the one that is known for establishing the so called Fourier analysis.

His observations in the beginning contained errors\(^1\), nevertheless were a big breakthrough. In 1822 he published “Théorie analytique de la chaleur” in which he mentioned, that every function of a variable, both continuous or discontinuous can be expanded into an infinite series of sines and cosines. From the point when his theory was published one question remained unanswered: when one can say, that a function can be expanded by a series of sines and cosines? Fourier's work was continued firstly by Joseph Lagrange, who acknowledged mentioned theorem, but his work was still not satisfactory. Finally, Johann Dirichlet was the first one that gave satisfactory presentation of Fourier's transformation.

The biggest achievement of Fourier's work was the fact, that functions in the frequency domain contain exactly the same information as originals: this means, that people are able to perform analysis of a function from a different point of view. This of course resulted in enormous numbers of applications of Fourier's theorem.

What Fourier Transform does – human language definition

The Fourier transform is defined as an integral (in terms of Riemann's integral):

\[
X(\omega) = \int_{-\infty}^{\infty} x(t) e^{-j\omega t} \, dt
\]

Above integral defines an operation which causes, that to the function \(x(t)\) a function \(X(\omega)\) of real variable \(\omega\) is assigned. Analogous situation is when we define the inverse Fourier transform:

\(^1\) He did not establish conditions that have to be fulfilled by a function
This time a function \( X(\omega) \) is assigned a function \( x(t) \). If we have a function \( x(t) \) and its Fourier transform, then we can say that we have a transform pair, which in symbolic language is defined as follows:

\[
x(t) \Leftrightarrow X(\omega)
\]

Before the Fourier transform theorem can be used, a set of conditions has to be established. They can be divided into two categories: the first one sets conditions that function \( x(t) \) has to meet in order to have a transform pair, and the other one starts the discussion about mutual properties of the transform – it means, that it tries to define conditions at which it is possible to apply inverse transform and obtain original function, \( x(t) \).

Above definitions can be considered by mathematicians and people already familiar with Fourier transform as “written in human language”, but what in fact Fourier transform does? What is the result of applying it and what does it really mean, that we can analyze a function from a different point of view?

Without tedious calculations above definitions do not say anything about possible outcome, thus frankly speaking, Fourier transform decomposes original function \( x(t) \) into its basic components – as it was mentioned earlier in this text – to a sum of sines and cosines. Even simpler approach is to say, that by applying Fourier transform to the function (signal) \( x(t) \) we are able to see different frequencies present in \( x(t) \). Each basic component carries two pieces of information: amplitude (gain) and frequency. In most applications we will be interested in altering amplitude of elements having certain frequency/frequencies. Of course, previously mentioned feature that for all the time we are working on the same function holds true. \( x(t) \) is the same, different is the way of presenting it.

Our function becomes a sum of infinitely many terms composed of sine and cosine functions, each of them having different frequency, which is a multiplication of the so called fundamental frequency. In music the fundamental frequency can be referred to as the lowest waveform present in a signal that determines its pitch. All higher frequencies, which are multiplies of the fundamental one, are called harmonics. All higher frequency signals in this case are responsible for giving the so called “color” to the instrument. That is why different instruments when play the same tone give different musical texture – just because they have different acoustic spectrum.

Why are we talking about sines and cosines as expansion functions? In fact, any set of functions can be used as long as all functions in that set are orthogonal. This property can be defined in the following way:

\[
fundamental frequency : \omega \\
harmonics: 2\omega, 3\omega, 4\omega \ldots \infty \omega
\]

\[\omega = 2\pi f, \text{so that} \omega \propto f\]

the fundamental one, are called harmonics. All higher frequency signals in this case are responsible for giving the so called “color” to the instrument. That is why different instruments when play the same tone give different musical texture – just because they have different acoustic spectrum.
Orthogonality assures, that elements of a sequence will not interfere with each other, and additionally it brings in a lot of other applications and possibilities, like for Parseval’s theorem which says, that energy of a signal that is composed of orthogonal elements is equal to the sum of energies of signal components.

Exponential series is orthogonal and definitions presented with use of exponential series are believed to be the most readable ones. Of course, as it was stated above, all these definitions could be also presented by means of:

- trigonometric series
- Legendre polynomials
- Walsh’s functions
- Haar's functions

**Fourier transform: mathematical approach**

**Conditions**

To be able to obtain a Fourier transform pair, function \( x(t) \) hast to fulfill the following conditions:

- \( x(t) \) has to be absolutely integrable, meaning that \( \int_{-\infty}^{\infty} |x(t)| dt < \infty \). This condition is considered to be a sufficient condition.

- if \( x(t) = \beta(t) \sin(2\pi ft + \alpha), f = \text{const}, \alpha = \text{const} \) and \( \beta(t+k) < \beta(t) \) and for \( |t| > \lambda > 0 \) the function \( \frac{x(t)}{t} \) is absolutely integrable, then \( X(\omega) \) exists and satisfies the inverse Fourier transform. Please have in mind the fact, that \( \omega = 2\pi f \). This condition has an application when it comes to the so-called sampling function very often denoted as \( Sa(t) \) [ \( Sa(at) = \frac{\sin(at)}{at} \) ]. It is purely visible that this function is not absolutely integrable. Nevertheless, Fourier transform for a \( h(t) = 2\pi f_0 \sin(2\pi f_0 t) \) exists and is given by the following formula:

\[
H(f) = \int_{-\infty}^{\infty} 2\pi f_0 \frac{\sin(2\pi f_0 t)}{2\pi f_0 t} e^{-j2\pi ft} dt = \frac{A}{\pi} \int_{-\infty}^{\infty} \frac{\sin(2\pi f_0 t) \cos(2\pi ft)}{t} dt
\]

Integrand function is odd, thus imaginary part will be equal to zero. After substituting \( \sin(x)\cos(y) = \frac{1}{2} [\sin(x+y) + \sin(x-y)] \) to obtained integral:
\[ H(f) = A(f_0 + f) \int_{-\infty}^{\infty} \frac{\sin(2\pi t(f_0 + f))}{2\pi t(f_0 + f)} dt + A(f_0 - f) \int_{-\infty}^{\infty} \frac{\sin(2\pi t(f_0 - f))}{2\pi t(f_0 - f)} dt. \]

From the standard integrals table we know that \[ \int_{-\infty}^{\infty} \frac{\sin(2\pi ax)}{2\pi ax} dx = \frac{1}{2|a|}, \]
so that then our solution is as follows:
\[
H(f) = A \quad |f| < f_0
\]
\[
H(f) = A/2 \quad f = \pm f_0.
\]
\[
H(f) = 0 \quad |f| > f_0.
\]
For this condition to be true now it is enough to show, that it is possible to calculate inverse Fourier transform of the given example. Then of course, we obtain transform pair.

\[
h(t) = \int_{-f_0}^{f_0} A e^{j2\pi ft} df = A \int_{-f_0}^{f_0} \cos(2\pi ft) df = ... = 2Af_0 \frac{\sin(2\pi f_0 t)}{2\pi f_0 t}.
\]

Finally we can write that h(t) has a transform H(f) for \[ |f| < f_0. \]

These two conditions are sufficient conditions for existence of Fourier transform. By means of them we can establish a set of functions that can be represented by a curve of finite height in any finite time interval. One can easily spot, that this definition lacks singular (impulse) functions.

It is possible to obtain Fourier transforms for periodic or impulse functions, but if and only if the distribution theory is applied. Why are they so important? Developing transforms of singular functions can be used further for developing other transform pairs and can significantly simplify further calculations.

**Dirac delta: impulse function**

Dirac delta function or sometimes referred to as a unit impulse function is defined by the following set of equations:
\[
\delta(x) = \begin{cases} 
\infty, & x = 0 \\
0, & 0 < x < \infty 
\end{cases}
\]

Its integral from minus to plus infinity is equal to 1. Dirac delta is a very useful approximation for tall and narrow spike function, which is referred to as an impulse. As mentioned above, precise treatment of the Dirac delta function requires distribution theory.

The Fourier transform of Dirac delta functions is as follows:
\[
\int_{-\infty}^{\infty} A \delta(t) e^{-j\omega t} dt = A e^{0} = A
\]
And the inverse transform:
\[
\int_{-\infty}^{\infty} K e^{j\omega t} d\omega = \int_{-\infty}^{\infty} [K \cos(\omega t)] d\omega + j \int_{-\infty}^{\infty} [K \sin(\omega t)] d\omega = ... = K \int_{-\infty}^{\infty} \cos(\omega t) d\omega = K \delta(t)
\]
This yields of course that we have established a transform pair for Dirac delta function.

**Periodic function**

It would be also very helpful to have a transform pair defined for periodic functions. Let us consider the following example:

\[ h(t) = \sin(\omega_0 \cdot t) \Leftrightarrow H(j \omega) = j \pi \left[ \delta(\omega + \omega_0) - \delta(\omega - \omega_0) \right] \]

Exactly the same result we would obtain in case of cosine function.

**Exponential signal**

Let us consider an exponential signal of the following form:

\[ h(t) = e^{-\alpha t} \Leftrightarrow H(j \omega) = \frac{1}{\sqrt{\alpha^2 + \omega^2}} e^{-j \arctg(\frac{\omega}{\alpha})} \]

**Unit step**

This function is very often used especially in electrical engineering and automatics.

\[ 1(t) \Leftrightarrow \pi \delta(\omega) + \frac{1}{j \omega} \]

**Constant signal**

\[ A \Leftrightarrow 2 \pi A \delta(\omega) \]

**Gate function**

\[ \sigma_{\tau}(t) = A, \quad |t| \leq \frac{\tau}{2} \Leftrightarrow A \sin(\frac{\omega \tau}{2}) \]

\[ \sigma_{\tau}(t) = 0, \quad |t| > \frac{\tau}{2} \]

**Properties of Fourier transformation**

Fourier transform comes with a set of basic properties. These basics can be further applied to more complicated calculations and derivations.

- **linearity** - \[ x(t) + y(t) \Leftrightarrow X(\omega) + Y(\omega) \]

- **symmetry** - \[ H(t) \Leftrightarrow h(\omega) \]

- **time scaling** - \[ h(kt) \Leftrightarrow \frac{1}{|k|} H\left(\frac{j}{k}\right) \] . Time scale expansion corresponds directly to the frequency scale compression. In other words, while the time scale is expanding the frequency scale contracts. In order for the equation to hold true, the amplitude must increase, so that area under the function remains unchanged. This property is used in radar and antenna theory.
- frequency scaling - \( \frac{1}{k} h\left(\frac{t}{k}\right) \Leftrightarrow H(kf) \). Frequency scaling is analogous to the time scaling property, that is, when frequency scale is expanded, then the amplitude of the time function increases (so that area under both functions is equal).

- time shifting - \( h(t-t_0) \Leftrightarrow H(f) e^{j2\pi f_0 t} \). Crucial thing is to note, that time shifting affects only phase – amplitude remains unchanged.

- frequency shifting - \( h(t) e^{j2\pi f_0} \Leftrightarrow H(f-f_0) \). Function (signal) is shifted by \( f_0 \) Hz in frequency. This property is a fundamental property used in modulation of signals.

As it is known, each signal \( x(t) \) can be represented as sum of two signals

\[
x(t) = x_o(t) + x_e(t)
\]

which are odd and even parts of \( x(t) \). The Fourier transform of \( x(t) \) is as follows:

\[
X(\omega) = \int_{-\infty}^{\infty} x_o(t) \cos(\omega t) dt - j \int_{-\infty}^{\infty} x_o(t) \sin(\omega t) dt = P(\omega) + jQ(\omega)
\]

Equation presented above yields several features of spectral characteristics. If \( x(t) \) is function of real numbers, then its real spectrum is even function and complex spectrum is odd:

\[
P(\omega) = P(-\omega) \quad Q(\omega) = -Q(-\omega)
\]

For complex functions situation is a bit different. To simplify analysis we will establish so-called general representation of complex function (which can be also referred to as a signal):

\[
x(t) = \mathbb{R}[x_e(t)] + j\mathbb{I}[x_e(t)] + \mathbb{R}[x_o(t)] + j\mathbb{I}[x_o(t)]
\]

Equation presented above yields the following properties:

<table>
<thead>
<tr>
<th>Function (signal)</th>
<th>Spectrum</th>
</tr>
</thead>
<tbody>
<tr>
<td>even</td>
<td>even</td>
</tr>
<tr>
<td>odd</td>
<td>odd</td>
</tr>
<tr>
<td>real and even</td>
<td>real and even</td>
</tr>
<tr>
<td>real and odd</td>
<td>complex and odd</td>
</tr>
<tr>
<td>complex and even</td>
<td>complex and even</td>
</tr>
<tr>
<td>complex and odd</td>
<td>real and odd</td>
</tr>
<tr>
<td>hermitian</td>
<td>real</td>
</tr>
<tr>
<td>anti hermitian</td>
<td>complex</td>
</tr>
<tr>
<td>real</td>
<td>hermitian</td>
</tr>
</tbody>
</table>
\[
x(t) = \Re[x_e(t)] + j\Im[x_e(t)] + \Re[x_o(t)] + j\Im[x_o(t)] \\
x(t) = \Re[x_e(t)] + j\Im[x_e(t)] + \Im[x_o(t)] + j\Re[x_o(t)] \\
\uparrow \\
\downarrow \\
\Re_e \quad \Im_e \quad \Re_o \quad \Im_o
\]

Matrix above offers an easy and fast way of determining the spectrum of a signal. Upper row presents the function whereas bottom row is the spectrum².

**DFT - Discrete Fourier Transform**

**Sampling theory**

Result of sampling of a continuous signal is discrete signal, which is a representation of the continuous signal but in discrete domain. Usually sampling is performed in regular time intervals, which simplifies further operations on the discrete signal. One should imagine a process of taking a sample of the input signal at regular time intervals – these samples when considered to be a sequence form discrete signal. It is important to notice, that discrete signal is a sequence – because when a signal is sampled by an ideal sampler, it means, continuous signal is multiplied by a Dirac comb, then the result is again a continuous signal!

One has to be aware of the fact, that sampling introduces errors, as the discrete sequence does not contain any information about the signal between sample periods. Of course, value of the error can be influenced and minimized so that it will not affect further analysis of the signal. This leads to another, fundamental principle of signal processing: Nyquist – Shannon theorem. The more points are in the discrete sequence, the higher sample frequency and as a result the smaller error sneaks into the data. According to the theory, sampling frequency has to be twice as big as the highest frequency present in the signal.

\[\text{theory} : \omega_{\text{sample}} > 2 \cdot \omega_{\text{max}}\]
\[\text{practice} : \omega_{\text{sample}} > 2.5 \cdot \omega_{\text{max}}\]

In practice, this relationship should be equal to 2.5 or more. Why is it so important to sample signals with proper sampling frequency? If the signal is sampled with frequency that is less than the frequency of the sampled signal, then of course one will obtain false results. Phenomenon responsible for this is called aliasing. In frequency domain it means, that signals spectra are overlapping (spectrum is periodic!) and there is no way that some of the frequencies present in the signal can be distinguished. In time domain it means, that one would think that the signal has much smaller frequency.

---

² Idea of presenting this property was taken from “Podstawy teorii sygnałów”, Jerzy Szabatin
In fact it is possible to sample signals that have frequency higher than the sampling frequency, but only when one knows signal’s bandwidth and band limits.

### Quantization

The main idea of sampling is that we would like continuous signals to be analyzed and processed by discrete-time machines – computers, DSPs and others. Discrete signal is not a digital signal. There is one more step required which is called quantization. A sequence of discrete values has to be quantized and then in turn, it becomes a digital signal. In other words it means, that quantization is a method of approximating values. Approximation levels are said to be fixed and relatively small (CDDA discs can be used as an example: audio data is sampled at 44.1kHz, but is quantized with 16 bits, meaning, that there are $2^{16}$ levels for representing the output). Sampling and quantization are fundamental principles used by DACs and ADCs (digital-to-analog and analog-to-digital converters).

### DFT: special case of Fourier Transform

As stated in the headline, DFT can be treated as a special case of Fourier transform. It is possible to derive DFT itself, but in most cases it is referred to its continuous equivalent.

As it was stated before, this time we are working on a sampled signal. Idea of DFT is to analyze frequencies contained in these samples. Of course, once we have digital data there are thousands of other possibilities and applications, like convolutions, solving partial differential equations and similar.

Let us consider a series of samples that are complex numbers (note: data can
have different forms, i.e. samples can be purely real, purely imaginary or combination of both). DFT can be obtained from the following formula:

\[ X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N} kn}, k=0, ..., N-1 \]

Result of computations is sequence of N complex values. Discrete Inverse Fourier Transform is defined as follows:

\[ x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j\frac{2\pi}{N} kn}, n=0, ..., N-1 \]

Difference between forward and inverse transforms is mainly in the sign of the exponent (signs are very often also treated as conventions; the most important thing is to keep the relation, that for the inverse transform sign of the exponent should be opposite). Scale factor\(^3\) is mostly a convention and might differ among other definitions. It is very important to realize the fact, that first element of the transformed series is the so called DC component that is more commonly known as average of the input series:

\[ X_0 = \sum_{k=0}^{N-1} x_n e^{-j\frac{2\pi}{N} kn} = \sum_{k=0}^{N-1} x_n e^0 = \sum_{k=0}^{N-1} x_n \]

\[ e^0 = 1 \]

Mentioned above scale factor is a convention, thus definitions could be also rewritten in the following form:

\[ X_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N} kn}, k=0, ..., N-1 \]

and the inverse:

\[ x_n = \sum_{k=0}^{N-1} X_k e^{j\frac{2\pi}{N} kn}, n=0, ..., N-1 \]

So that now, the first element of transformed sequence will be average of input:

\[ X_0 = \frac{1}{N} \sum_{k=0}^{N-1} x_n \]

As it was mentioned above, DFT approximates continuous Fourier Transform. Quality of this approximation depends on a function that is being analyzed and number of samples present in a sequence. Once again it should be stressed out, that DFT will not look exactly like continuous FT as data present in DFT is truncated (that is why DFT is sometimes called Finite Fourier Transform). There are different ways of truncating data samples – the most logical one is when we have a periodic signal \( s(t) \) that is sampled (sampling frequency high enough so that aliasing does not occur) and number of samples contains one full period. Of course, DFTs calculated from the same signal but truncated in different way, where for example number of samples does not resemble integer number of periods, will result in spectrum looking considerably different. This effect will be visible as a periodic function with visible

---

\(^3\) In most implementations scale factor is the length of input vector
discontinuities. Below there are two sets of images: first set presents proper interval and the second one depicts what happens when the interval is too short.

Below we have aperiodic function (part of sine):
NOTE: The highest frequency present in a signal is called the Nyquist frequency, which in turn determines the minimum sampling frequency according to the Nyquist – Shannon (or Shannon – Kotielnikov) theorem.

**Properties of DFT**
Because DFT is a special case of Fourier Transform, it has the same properties as continuous form.

**Convolution**
When there is discussion about Fourier Transform usually convolution is also included. Why – this will be revealed in a moment. In time domain convolution of two functions is defined as:
\[(f * g)(t) = \int f(\tau)g(t-\tau) d\tau\]

and in discrete domain:

\[(f * g)(m) = \sum_n f(n)m(m-n)\]

Please note, that for in both cases the second function is reversed and shifted. Outcome of convolving two functions is a third function, which is a measure of amount of overlap between two convolved functions. It is clearly visible, that in either time or discrete domain convolving two functions requires lots of complex computations (in case of time domain numerical methods have to applied as well). With the Fourier Transform, this tedious operation becomes as easy as multiplication - the convolution theorem says:

\[\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)\]

This way of convolving signals, which in terms has application in signal processing filtering tasks can be very fast and efficient. The process of transforming the data into frequency domain is handled by the FFT algorithms, which will be discussed later on. Difference between both approaches is rather huge: in the first case, when using discrete convolution it would require \(O(n^2)\) operations, whereas with FFT it requires only \(O(n \log n)\) computations.

**Two-dimensional Fourier Transform**

**Introduction**

We are already acquainted with all advantages that come with one dimensional Fourier Transform: much easier filtering and less time needed for completing it. Why should not then this be applied to more dimensions? In fact, Fourier transform can be expanded to arbitrary number of dimensions. The following formula presents this possibility:

\[f(x) = (\mathcal{F}^{-1}_n F)(x) = \frac{1}{(2\pi)^n} \int F(\omega)e^{i\omega, x} d\omega\]

Vectors \(\omega\) and \(x\) are n-dimensional vectors, \((\omega, x)\) is the inner product. Of course, integration is performed over all dimensions. This significantly increases required number of computations (applies especially to computer aided analysis), but from the other point it enables us to reuse knowledge we already posses about the one dimensional transform (will be discussed later on).

The more dimensions, the harder the analysis is. In case of two dimensional transform it is very hard to interpret phase spectrum images (“We generally do not display PHASE images because most people who see them shortly thereafter succumb to hallucinogenics or end up in a Tibetan monastery” - from John M. Brayer's website). Magnitude spectrum images are of the most importance.

As it was mentioned previously, two dimensional transform will speed up
filtering and other analysis of two dimensional waveforms. Nowadays 2D Fourier Transform is computed by means of 2D FFT algorithm and is applied to images, geophysical arrays, gravity and magnetic data and is helpful in antenna analysis.

**Definition of two-dimensional Fourier Transform**

Let us assume that we have a two-dimensional function \( h(x,y) \), Fourier Transform for this function will be defined by the following integral:

\[
H(u,v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x,y) e^{-j2\pi(ux+vy)} \, dx \, dy
\]

If the function \( h(x,y) \) is separable (topological space is separable if it contains a countable dense subset), then integration can be performed in two steps, which will considerably simplify the whole case:

\[
H(u,y) = \int_{-\infty}^{\infty} h(x,y) e^{-j2\pi ux} \, dx
\]

\[
H(u,v) = \int_{-\infty}^{\infty} h(u,y) e^{-j2\pi vy} \, dy
\]

Similar to one dimensional case, function \( h(x,y) \) will be decomposed into components of the form \( \cos[2\pi(ux+vy)] \) and \( \sin[2\pi(ux+vy)] \).

Since it is possible to calculate two-dimensional transform in two steps, then it yields the following conclusion, that two-dimensional transform can be viewed as two successive one-dimensional Fourier transforms. For example, when calculating two-dimensional FT of an image (spatial data set on the contrary to time referenced, which is named temporal), firstly a one dimension FT is calculated over all rows and substituted and in the second step again one dimensional transform is calculated over all rows (notice, that results from transforming columns are in the input array!).

\[
H(u,v) = \int_{-\infty}^{\infty} e^{-j2\pi v} \left[ \int_{-\infty}^{\infty} h(x,y) e^{-j2\pi ux} \, dx \right] \, dy
\]

Possibility of applying two successive one-dimensional transforms will be a crucial feature for calculating 2D FT. Details of FFT will be discussed later on.

**Inverse two-dimensional Fourier Transform**

Inverse transform exists and is defined by the following equation:

\[
h(x,y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} H(u,v) e^{j2\pi(ux+vy)} \, dudv
\]

The principle is still exactly the same is in case of one-dimensional transform: it is enough to transform the equation once again with different sign by the exponent.

Please note, that function \( h(x,y) \) has to satisfy all previously, but extended to
two dimensions rules in order for the forward and inverse transform to exist.

**Discrete two-dimensional Fourier Transform**

Again it is possible to reuse all statements derived previously – thus – we can simply rewrite continuous definition into discrete one:

\[
H(u, v) = \frac{1}{MN} \sum_{x=0}^{M} \sum_{y=0}^{N} h(x, y) e^{-j2\pi \frac{ux}{M} \frac{vy}{N}}
\]

And similarly for the inverse two-dimensional transform:

\[
h(x, y) = \sum_{u=0}^{M} \sum_{v=0}^{N} H(u, v) e^{j2\pi \frac{ux}{M} \frac{vy}{N}}
\]

It is clearly visible, that in order to compute a discrete Fourier Transform it is necessary to perform x*y complex multiplications and x*y-1 additions to get the result. For a picture that has size 1024x768 pixels this yields 786432 complex multiplications and 786431 summations to get the final result. Imagine, that the picture was taken with a 5Mpx digital camera – then image size would be 2592x1944 pixels, which yields 5038848 complex multiplications. That is why it said, that DFT is of complexity N^2 (in case of image it might be N*M according to previous definitions).

**DFT – computational bottleneck, FFT – efficiency winner**

As it was mentioned previously, DFT of N samples will result in N*N complex multiplications and N-1 additions. To be more precise, it is said that DFT is of O(N^2) complexity. The big O, which can be referred to as the “big O notation” is used to describe asymptotic behavior of functions, which in terms can be expressed by simpler functions. In other words, by means of the O notation it is possible to establish upper bound (asymptote) of the number of required computations.
The big O notation is very useful when it comes to describe an algorithm, as it immediately gives “the cost” of execution. Recalling, DFT is of $O(N^2)$ complexity, which is usually referred to as **quadratic complexity**. Sequences that contain large number of samples will take a lot of time to compute even on very fast computers. That is why nowadays FFT algorithm is used for calculating DFT. It is a very efficient algorithm that exploits properties of Fourier Transform and gives the same result as direct calculation of DFT in only $O(N \cdot \log_2(N))$ operations. FFT and its properties will be discussed in details in the next chapter.

**Fast Fourier Transform algorithm**

Fast and efficient way of calculating Discrete Fourier Transform, which reduces number of arithmetical computations from $O(N^2)$ to $O(N \cdot \log_2(N))$. Key of the algorithm is data reorganization and further operations on it.

In 1965 a first paper about efficient way of calculating DFT was published by IBM researcher James Cooley and Princeton faculty member John Tukey. Surprisingly, they were not the first ones that invented FFT. It was proved, that FFT was firstly invented by Carl Fredrich Gauss around 1805. It was not publicly known just because Gauss did not publish it. Cooley and Tukey reinvented exactly the same thing which is being used till today. Great rediscovery made by Cooley and Tukey put also some new features into the algorithm, which makes it more flexible.

**Radix-2 Butterfly**

The so called “Cooley-Tukey” algorithm is by far the most common FFT algorithm. The main principle is that the DFT of size $N$ is split into smaller DFTs of sizes $N_1$ and $N_2$, where the relation between both is $N = N_1 \cdot N_2$. DFT is split recursively, but of course it is possible to create iterative version of the algorithm. Depending on which is the radix, the algorithm can be:

- DIT, which is called “Decimation in Time” and is used when $N_1$ is radix
- DIF, which is called “Decimation in Frequency” and is used when $N_2$ is radix

**DIT**

Decimation in time is the simplest version of Cooley-Tukey algorithm. Because of that it lacks performance, but is often used as an example. The decimation in time divides the problem into two subproblems:

$$
X_r = \sum_{k=0}^{N-1} x_{2k} e^{-j \frac{2\pi}{N} (2k+1)r} + \sum_{k=0}^{N-1} x_{2k+1} e^{-j \frac{2\pi}{N} (2k+1)r} = \sum_{k=0}^{N-1} x_{2k} e^{-j \frac{2\pi}{N} (2k+1)r} + e^{-j \frac{2\pi}{N} r} \sum_{k=0}^{N-1} x_{2k+1} e^{-j \frac{2\pi}{N} (2k+1)r}
$$

$r=0, 1, \ldots, N-1$
These two subproblems form sets of even- and odd-indexed data each of size \( \frac{N}{2} - 1 \) and are solved recursively. Both summations resemble two DFTs and thus can be rewritten into the following form (will simplify further derivations):

- even set

\[
E_r = \sum_{k=0}^{\frac{N}{2}-1} x_{2k} e^{-j \frac{2\pi}{N} (2k)r}, \quad r = 0, 1, ..., \frac{N}{2} - 1
\]

- odd set

\[
O_r = \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1} e^{-j \frac{2\pi}{N} (2k+1)r}, \quad r = 0, 1, ..., \frac{N}{2} - 1
\]

First \( \frac{N}{2} \) terms of our solution is obtained from:

\[
X_r = E_r + e^{-j \frac{2\pi}{N} r} O_r, \quad r = 0, 1, ..., \frac{N}{2} - 1
\]

Remaining terms can be obtained by taking into account the following identities:

\[
(e^{-j \frac{2\pi}{N}})^{\frac{N}{2} + r} = -e^{-j \frac{2\pi}{N} r} \quad \text{and} \quad (e^{-j \frac{4\pi}{N}})^{\frac{N}{2}} = 1.
\]

Having in mind above identities solutions for elements \( r + \frac{N}{2} \) can be obtained from:

\[
X_{r + \frac{N}{2}} = E_r - e^{-j \frac{2\pi}{N} r} O_r, \quad r = 0, 1, ..., \frac{N}{2} - 1.
\]

Equations

\[
X_r = E_r + e^{-j \frac{2\pi}{N} r} O_r, \quad r = 0, 1, ..., \frac{N}{2} - 1
\]

and

\[
X_{r + \frac{N}{2}} = E_r - e^{-j \frac{2\pi}{N} r} O_r, \quad r = 0, 1, ..., \frac{N}{2} - 1
\]

are the so called Cooley-Tukey butterfly, as the flow diagram resembles a butterfly (image from Wikipedia):
The factor $e^{-j2\pi r/N}$, $r=0, 1, \ldots, \frac{N}{2}-1$ present in butterflies is the so called twiddle factor (root of unity). Twiddle factors can be precomputed and stored in memory for later usage; calculation of these can be further optimized either by using trigonometric identities or by using ASM level optimizations (subsequent calls to \texttt{fsin} and \texttt{fcos} can be substituted by one \texttt{fsincos} call; this instruction is available in the x87 floating-point extension).

It should be also noted, that the length of input vector should be a power of two. It is a general rule, that for \texttt{radix-r} FFT the input vector should be of length $r^n$. For example, \texttt{radix-4} version is very attractive because of the twiddle factors that have values 1, -1, j or -j which simplify a lot further multiplications – but of course, this costs the fact, that input vector has to be the power of 4.

As it was mentioned previously, Cooley and Tukey made the algorithm more flexible. This means, that for a non-prime length of a sequence we can calculate a composite FFT (this is the so called “mixed radix”). Let’s say, that we have a sequence consisting of 200 samples. It can be calculated by means of \texttt{radix-2} and \texttt{radix-10}. How? Length of the sequence is 200 samples and this can be split into $200 = 2*10*10$, so that we can use radix-10 twice and radix-2 once. Cooley and Tukey in their paper showed that it is possible to provide algorithms for arbitrary $r$ in the radix. Exactly the same things were derived by Gauss, who left in his work derivations of \texttt{radix-3} and \texttt{radix-6} algorithms.

Mixed radix method cannot be applied to prime sizes. It is possible to calculate FFT for a prime-size sequence, but these methods are much less efficient than ones for non-prime sizes. Besides, there are several ways of avoiding prime sizes in sequences (in general, the first step should be to do everything that is possible to get a non-prime size of the sequence; this operation is almost always successful).
Decimation in frequency, which is also referred to as Sande-Tukey algorithm, decimates the output frequency series into even- and odd-indexed sets. Once again there are two problems to be solved:

\[
X_r = \sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}kr} + \sum_{k=\frac{N}{2}}^{N-1} x_k e^{-j\frac{2\pi}{N}kr} = \sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}kr} + \sum_{k=0}^{\frac{N}{2}} x_k e^{-j\frac{2\pi}{N}(k+N)r}
\]

\[
X_r = \sum_{k=0}^{\frac{N}{2}-1} \left( x_k + x_{k+N/2} e^{-j\frac{2\pi}{N}r} \right) e^{-j\frac{2\pi}{N}rk}, r=0, 1, ..., N-1
\]

Similarly to the DIT case it is possible to define subproblems as:

- even set

\[
E_r = \sum_{k=0}^{\frac{N}{2}-1} \left( x_k + x_{k+N/2} e^{-j\frac{2\pi}{N}kr} \right), r=0, 1, ..., \frac{N}{2}-1
\]

- odd set

\[
O_r = \sum_{k=0}^{\frac{N}{2}-1} \left( \left( x_k - x_{k+N/2} \right) e^{-j\frac{2\pi}{N}kr} \right), r=0, 1, ..., \frac{N}{2}-1
\]

Terms in "( )" brackets will be calculated first and will complete the subdivision part of calculations. After mentioned subproblems are solved there are no other calculations that have to performed. Most of the work is done during the subdivision phase, when the subproblems are prepared. Computations in the subdivision part are called Gentleman-Sande butterfly (can be presented the same way as Cooley-Tukey butterfly).

Summarizing, during the subdivision part it is necessary to calculate the following values: \( e_k = x_k + x_{k+N/2} \) for even elements and \( o_k = (x_k - x_{k+N/2}) e^{-j\frac{2\pi}{N}k} \), where \( k \) in both cases is \( k=0, 1, ..., \frac{N}{2}-1 \).

**Bit reversed**

Bit reversal is a permutation that reorders elements in the vector in the following way:

- index of an element is converted into binary number (3\(_{\text{dec}}\) = 011\(_{\text{bin}}\))
- binary representation is reversed (mirror reflection) (011\(_{\text{bin}}\) becomes 110\(_{\text{bin}}\), which is 6\(_{\text{dec}}\))

In case of radix-2 and DIT, output produced by the algorithm is scrambled, meaning, the algorithm should take bit-reversed data in order to return in-order data. Opposite situation is for the DIF, where the algorithm takes in-
order data and produces bit-reversed result.

Usually bit-reversal process is a distinct part of the FFT routine, as most users prefer natural order of data. Of course, from the machine's point of view it does not matter. Convolution works equally on both in-order and bit-reversed data.

Bit reversal problem used to be a field of active research. Nowadays complexity of bit reversal algorithms is O(N) (linear). It is very important to remember to either pre or post process the data. It is of great importance when talking about in-place transforms, meaning, transform whose results overwrite input data. Completely different situation occurs when we talk about out-of-place transforms (result is written to a separate array and is usually in natural order).

**Implementing radix-2 DIF**

The best way of presenting theory is to show its application – meaning – to learn by example. In this paragraph a radix-2 decimation-in-frequency algorithm will be implemented as well as routine for precomputing twiddle factors, which will use Singleton's algorithm. Function will be implemented in C according to the pseudo-code examples in the book “Inside the FFT Black Box”.

**Twiddle factors**

Twiddle factors are roots of unity. They are equally spaced on the unit circle, thus when a root of unity is raised to a $n^{th}$ power then the result should be one. The fact, that roots of unity are placed on the unit circle yields the following conclusion:

$$e^{j\alpha} = \cos(r\alpha) + jsin(r\alpha) \times \alpha = 2\frac{\pi}{N}, N=2^n$$

Having in mind that, we can say, that if $a+jb$ is a root of unity, then also $±a±jb$ and $±b±ja$ are roots of unity. According to that we can calculate only first $\frac{N}{2}−1$ values. The easiest way is to calculate first values of cos and sine functions using standard library functions or ASM optimized calls (fsincos that was mentioned before). Other solution will exploit trigonometric identities. Algorithm that is based on them was proposed by Singleton, and we are going to develop our twiddle factor routine according to the Singleton's algorithm.

Below is a C implementation. Pointers $wcos$ and $wsin$ should point to already allocated memory, and that should be $\left\lfloor \frac{N}{2}+1 \right\rfloor \cdot \text{sizeof(double)}$. $N$ of course should be equal to number of samples in the input vector. Resulting complex number is in algebraic form, where $wcos$ is the real part and $wsin$ imaginary part.
void twiddle_factors(double *wcos, double *wsin, int N) {
    double alpha, S, C;
    int K, L;

    alpha = (double) TWO_PI/N;
    S = sin(alpha);
    C = 1 - 2*pow(sin(alpha/2), 2);
    wcos[0] = 1;
    wsin[0] = 0;

    for(K=0; K<=(N/8)-2; K++) {
        wcos[K+1] = C*wcos[K] - S*wsin[K];
        wsin[K+1] = S*wcos[K] + C*wsin[K];
    }

    L = N/8;
    wcos[L] = sqrt(2)/2;
    wsin[L] = sqrt(2)/2;

    for(K=1; K<=(N/8)-1; K++) {
        wcos[L+K] = wsin[L-K];
        wsin[L+K] = wcos[L-K];
    }

    L = N/4;
    wcos[L] = 0;
    wsin[L] = 1;

    for(K = 1; K<=(N/4); K++) {
        wcos[L+K] = -wcos[L-K];
        wsin[L+K] = wsin[L-K];
    }
}

All twiddle factors should be computed only once in the beginning. They can be also stored in a file for further retrieval (in almost all cases it is not necessary).

**Radix-2 DIF FFT**

Since we have precomputed twiddle factors, now it is time to implement the FFT routine. Please, keep in mind that for radix-2 DIF FFT we insert in-order data and as a result we get bit-reversed DFT (or in other words scrambled result).
Native Complex format is not used. Instead of that, the input vector is a 2 element array of double (which is binary compatible with C99 revision, thus the native complex format):

```c
typedef double cplx[2];
```

The same “trick” was used by developers of the FFTW library (fftw complex), which makes the development process much easier, as the implementation does not require special index calculation (as opposed to the “Numerical Recipes” version, which stores data as consecutive elements, where the first one is Real part and second one is Imaginary). “REAL” and “IMAG” are preprocessor macros (REAL is 0 and IMAG is 1). Function relies on previously computed twiddle factors.

```c
void fft_r2_dif(cplx *a, double *wcos, double *wsin, int N) {
    int NumOfProblems = 1;
    int ProblemSize = N;
    int HalfSize;
    int K, J; int JFirst, JLast, Jtwiddle;
    cplx W, Temp,Tmp;

    while(ProblemSize > 1) {
        HalfSize = ProblemSize/2;
        for(K=0; K<=NumOfProblems­1; K++) {
            JFirst = K*ProblemSize;
            JLast = JFirst + HalfSize -1;
            Jtwiddle = 0;

            for(J = JFirst; J<=JLast; J++) {
                W[REAL] = wcos[Jtwiddle];
                W[IMAG] = wsin[Jtwiddle];

                Temp[REAL] = a[J][REAL];
                Temp[IMAG] = a[J][IMAG];

                a[J][REAL] = Temp[REAL] + a[J + HalfSize][REAL];
                a[J][IMAG] = Temp[IMAG] + a[J + HalfSize][IMAG];

                Tmp[REAL] = Temp[REAL] - a[J+HalfSize][REAL];
                Tmp[IMAG] = Temp[IMAG] - a[J+HalfSize][IMAG];

                W[REAL]*Tmp[REAL];
                a[J+HalfSize][REAL] = W[REAL]*Tmp[REAL] -
                W[IMAG]*Tmp[IMAG];
                a[J+HalfSize][IMAG] = W[REAL]*Tmp[IMAG] +
                W[IMAG]*Tmp[REAL];
        }
        Jtwiddle = Jtwiddle + NumOfProblems;
    }
}
```
Transform is performed in-place, so that inputs are overwritten with outputs. The last operation on data will unscramble the result, so that it will be in natural order:

```c
/*
 * BIT-REVERSE
 * Based on numerical recipies. Contains João Martins modifications.
 * Data has to be post-processed, as the radix-2 DIF takes in-order data and
 * produces bit-reversed result.
 */
j=0;
for (i=0;i<(N/2);i++) {
    if (j > i) {
        // Swap Re and Im parts
        SWAPR(inp[j],inp[i]);
        SWAPI(inp[j],inp[i]);
        // checks if the changes occurs in the first half
        // and use the mirrored effect on the second half
        if((j/2)<(N/4)){
            // Swap Re and Im parts
            SWAPR(inp[(N-(i+2))],inp[(N-(j+2))]);
            SWAPI(inp[(N-(i+2))],inp[(N-(j+2))]);
        }
    }
}
m=N/2;
while (m >= 2 && j >= m) {
    j -= m;
    m = m/2;
}
j += m;
```

Macros “SWAPR” and “SWAPI” have to be defined before the code and they look as follows:

```c
#define SWAPR(a,b) tempr[0]=a[0];a[0]=b[0];b[0]=tempr[0];
define SWAPI(a,b) tempr[1]=a[1];a[1]=b[1];b[1]=tempr[1];
```

Routine does not perform any other actions on the data (DFT is not centered).
Testing radix-2 DIF FFT

The easiest way of testing is to take a sequence of values, calculate it in two different pieces of software and then compare results. In all cases MATLAB was used for comparing obtained results.

- test for a N = 8 input vector

  MATLAB

```matlab
>> input = [ 1 1 5 5 5 5 1 1 ];
>> fft(input)'

ans =

    24.0000
   -9.6569 + 4.0000i
         0
   1.6569 - 4.0000i
         0
   1.6569 + 4.0000i
         0
   -9.6569 - 4.0000i
```

- radix-2.c

Twiddle factors (first N/2 = 4 only):

```
[0] cos = 1.000000, sin = 0.000000
[1] cos = 0.707107, sin = 0.707107
[2] cos = 0.000000, sin = 1.000000
[3] cos = -0.707107, sin = 0.707107
[4] cos = -1.000000, sin = 0.000000
```

Input data (N = 8 samples):

```
[0] Re(1.000000)        Im(0.000000)
[1] Re(1.000000)        Im(0.000000)
[2] Re(5.000000)        Im(0.000000)
[3] Re(5.000000)        Im(0.000000)
[4] Re(5.000000)        Im(0.000000)
[5] Re(5.000000)        Im(0.000000)
[6] Re(1.000000)        Im(0.000000)
[7] Re(1.000000)        Im(0.000000)
```

radix-2 DIF FFT: 0 ms (* see the text below)  

Bit-reversed result:

```
[0] Re(24.000000)        Im(0.000000)
[1] Re(-9.656854)        Im(4.000000)
[2] Re(0.000000)         Im(0.000000)
[3] Re(1.656854)         Im(-4.000000)
```
Assessment: radix-2.c works correctly.

(*) “0 ms” is because the counter's resolution is too small. This function will be used to evaluate this algorithm's speed later on.

Source code for the radix-2.c program can be found in Appendices.

**Radix-2 in numbers: cost**

DIF uses Gentleman-Sande butterfly, which for sequence of length $N$ requires $N$ complex additions and $\frac{N}{2}$ complex multiplications. Of course, one has to realize the fact, that complex addition consists of two real additions and complex multiplication consists of three real additions and multiplications (assuming precomputed twiddle factors). In terms of floating point operations (FLOPs), the total cost can be established as:

- $N$ complex additions is $2*N$ flops
- $\frac{N}{2}$ complex multiplications yields $3N$ flops (applied only to half of the sequence, that is why there is only $3N$)

Total complexity of the algorithm is then $O(N)=5N\log_2(N)$.

**Two-dimensional FFT: adapting 1D FFT**

As it was mentioned before, in case of calculating two-dimensional DFT it is possible to divide the process into two parts, where only one-dimensional DFTs are involved. Thus, for implementing 2D FFT we are going to use 1D routines.

**2D – computational madness**

For two-dimensional case the number of arithmetical calculations grows rapidly and requires much more storage space (memory). For an input vector that has 1024 samples, the situation is rather easy and amount of occupied space is not critical (assuming 8bytes double precision type it would be $1024*8 = 8192$bytes). In case of image, whose size is 1024x1024 pixels (gray scale, 8bpp\(^4\)) we have an array of the same size, which gives 1048576 values to be stored in memory. For a double precision\(^5\) format it would be 8192kB, and if there is no distinction between purely real and mixed input for the FFT routine, then it will be 16384kB just for one gray scale image! This of course

\[4\] Bpp means “bits per pixel”; number of bits available for representing the pixel's color
\[5\] IEEE 754 Double Precision
can incur several other problems: it is harder to distribute computations\(^6\) and in most cases cache memory will not be used efficiently (cache miss occurrence). Making the routine and software itself cache friendly can be beneficial in terms of speed. Cache strategies, design and general impact on input array size will be discussed later on. As for now it should be assumed that we are going to use the C-style way of storing arrays, which can be either row-major or column-major format.

Let us assume to have N*M matrix. Row-major format simply puts all rows (N) of the array one after another (M times), so that as a result we have a N*M vector. Similar situation is for column-major format – but this time columns are put one after another (column-major is more popular among Fortran programmers).

![Diagram](Illustration 3: Row- and column-major ways of storing arrays (also called “C style way”))

**Programming considerations and overview of the method**

We are going to have our data in the row-major format, which is somehow more popular among C programmers. The vector has N*M entries (NxM matrix) and we are going to apply 1D transforms firstly to all rows and then to all columns. This will incur:

- N 1D transforms over rows, each row having M samples
- M 1D transforms over columns, each column having N samples

The first question that arises is: how the second step of the 2D FFT method will be performed, since in the first one the FFT routine is going to take rows as they are placed in the memory – then what with the other step? It would be

\(^6\) Case, when data has to be transferred to another machine (CPU) and system is supposed to work in real-time

31 of 116
very inefficient to read column data from exactly the same vector, as it would
generate lots of cache misses. This requires intermediate stage that will
transpose the matrix, so that rows will be in places of columns. After the
second step, matrix can be transposed once again if necessary to obtain data
in the same format as input. This in fact means, that input data can be in
either row- or column- major formats – it has no difference at all for the FFT
routine. For better performance, for example if on the output one wants row-
major format, then input vector can be in column-major and then the last
transposition can be omitted.

Transposition

In-place matrix transposition can be tricky and difficult, but nevertheless it is
possible even to parallelize this process. The most common sequential solution
uses the divide and conquer method, which by definition involves recursion.
Input matrix is divided into smaller sub matrices. Most general algorithms
work well on both square and rectangular matrices.

For purposes of this example we are going to use the simplest method which
is of course out of place transposition (requires a copy of the transposed
matrix).

Parallelization and 2D FFT

In current situation for a N*M matrix when we have to perform firstly N 1D
FFTs and then M 1D FFTs on distinct rows and columns (afterwards), then it
is of course relatively easy to parallelize this process. We setup appropriate
number of threads, which should be equal to number of processors present in
the system\(^7\). Threads have to be synchronized only between FFT stages, when
the matrix has to be transposed. When performing out of place FFT transform,
then it is even easier, as the intermediate result (FFT from rows) can be
written to the new matrix in opposite order and then used again by the same
routine (second step can use input vector to store data, afterwards it would be
enough just to swap pointers). It is worth noticing, that precomputed twiddle
factors\(^8\) can and should be used regardless of the fact, if the FFT is going to
be computed fully sequentially or in parallel. Calculating them on the fly is a
bad idea, as for two-dimensional input arrays total number of computations

\(^7\) This is a general rule, but of course, it is worth experimenting with this number (having in
mind, that too big number of threads will decrease the performance significantly).

\(^8\) Threads give you shared memory space and twiddle factors will be only read, thus no
additional work and synchronization is needed in this case.
grows very fast (twiddle factors calculations = N*M).

Further analysis of parallel algorithms for two-dimensional FFT involves different flavors of one-dimensional transform. 2D FFT is also very attractive when it comes to radix-4, for which twiddle factors are just 1, -1, j and -j. This can additionally give some speed up, but at the cost of input size limitations (power of 4).

**Implementation of 2D FFT: reusing 1D FFT**

For implementing 2D FFT it is possible to reuse previously developed functions, both for twiddle factors calculation and the FFT routine. This time it is not enough just to have the FFT routine, as number of additional operations is much higher and demands much more attention that in previous cases. 2D implementation can be divided into the following steps (pseudo-code):

```plaintext
input := N*M matrix
twiddle_factors(length M)
from 0 to N do
    fft_transform(slice of length M)
    bit_reverse(slice of length M)
done
transpose_matrix(N*M becomes M*N)

from 0 to M do
    fft_transform(slice of length N)
    bit_reverse(slice of length N)
done
transpose_matrix(M*N becomes N*M)
```

Routines `fft_r2_dif()` and `twiddle_factors()` are exactly the same as in previous, 1D example. Bit-reverse part of code was put into function named `bit_reverse()`. Knowing this it has much more sense to present the body of the program to give global overview of what is happening (debugging routines were cut):

```plaintext
/*
 * ROWS
 * /
    twiddle_factors(tc, ts, cols);
    print_twiddles(tc, ts, cols);

    TIME_START
    for(i=0; i<rows; i++) {
        fft_r2_dif(inp+i*cols, tc, ts, cols);
        bit_reverse(inp+i*cols, cols);
    }
```
TIME_STOP("FFT over rows")

/*
 * Matrix transposition: quick and dirty
 */

TIME_START
for(i=0; i<rows; i++) {
    for(j=0; j<cols; j++) {
        transp[j*rows + i][REAL] = inp[i*cols+j][REAL];
        transp[j*rows + i][IMAG] = inp[i*cols+j][IMAG];
    }
}
// Rewrite results
for(i=0; i<rows*cols; i++) {
    inp[i][REAL] = transp[i][REAL];
    inp[i][IMAG] = transp[i][IMAG];
}
TIME_STOP("Transposition")

/*
 * COLUMNS
 */

TIME_START
twiddle_factors(tc, ts, rows);
TIME_STOP("Twiddle factors for FFT over columns")
print_twiddles(tc, ts, rows);

TIME_START
for(i=0; i<cols; i++) {
    fft_r2_dif(inp+i*rows, tc, ts, rows);
    bit_reverse(inp+i*rows, rows);
}
TIME_STOP("FFT over columns")

printf("T R A N S P O S E...
");
TIME_START
for(i=0; i<rows; i++) {
    for(j=0; j<cols; j++) {
        transp[i*cols+j][REAL] = inp[j*rows + i][REAL];
        transp[i*cols+j][IMAG] = inp[j*rows + i][IMAG];
    }
}

for(i=0; i<rows*cols; i++) {
Matrix transposition could be of course solved in a more elegant way. Version implemented in the example is the fastest one in terms of time needed to write it – not to execute it. Matrix transposition was mentioned before – it seems to be a very simple matrix operation, and in fact it is very easy to do it on a piece of paper. In case, when one has to do it in-place, and at the same time maintain the row-major or column-major way of storing data, then it becomes really tricky.

Another thing is, that since we know the matrix size from the very beginning, then it is also possible to calculate twiddle factors in advance and store them in two different memory locations (only for non-symmetrical matrices and thus should be applied only in case, when the first transform has to be calculated as fast as all others).

2D FFT: test

Test method was exactly the same as in previous case: the same input array was computed in MATLAB and by the 2d.c program. Results are below:

- MATLAB results

```matlab
a =

 1   2   3   4   5   6   7   8
 1   2   3   4   5   6   7   8

>> fft2(a)'

% Result is transposed! (clarity purposes)
ans =

  72.0000        0
-8.0000  -19.3137i  0
-8.0000  - 8.0000i  0
-8.0000  - 3.3137i  0
-8.0000        0
-8.0000  + 3.3137i  0
-8.0000  + 8.0000i  0
-8.0000  +19.3137i  0
```

- 2d.c results

```
Input data (N = 16 samples):
==== Input data ====
```
Processing rows: twiddle factors, ffts, bit-reversal, transposition.

Twiddle factors (first N/2 = 4 only):

\[
\begin{array}{ll}
[0] & \cos = 1.000000, \sin = 0.000000 \\
[1] & \cos = 0.707107, \sin = 0.707107 \\
[2] & \cos = 0.000000, \sin = 1.000000 \\
[3] & \cos = -0.707107, \sin = 0.707107 \\
[4] & \cos = -1.000000, \sin = 0.000000 \\
\end{array}
\]

R_FFT: 2 FFTs of 8 samples

* * * * * FFT over rows: 0 ms

TRANSPose...

* * * * * Transposition: 0 ms

Processing columns: twiddle factors, ffts, bit-reversal.

* * * * * Twiddle factors for FFT over columns: 0 ms

Twiddle factors (first N/2 = 1 only):

\[
\begin{array}{ll}
[0] & \cos = 0.000000, \sin = 1.000000 \\
[1] & \cos = 0.707107, \sin = 0.707107 \\
\end{array}
\]

C_FFT: 8 FFTs of 2 samples

* * * * * FFT over columns: 0 ms

TRANSPose...

* * * * * Transposition: 0 ms

==== Result (transposed) ====

\[
\begin{array}{cccccccccc}
72.0 & 0.0 & -8.0 & -19.3 & -8.0 & -8.0 & -8.0 & -8.0 & 0.0 & -8.0 \\
-8.0 & 19.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\end{array}
\]

Total time: 0

Computations vs. methods

If one plans to create a general purpose FFT package, then it should include lots of error handling mechanisms, data padding methods and similar. In the 2D FFT implementation where 1D FFT routine was reused, the main program that was calling \texttt{fft_r2_diff()} became longer and more complex. Putting additional logic and data correction routines (for transform sizes not being radix powers) will result in much more lines and even more complex structure. As it was mentioned before, change of the \textit{radix} parameter (especially increasing) can decrease both complexity and number of computations. The price that one has to pay for doing that are limitations that are put on input data (number of samples). The perfect solution in this case would be to have a set of implemented algorithms that before first run can be tested, put into a tournament or similar time-challenge, so that it would be possible to pick up the fastest solution.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Real multiplications</th>
<th>Real additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix-2</td>
<td>81924</td>
<td>139266</td>
</tr>
<tr>
<td>Radix-4</td>
<td>57348</td>
<td>126768</td>
</tr>
<tr>
<td>Radix-8</td>
<td>49156</td>
<td>126978</td>
</tr>
<tr>
<td>Radix-16</td>
<td>48132</td>
<td>125442</td>
</tr>
</tbody>
</table>

Table 1: Operations needed for computing FFT for \( N = 4096 \) (table taken from "Fast Fourier Transform and its applications", E. Brigham)

It was also mentioned a few times, that for the **radix-4** twiddle factors have very attractive form. Similar situation is for **radix-8**, where twiddles are just \( \pm 1, \pm j, \pm e^{\frac{\pi}{4}}, \pm e^{-\frac{\pi}{4}} \). Runtime checks might be too expensive – but plan-based computations which include preparations, that can take much more time than actual execution can be beneficial. According to this principle FFTW library was designed. Library itself will be covered in later chapters.

**Algorithms for real data**

Especially for real-time systems it might be worth distinguishing between the input type, if it is either purely real or complex. Why? In case, when we know that input data is purely real, we also know that complex part of each number is equal to zero. Complex multiplication, which normally looks as follows:

\[
(a+jb)(c+jd) = (ac-bd) + j(ad+bc)
\]

becomes

\[
a(c+jd) = ac+jad
\]

Input vector will take only half of the memory space, thus it will be more cache friendly.

For real-systems each millisecond is very important, because usually the same steps are performed several times over one second. “One millisecond here, one millisecond there...” - and finally the improvement might be something we would not even expect it to be.
Part II - “Not everything that can be counted counts, and not everything that counts can be counted.”

Albert Einstein

Linux as destination operating system

The whole story began much earlier before Linux was launched in 1991 by Linux Torvalds. Unix is believed to be the best planned operating system for multiuser environments. Being Multics successor, Unix grew and became a model for many other operating systems, that nowadays are called “*nix family” operating systems.

Linux being not a direct derivative of Unix\(^9\) implements POSIX and SUS standards, so that it is *nix compatible. But the biggest breakthrough is the fact, that Linux is a completely free software, which is released under the General Public License. Unix's experience and ability to develop an operating system by programmers from the whole world became Linux's success. Good design, lightweight architecture and possibility of changing every single piece of the system made it a perfect tool for research environments, as well as it became a perfect way of decreasing TCO\(^{10}\) in companies, which run Linux OS on their servers or desktops.

Introduction to Linux environment

Below paragraphs will try to provide only the most important pieces of information from the operating systems theory followed by examples from Linux OS, and those that are relevant in understanding next chapters, where there will be a word about setting up a Linux research box.

One should ask, why Linux and not other operating system? The idea is quite simple: with Linux one gets a robust and open operating system, which gives a lot of power – but only to those that can embrace it and know enough to make use of every single piece of it. In case of fully commercial operating systems, what one gets is a closed-source piece of software. In these cases possibilities of making changes are of course much smaller.

The idea is not to criticize closed-source commercial products – because they are also good, but in this particular case they might not be the best choice. Many closed-source applications are business-oriented, thus they are universal and user friendly, and this means that lots of useful CPU power is used for running useless things from the research point of view.

---

\(^9\) “Not direct derivative” - in terms of the source code and other aspects of the operating system; some of early Unix assumptions are different in Linux, like swapping.

\(^{10}\) TCO – Total cost of ownership; TCO includes costs of purchase, maintenance and others, that are needed to run a piece of hardware/software. TCO is mainly used to estimate hardware/software investments, but it is not limited to them.
Operating system – organization

Nowadays operating systems have different structures depending on target group of users. It can be viewed also from a perspective where the system either has or does not have a graphic interface and supports or does not support certain groups of hardware or software. Besides these “salesman” arguments and features, there are other, more important concerns that are on the bottom of the implementation. An operating system has to:

- be an abstract layer between user and hardware; it means, that it has to become an abstract machine that creates other abstract elements that can be used at random times (including concurrent). This principle is a foundation of multiprogramming environments (and further time-shared, multi-tasking environments)

- coordinate use of all elements present in the system; this of course has to be bound to policies defined by the administrator (can be also called resource management).

A well designed operating system provides clean and simple abstraction for accessing its resources. Abstraction is a way of designing a homogeneous interface for accessing and controlling hardware, but also other abstractions.

Usually abstraction consists of two halves: one being a part of homogeneous interface system, and the other one, being more or less a pluggable module that provides device-dependent routines. Without this, it would be almost impossible to cope with thousands different hardware implementations.

In Linux the most important abstraction is a file, which can be everything. This abstraction is used for all devices, so that operating on them is as easy as reading/writing to a file. For this purpose operating system has a set of calls that are executed on behalf of the operating system. This is due to the fact, that Linux has a memory manager with protection (direct example of a policy applied to one of the resources). Going even deeper we will find out, that kernel and user applications run in different address spaces and there is no direct access from user space to the kernel space. Mentioned earlier memory protection could not be solved without hardware support. The protection bit is called supervisor bit (so that for example assembly program would not interfere with the operating system giving access to all its resources) and instruction that switches from unprivileged to privileged mode is a called a trap instruction. Of course, all I/O operations are privileged instructions and thus they are available only through system calls.

A process itself is an abstraction, which is represented again by a file in Linux – handling sets of processes will of course result in another abstraction, which
will provide set of routines for modifying its behavior. Full separation of processes is a hard task and requires lots of lines and sophisticated algorithms to minimize the overhead, but the outcome is that then one can create processes on an abstract machine that will have for their own purposes a CPU, memory and access to other devices – an abstract machine, called sometimes “a virtual CPU”.

There should be no doubt about the fact, that it is much easier to build new abstractions with other abstractions. Mentioned above basics about operating systems theory will be quite important when it will be the time to prepare a Linux research box.

**Abstractions make it easier also to port the solution, as whatever is behind “the call” should be transparent to the programmer. Well designed operating system should provide a compromise between security, ability to configure certain things and efficiency.**

A great example of abstraction might be for example the `gettimeofday()` routine, which gives access to the high resolution clock. It might be used for calculating execution time of functions and will work exactly the same on all platforms (where it is implemented of course). Behavior is the same, difference might be in accuracy. But the idea is, that in Linux there is `gettimeofday` that will work exactly the same on x86, x86_64, SPARC, ARM and so on.

**Linux means kernel**

Nothing less, nothing more. The main idea was to provide a set of abstractions that would implement only the most important operations/actions. All others should be placed outside the kernel and use library functions stored on the disk (which will finally use kernel's system calls). The simplest case is when we talk about the `printf();` function. It provides a set of formatting rules and other options and is a part of the `stdio` library. All string operations are made in the user space by `stdio` – the last stage consists of calling `write()`, which is a system call, and printing the string on selected device (screen, file etc.). This makes it possible to keep kernel small and gives also much more possibilities for other applications (as then other applications form the shape of a bare operating system, which is just a kernel).

**Kernel – organization**

Kernel, also referred to as the core is a monolithic software. Despite this fact, it is possible to add and remove pieces of code called modules at runtime. This pluggable architecture is based on system calls responsible for registering/removing modules. A full schematics of the kernel is way too complex and contains too many details that are mostly not needed for our purposes. As it was mentioned before, the kernel provides several services like\(^\text{11}\):

\[^{11}\text{Behind these nice-named services there are thousands of lines that create lower level abstractions and that are responsible for other important aspects (on the contrary to user space, kernel has limited stack, no memory manager and pieces of code should be SMP-aware, thus full synchronization is needed when working in multi-CPU mode).}\]
- task/process manager (scheduler)
- device manager (I/O scheduler)
- memory manager (full separation)
- interrupt manager (all computers use interrupts for communicating with devices)

Memory and in fact operating mode is further divided into two pieces: user and kernel mode. The only way of going through between these two spaces is to do it over system calls. As for now in the 2.6.x kernel there are around 300 system calls. Only necessary system calls get implemented to keep the interface simple.\(^\text{12}\)

There are also other mechanisms and features present in the kernel, which will be especially interesting for us like Symmetric multiprocessing, which was not present in original Unix implementation, thread handling and preemption mechanisms. The last one is very important, as it determines the way the scheduler is going to work, which in turn will affect behavior of the whole system.

A few next paragraphs will describe in more details the whole idea about the scheduler and the way it can affect system's behavior.

**Scheduler and task management**

Scheduler is a part of kernel that is supposed to choose a task that will be executed next. This means, that scheduler is responsible for granting CPU time and thus is responsible for managing CPU resources in an optimal way. The outcome if this work is illusion of parallelism.

\(^{12}\)One of the ideas behind system calls is that they are never meant to do something specific, or in other words “one should never know what the system call can be used for”. 
The way scheduler deals with task execution can have a big impact on system's performance.

Please have in mind the fact, that it is relatively easy to create a parallel version of the 2D FFT routine, which is further called from the main program. The fact that we can influence the way in which tasks will be treated is very important, as it can help us building a system that will have best performance either as a low-latency real-time system or power computer, with bigger latency but high throughput. Of course, combination of both is also possible and should be carefully tested. One could ask: “why it should be tested?”. Generally speaking, “low-latency kernel” will be for sure the best solution for a low-latency system, but it is worth testing all possibilities (currently only 3), because sometimes it might be possible to get out more from the hardware – completely for free.

**Scheduler types**

Before discussing schedulers and their features it is necessary to cover basics of multitasking. Currently most – if not all – operating systems support multitasking. There are several ways of designing multitasking systems – but there is no golden middle that will always provide maximum throughput and minimum latency system.

Generally speaking, multitasking can be implemented by taking into account different scheduling strategies. Main ideas behind these implementations are strictly connected with time slices that are granted for processes in task queue and with possible methods for interrupting tasks that are currently being executed. Some of these approaches are no longer being used, like for example multiprogramming that was replaced with time-sharing systems.

Nowadays we can say, that multitasking systems can be divided into two categories: multitasking with cooperation and multitasking with preemption.

In case of the cooperative scheduler the system relies on processes as they are responsible to give out CPU time regularly to other processes. This means, that wrongly designed program can even halt the whole system. This kind of solution can maximize throughput on stations that are supposed to run particular piece of software, e.g. servers or scientific machines. This approach was slightly revised and implemented as one of available schedulers in 2.6 kernels. It is called “No Forced Preemption”. Of course, when configuring system with this scheduler one has to be aware of the fact that occasional longer delays are possible and there are no other guarantees – in the worst case a process can simply hang whole system. Despite this fact, for scientific purposes this scheduler seems to be the one worth looking at.

Other option is preemptive scheduling. In this approach scheduler decides whether to stop executing a process and grant CPU time to another one waiting in the queue\textsuperscript{13}. The time which is used exclusively by a process being executed is called time slice. Its value is determined dynamically.

\textsuperscript{13}As long as execution takes place in kernel space it is possible to stop execution (preempt the process).
In Linux kernel 2.5 the scheduler was revised and rewritten, so that now it is called a scheduler O(1) which means, that the algorithm is independent of the number of elements in the input set (processes in this case).\(^\text{14}\)

Currently Linux kernel offers two different scheduling algorithms for preemptive systems:
- Voluntary Kernel Preemption (desktop)
- Preemptible Kernel (low-latency desktop)

**Voluntary Kernel Preemption**

In this mode programmers focused their attention on reducing the latency of the kernel. In order to achieve this aim several “explicit preemption points” were added to the kernel code, that result in reducing maximum latency of rescheduling. Frankly speaking, we get a faster response at a cost of slightly lower throughput. Users observe illusion that all applications run “smoothly” even if the system is heavily loaded. In this mode it is possibly to preempt a task even if it is executing code on behalf of the operating system during a system call.

**Preemptible Kernel**

In this case it is possible to preempt the kernel at every point (does not apply to critical sections). This means, that it is possible to preempt a process that is executing for example a system call before natural point of preemption. This scheduler reduces overall throughput, but offers low response time, thus it might be used on both desktop computers and real-time embedded systems, where latency requirements are measured in milliseconds. This scheduler is definitely worth testing, as obtained results might be quite interesting.

For systems that are supposed to work real-time and that will work with huge amounts of data it should be carefully tested which of the schedulers will give maximum performance, but it should be kept in mind for all the time, that throughput means no preemption and low-latency means smaller amount of data to be processed.

As it was mentioned previously, scheduler is responsible for managing time slices. Current strategy must involve time slice changes, as it might influence overall performance. Of course, length of time slice depends on the type of process and on its priority.

Summarizing: it is good to know both the type of the software we are going to use/develop (in terms of latency) and OS internals in case of Linux.

**I/O and CPU bound processes**

I/O bounded processes spend most of their time on initializing and serving I/O operations, thus they require less CPU time, as I/O devices are considered to

\(^\text{14}\) Scheduler in 2.6 kernels is more efficient than the one present in 2.4 series.
be much slower than CPU. It also implies the fact, that I/O bounded processes are not interactive, thus there is no response boundary: it is enough to satisfy I/O devices. In UNIX-like operating systems the scheduler usually favor I/O bounded processes.

Situation is slightly different in case of CPU bounded processes, that require just CPU time either to finish computations or because the process is highly interactive. Scheduler this time tries to execute these processes less frequently, but the time slice is enlarged, as the process is not interrupted by any other events until it is preempted by the kernel.

Above text leads to the following conclusions:
- non-interactive and I/O bounded (oriented) processes are executed more often, but time slice is much shorter (e.g. 10ms)
- highly interactive processes and CPU bounded (oriented) processes are executed less frequently, but time slice is much larger (e.g. 200ms)

Above values should be compared with a standard time slice length, that in this case could be 100ms.

If we are supposed to design a real-time system, that is supposed to perform filtering in frequency domain, we have to take into account the following considerations:
- images will likely be of 0,5 and more Megapixels
- hardware will have to provide high-bandwidth link between CPU and memory
- scheduler's tests should be started from “No Forced Preemption” kernel (and this one will be used for further measurements)

Priorities – influencing scheduler's queue

One of the most popular methods of scheduling tasks is to use prioritized queue. It simply means, that processes that have higher priority will be always in the beginning of the executable queue – and what is very important – in Linux systems they will be also granted longer time slice.

Linux uses also dynamic priorities, thus they can be modified during the runtime. A parameter that carries information about the priority is the “nice” level. Its range is from -20 to 19 – default is 0. The lower the nice is, the higher priority a process has. It is possible to modify “nice” level from both operating system and user space.

#include <unistd.h>
int nice(int inc);

Currently there are two implementations of nice – one that returns 0 when priority change was successful and the other one returns current priority (after change, no matter if it succeed or not). The first case applies to glibc in versions <= 2.2.4 (there is a call getpriority() that returns current nice value).

<table>
<thead>
<tr>
<th><strong>System call</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>nice()</td>
<td>Sets the “nice” level</td>
</tr>
<tr>
<td>sched_setscheduler()</td>
<td>Sets scheduling strategy</td>
</tr>
<tr>
<td>sched_getscheduler()</td>
<td>Returns current scheduling strategy</td>
</tr>
<tr>
<td>sched_setparam()</td>
<td>Sets the real-time priority of a process</td>
</tr>
<tr>
<td>sched_getparam()</td>
<td>Returns current real-time priority of a process</td>
</tr>
<tr>
<td>sched_get_priority_max()</td>
<td>Returns maximum real-time priority</td>
</tr>
<tr>
<td>sched_get_priority_min()</td>
<td>Returns minimum real-time priority</td>
</tr>
<tr>
<td>sched_rr_get_interval()</td>
<td>Sets length of the time slice</td>
</tr>
<tr>
<td>sched_setaffinity()</td>
<td>Sets the CPU affinity mask of a process</td>
</tr>
<tr>
<td>sched_getaffinity()</td>
<td>Gets the CPU affinity mask of a process</td>
</tr>
<tr>
<td>sched_yield()</td>
<td>Release CPU time</td>
</tr>
</tbody>
</table>

*Table 2: Scheduling: system calls*

Most of system calls presented in the table above are connected with the so-called real-time scheduling. The only drawback of using them is that there is no guarantee that requests will be fulfilled according to the real-time boundaries.

**At a first glance priorities might seem to be just a “cosmetic” tool - but they should not be considered this way. It is worth experimenting with priorities, as they can give better real-time results. Of course, the best results will be obtained for applications that are supposed to run for a longer time, as for short periods of time that are close to the time slot, impact of the scheduling priority will not be visible (or only partly).**

**Quick note about I/O schedulers**

Similarly to task management there are also I/O schedulers that can influence the way I/O operations are serviced on a system. This topic is less relevant in case of the FFT problem, but still, it is worth knowing which one should be picked up when configuring the kernel.

**Available I/O schedulers**

**Anticipator**
It is a default disk scheduler. This scheduler is suitable for most environments, but its implementation is complex and code size is large and it might not be a good solution for database systems.

**Deadline**

Simple and compact – the best choice for heavily loaded database systems. In some cases its behavior is almost identical to the anticipatory, thus it is a good choice.

**CFQ**

Best choice for desktop systems – it tries to distribute bandwidth equally among all processes in the system.

**Deadline I/O scheduler in this case seems to be the best choice, as it works similarly to the anticipatory scheduler and its implementation is less complex. This might be interesting, when some results, i.e. images have to be written back to the disk (or some initial images have to be read from the disk). Of course, in case when there is enough RAM memory, designers should always consider possibility of using memory mapped files, which offer much better read/write performance when compared with fastest hard disks. Exactly the same procedure should be followed with all filters that will be used in convolutions.**

**Multi-threaded applications**

Linux treats threads in a very special way – but still, they are just processes and obey exactly the same rules as other tasks present in scheduler's queue. The fact, that they share common address space is not of any interest for the scheduler, because it does not affect scheduling in any way. There are several design issues connected with threads which are directly related to the specifics of an operating system. Details and differences will be explain in next paragraphs.

**Classic vs. modern processes**

Using threads can be presented as “getting two things done at the same time” (exactly this way Java advertises its threaded API). As for now we are acquainted with a process as an entity having access to the abstract machine (which is abstract CPU and memory). With threads there comes a new entity, that is called a modern process. We can have several threads, that execute within one framework offered by the modern process (the biggest advantage of using threads over multi-processed solutions is that they have a common address space and there is no need for implementing complex IPC routines). On uniprocessor machines threads are treated exactly the same as normal processes (on 2.4.x kernels threaded applications had higher priority in
scheduler's queue by default), but the best results can be achieved when threads run in multiprocessor environment, as then concurrent execution turns into **parallelism** (remember, that concurrent does not mean parallel; in case of the first one it is just an information that something can be executed in random order and piece-by-piece, whereas parallelism depicts execution of entities at the same time, literally).

There are two main implementations of threads: user and kernel threads. The first one resembles a situation, where we have a modern process (time multiplexed) with abstract machine, that is again time multiplexed by the physical abstract machine. This simply means, that abstract machines include their own multiprogramming environment. User space libraries make use of this principle (Mach C, POSIX threads) – and this applies to operating systems that implement classic processes. There is one big drawback of this solution – when the master thread is blocked, then all other underlying threads are blocked, too.

Kernel threads offer opposite behavior – when one of the threads is blocked, others can still execute.

Under Linux parallelization can be implemented in two and more ways: either by using **Pthread** library or **clone()**, which in turn calls sys_clone system call.

### sys_clone

The clone() call works similar to the fork() system call, which creates a child process. The difference between them is that with clone() it is possible to create a child process that will have common address space with the parent process – so in fact – will be a thread. The biggest disadvantage of clone() is the fact that it is a Linux specific call and should not be used when the application is supposed to be portable.

### Pthread

LinuxThreads, library that offers portable threaded API seems to be the best available solution in this case (when working in multiprocessor environment). It implements POSIX 1003.1c API which makes it possible to run on several other *nix family operating systems\(^\text{15}\). What actually happens in Linux when threads are created is: each thread is mapped to a single LWP. “Lightweight process” is another name for kernel threads. Linux implements one-to-one strategy, so that each thread is bounded to one LWP. Different situation is under Solaris, where process of creating LWPs is rather expensive – they use many-to-many strategy. In other words, LWP is nothing else than an abstract machine (and as it was mentioned before, since threads share address space, file descriptors and so on, LWPs do not need all pieces of information that are associated to a process – that is why there is the name “lightweight process”). This is a huge advantage over a standard fork() call – firstly because communication between parent and children processes is not a problem, and the second thing is that creating a child process is much more expensive than

\(^{15}\)There exists Windows implementation of Pthread, so it is possible to run Pthreaded applications on a Windows OS, which natively implements modern processes.
creating a thread, as when `sys_fork` is invoked the kernel has to create an independent copy of the address space, file descriptors and so on. This takes time and of course resources.

Other advantage of Pthread over clone() might be the fact, that Pthread comes with a set of additional tools that were meant to simplify synchronization and further operations on threads. There should be more or less no doubt, that for most applications Pthread is the best available solution. It provides clean and understandable interface that makes multi-threaded design both easier and nicer (for example setting up Windows threads is much more complex, especially the syntax and number of parameters passed to the function). It also comes with set of tools needed for synchronizing threads (mutexes etc.).

**Why threads are so important?**

Not much time ago multi-CPU machines were a luxury. Nowadays they slowly become a standard. Generally speaking, whenever there is a situation that we have more than one CPU on the board, it is highly advisable to consider using threads. The word *consider* was used on purpose, as not always it is possible to implement threads and some problems can be solved even better without them. Of course, with or without threads in multi-CPU environment most applications will work much better. The reason for that is quite simple: a process can be present only in one queue (number of scheduler's queues is equal to number of CPUs). For example, an application that we are using and that is CPU-consuming will be bound to CPU A, whereas CPU B will take all other activities that happen in the operating system and will serve other applications (of course, scheduler is responsible for determining this situation).

From the example presented above we can draw one important conclusion: just for two-CPU problem (A and B), our single-process application will be able to use only 50% of available power, because it will be bound only to one CPU! The way of getting 100% of the power is either to implement threads (modern solution) or processes. This also clearly sets the rule, that the number of active threads should be set to the number of CPUs available in the system.

**SMP – more CPUs, more problems**

SMP itself is a computer architecture that allows to have more than one CPU on the board. Processors are connected to a shared memory and each of them can execute arbitrary code regardless of data location. The worst limitation is, that only one CPU can access memory at a time. On modern systems with very fast CPUs the bottleneck usually is the memory access, as the CPU speed is much higher than memory speed (on SMP systems it is of course even worse). Partial solution for this problem can be installing more fast cache memory – but one has to take into account that:

---

16 It is worth looking at OpenMPI, an open source implementation of the MPI protocol.
17 Before 1970 it was exactly opposite: memories were much faster than CPUs

48 of 116
- cache memory can't be infinitely large, as then cache management would take more time than ordinary memory access
- cache memories are usually static rams which offer access times in range of 0.5-5ns; they are too expensive to be used as main memories.

Apart from making memories faster and faster, there are other techniques that are being used, like Hyper Threading Technology introduced by Intel in the Pentium 4 processor. From the OS point of view, HT-capable processor offers SMP interface – thus it is visible as two processors. What HT actually does, is that when a CPU is stalled (needed data is not ready), then the CPU tries to execute another thread/process that is next in the queue. So it means, that it duplicates resources from the logical point of view, but still, there is only one CPU to do the job. Intel claims, that HT can give up to 30% in speed. There is no doubt in saying, that HT speeds up the machine – problem lays in the speed up factor. New approaches leave behind HT and concentrate on dual-core architecture.

**There are certain advantages of using HT under Linux – but not always. For some tasks it might be worth checking if with disabled HT one can obtain better results (and this should be always carefully tested).**

**Dual-core technology**

The main idea is to incorporate two processor cores on one die. From the business point of view, the manufacturing process is much more expensive and requires more transistors than standard one-core dies. From the operating system's point of view both cores can be recognized as physical processors (SMP architecture) or can work in the so called partitioned mode, where each CPU operates independently on its own physical memory. Under Linux dual-core CPUs work in shared memory mode, thus in SMP architecture. In most cases both cores have individual L1 caches and are further connected to one L2 cache (but this differs among processors).

**Both Intel and AMD manufacture dual-core CPUs now. In the first case, Intel Core Duo is a 32-bit processor with low power consumption and is a perfect solution for a portable computer, whereas AMD manufactures dual-core Opterons, which are 64-bit CPUs and have HyperTransport technology.**

**Northbridge problem**

On some of AMD processors this problem does not exist, as they have a memory controller built on the processor die. In case of Intel processors, they connect to the Northbridge via FSB (Front Side Bus), which further connects to the memory, PCI Express or AGP. There is one more link to the Southbridge which connects to slower peripherals (via slow connection). Of course, FSB and connection from the Northbridge to the memory should operate on the

---

18Higher number of transistors means more checks (transistors), higher quality of used materials and different manufacturing process (more complex which means more expensive).
same frequency. It is possible to have smaller frequency on the link from Northbridge to memory – but this of course is only because of backward compatibility and can become (usually becomes) another bottleneck. Operating frequency of the FSB is usually derived from the CPU speed (by modifying FSB it is possible to overclock the CPU; currently most CPUs prevent from doing that). Of course, the higher frequency, the more data can be transferred back and forth.

![Diagram showing the relationship between CPU, Northbridge, and Southbridge]

The fact, that memory is connected through the Northbridge is one of the shortcomings of the Intel Core Duo. This additional stage, which in fact is a connection node also for other elements causes high memory latency and limits available bandwidth (usually it is a shared link). Another problem is, that there are several compatibility problems, because it is impossible just to connect a Pentium CPU to the PCI bus. FSB expansion needs additional adapters – which in this case are North- and Southbridge.

This problem was solved on some of the AMD processors, which have memory controller on the CPU die and use HyperTransport\textsuperscript{19} as FSB replacement. With HyperTransport we gain two things: we can just connect together all HyperTransport aware hardware\textsuperscript{20} and there is a significant difference in throughput – also because HyperTransport is a dedicated link – available bandwidth on Northbridge is shared. Fast, scalable link that reduces total number of buses – this of course has positive impact also on the latency. Very important feature is, that HyperTransport is compatible with legacy PCI, PCI-E and PCI-X technologies.

<table>
<thead>
<tr>
<th>Max. clock speed (MHz)</th>
<th>133</th>
<th>266</th>
<th>333</th>
<th>800</th>
<th>1000</th>
<th>1400</th>
<th>2600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical speed in MB/s</td>
<td>4266</td>
<td>8533</td>
<td>10666</td>
<td>12800</td>
<td>14400</td>
<td>22400</td>
<td>41600</td>
</tr>
<tr>
<td>Comments</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>Hyper Transport</td>
<td>Athlon64, FX and Hyper Transport</td>
<td>Hyper Transport</td>
<td></td>
</tr>
</tbody>
</table>

\textsuperscript{19} More about HyperTransport and HyperTransport Consortium can be found on [http://www.hypertransport.org/](http://www.hypertransport.org/)

\textsuperscript{20} The so called “glueless” solution.
<table>
<thead>
<tr>
<th>Max. clock speed (MHz)</th>
<th>133</th>
<th>266</th>
<th>333</th>
<th>800</th>
<th>1000</th>
<th>1400</th>
<th>2600</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.x</td>
<td></td>
<td></td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opteron; HT + mc</td>
<td></td>
<td></td>
<td>3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is clearly visible, that Hyper Transport links offer much higher frequency, which yields higher transfer rates. Once again, please note, that HyperTransport link is a direct link!

What might be very interesting from the research point of view is the fact, that Intel Core Duo is a 32-bit processor, whereas AMD offers dual-core CPUs supporting x86-64 architecture, which was also adapted by Intel and named EM64T (Xeon, 5x6 Prescott P4 series). Intel Itanium\(^{21}\), which itself forms a new IA-64 architecture unfortunately cannot be taken into account, both because of high prices and poor support\(^{22}\) for the x86 (32-bit) legacy code. More about architectures will be covered in the next chapter.

**HyperTransport provides better performance – without any doubts. In the case of image processing or other processing that requires large streams of data, this kind of design should be considered first when buying a machine.**

### 64 bits to happiness

64-bit market seems to be growing day by day, but still 32-bit applications form vast majority and will not be forgotten for a long time\(^{23}\). Currently, when prices of x86-64 processors drop, for most research applications 64-bit programs should be the future. What is very important – it is up to the operating system to decide whether legacy 32-bit code should be supported or not – thus there is no way we can loose our favorite 32-bit applications.

**Gentoo Linux gives a possibility of running in both modes: long or mixed. Other distributions usually come in multi-lib mode.**

### Why 64 is better than 32?

Going back in time to 16-bit 80286 processor – at that time Intel added to it (16-bit processor!) a flat 32-bit addressing mode with 32-bit registers. Similar situation is in case of x86-64: what is new in AMD64 is that it supports 64-bit (flat) addressing and has 64-bit registers while still being able to execute 32-bit applications. AMD64 architecture enlarged width of all registers (GPR, PC) and additionally added 8 new GPRs and doubled number of SSE2 registers (XMM registers; from 8 to 16). In cases, when compiler is aware of these additional registers it can significantly speed up the execution, as the need for saving and restoring data is much smaller (register starvation occurs less frequently). Of course, this is still far away from most RISC implementations, which offer 32 GPRs and even further from IA-64 which has 128 GPRs.

---

21 “Itanic” is another name given to Intel Itanium by “The Register”. It is a direct reference to RMS Titanic which sank in 1912. What “The Register” wanted to say, is the fact that work on Itanium does not give back expected results and costs enormous amounts of money.
22 Itanium 2 should overcome this problem (according to Intel and HP).
23 Or until Microsoft decides to do so.
Of course, x86-64 has bigger physical \(2^{40} = 1024 \text{ GB}\) and virtual \(2^{48} = 262144 \text{ GB}\) address space. When compared to 32-bit mode, which allowed 32-bit \(2^{32} = 4 \text{ GB}\) virtual and 36-bit \(2^{36} = 64 \text{ GB}\) physical addresses it is a big difference. This enables to process large amounts of data and referencing them directly from the memory (like memory mapped files – we omit disk I/O which is measured in milliseconds, whereas memory access is measured in nanoseconds – that is six orders of magnitude faster!).

There is also another feature that was introduced by AMD and that should increase security: the NX bit (no-execute). Currently on IA-32 platforms when a buffer-overflow occurs it is possible under some circumstances to execute malicious code (that can also come from the remote site). That is because there is no low-level control over memory pages. NX bit determines if a page contains executable code – thus – in case of mentioned attack it should simply generate a memory violation error. This feature was also implemented by Intel in some Pentium4 processors which have PAE\(^{24}\).

From the architectural point of view, x86-64 seems to be an immediate solution for some of the IA-32 problems (IA-32 itself was difficult to modify because of the backward compatibility it had to offer; this problem is also called “golden handcuffs problem”, because IA-32 is a marketing success, but the design has to obey some old-fashioned rules, that simply make it bad).

### 64 bits of problems

To be able to run applications in the so called **long mode** (64-bit mode), 32-bit applications have to be recompiled. Some of them of course will not compile, and this might be a big problem, especially for business oriented people. Linux of course supports x86-64 architecture (especially AMD64). The biggest players on 64-bit Linux market are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Platforms</th>
<th>Website</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gentoo</td>
<td>Intel compatible, PPC (32 &amp; 64), Alpha, 64-bit, Sparc</td>
<td><a href="http://www.gentoo.org">http://www.gentoo.org</a></td>
<td>For advanced users.</td>
</tr>
<tr>
<td>Open SuSE</td>
<td>Intel compatible, 64-bit</td>
<td><a href="http://www.opensuse.org/">http://www.opensuse.org/</a></td>
<td>Open Source version of SuSE.</td>
</tr>
<tr>
<td>Fedora Core</td>
<td>Intel compatible, 64-bit, PPC</td>
<td><a href="http://fedora.redhat.com/">http://fedora.redhat.com/</a></td>
<td>Open Source version of Red Hat</td>
</tr>
<tr>
<td>PLD</td>
<td>Intel compatible, PPC, Alpha, Sparc, 64-bit</td>
<td><a href="http://www.pld-linux.org/">http://www.pld-linux.org/</a></td>
<td>For power users.</td>
</tr>
<tr>
<td>Mandriva</td>
<td>Intel compatible, PPC, 64-bit</td>
<td><a href="http://wwwnew.mandriva.com/">http://wwwnew.mandriva.com/</a></td>
<td>One of the first distributions fully</td>
</tr>
</tbody>
</table>

---

\(^{24}\)Physical Address Extension – enables up to 64GB of memory to be used on x86 computers.
<table>
<thead>
<tr>
<th>Name</th>
<th>Platforms</th>
<th>Website</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHEL</td>
<td>Intel compatible, 64-bit, others</td>
<td><a href="http://www.redhat.com/">http://www.redhat.com/</a></td>
<td>supporting EM64T processors</td>
</tr>
<tr>
<td>Slamd64</td>
<td>64-bit</td>
<td><a href="http://slamd64.com">http://slamd64.com</a></td>
<td>Unofficial Slackware port.</td>
</tr>
</tbody>
</table>

X86-64 is a new architecture and it will still take some time for the companies and development communities to write or port their software to support long mode. Despite this fact, it is highly visible that 64-bit is the future and all applications sooner or later will be forced to support it.

There are also some difficulties and limitations when working on x86-64 Linux. It might be the case with sound drivers, for which additional emulation libraries have to be installed. If the main application is going to work with video streams, then it might be also difficult to work with certain file formats, as some CODECs are still not available in their 64-bit version.

There might be also some difficulties with newest graphic boards – but in this case there is a huge community of commercial developers that is working on improving them.

The general rule should be: if one plans to buy a Linux with support or from a certain vendor, it would be good to check if one or both sites provide some kind of certificates (that “this Linux can run on that box”). If the box will be assembled manually, then each piece of hardware should be carefully checked if it is fully supported. It might be too late to discover at some point, that the hardware that was ordered will not work as we wanted. Good approach is also to check Bugzillas of certain distributions that offer 64-bit Linux versions and search them against both hardware and software pieces one plans to use.

### 64-bit computational research box

Having in mind all previously mentioned features and possibilities that come with the x86-64 architecture makes it a good choice and target platform for applications that are CPU and memory consuming. For the 2D FFT, where the transform size grows very fast and requires a lot of processing power and memory, this platform with all its advantages is currently the best solution that can be obtained for decent prices. It is also a perfect solution for a dedicated computational machine or a server as most of the software available for them includes threads (if applicable).

25 At the time of writing – 2006.
Generally speaking, the biggest computational power is inside Intel processors, especially when using Intel Compiler. It is a perfect solution for processes that have to perform many operations on a data set that does not change frequently over time – thus, it might not be the best solution for real-time systems. AMD with HyperTransport can be considered to be a very good solution for real-time system, where large amounts of data have to be transferred back and forth between CPU and memory. Even though, AMD's computational capabilities are smaller, the speedup will be visible.

**Inside and close to CPU**

This chapter will try to provide some background about the most important enhancements. Most of them, like pipelining or cache memory were invented some time ago and now are implemented in almost every computer (previously they were considered to be luxury enhancements). Additionally, every CPU comes with extensions: some kinds of magic abbreviations, that when properly used can give “golden” results. Survey will include also corresponding flags that can be used with for example the gcc\textsuperscript{26} compiler.

**Importance of this chapter**

In most cases our feeling about the processor market is as follows: “there are many CPUs, some of them come from the same series, thus, they must be the same”. No! They are not. There are several code names that are assigned to processors which try to reflect core’s capabilities and possible other enhancements. Many people might be surprised if they knew that for example Pentium 4 has 16 different versions\textsuperscript{27}. All these versions differ a lot – starting from a P4 that is a plain 32-bit CPU and ending on the version that supports EM64T with NX bit and SSE3. It is good to know what to buy and how it will affect further development – at least in theory.

**Stage 1: pipelining**

Idea of pipelining in its basic, virgin form is quite easy and can be presented on the example of a production line or very famous “laundry example”. In most cases concept of the pipeline is known – but this is just the pick of a mountain. There are several other things that can have impact on the pipeline and its behavior – and this of course will be visible in the performance gain (or loss) that pipeline gives. An ideal pipeline consisting of n stages should give performance gain close to n – but of course, the more stages, the more problems and resources that have to be used (each stage needs its exclusive set of elements).

---

\textsuperscript{26}GNU Compiler Collection. Learn more at http://gcc.gnu.org/.

\textsuperscript{27}At the time of writing – 2006. Number is just an estimate – there might be more versions.
Number of pipeline stages

This of course depends on the implementation. As it was mentioned – the more stages the better performance gain, but also the more difficult implementation and higher penalty for stalling the pipeline. This is especially important in case of branch instructions and data loads, which can break whole pipeline (causing pipeline flush). Another problem arises when instructions depend on each other – this hazardous situation also has to be solved efficiently by designers.
There are several methods to minimize the latency. Good example can be Pentium 4 processor, which implements speculative pipeline with around 21 stages. But this number changes and for example all Pentiums 4 code name Prescott have 31 stages pipeline. Opposite situation is in case of RISC processors, for example MIPS offers 5 stages pipeline. Natural question would be then, where is the border? What is happening in P4 processor is a high-level engineering, which tries to get most power of available hardware. Intel engineers put much more hardware to implement this aggressive queue – otherwise a branch instruction could simply flush pipeline every time it was not taken. But this is also result of the way Intel treats instructions, which are of variable size (on the contrary to MIPS, where everything is aligned). An instruction is decoded and split into several micro operations (it is said, that Intel works like RISC being a CISC), which are further organized (and can be reordered – OoO mechanism) into this multi-stage pipeline. Additionally Intel uses trace cache and register renaming to minimize penalty when it comes to recover the pipeline. Compilers can also influence the way pipeline will be executed by scheduling instruction in a pipeline-friendly way – this includes also branch prediction, which nowadays is present also on CPU dies. Both AMD (new, x86-64 ones) and Intel processors have register renaming and OoO (out-of-order execution) implemented in their pipelines.

Current CPUs usually have distinct pipelines for floating-point units (or different number of stages). Number of stages also differs among manufacturers and CPU types: Intel (largest, up to 31 stages), AMD (12 stages for integer and 17 stages for floating-point and MIPS (5 stages).

Pipeline increases throughput, and this increase is proportional to the number of stages. Involved latency, additional hardware and complex structure of the pipeline is worth the effort, because the overall result still makes it much faster that a non-pipelined solution.

The general rule is, that the more stages – the more problems. But honestly, it is of second importance to pay attention to number of stages in the pipeline. For sure, it would be immediately visible if a CPU was running without pipeline.

Summarizing, purpose of this paragraph was just to provide a few pieces of information about pipelining and its current forms.

**Floating-point unit**

Early CPUs did not contain floating-point units. These calculations were handled usually either by a co-processor (which itself was an addition) or were emulated (either in software or were implemented as a microcode). Currently the tendency is to have more than one FPU with shared or separate pipelines (integer, floating-point pipelines). Early superscalar processors that did not implement OoO (out of order execution) had to have separate pipelines. Now it is not a problem, since instructions can be easily reordered (but there might be a different number of pipeline stages).

This chapter is extremely important, as these extensions solve the

---

28That is why pipeline is covered – this parameter changes over the time for the same processor name!
biggest problem that exists on x86 and x86-64 architectures, and that is “register starvation” problem. FPU extensions usually provide set of registers and additional functions for performing specialized tasks on them (which usually tries to exploit higher level of parallelism). It is important because these extensions can be used directly, as gcc compiler offers set of built-in functions for accessing them (and can also generate FP code automatically – similarly as icc compiler).

Many processors – one floating point standard (IEEE 754)

IEEE 754 floating-point standard is the one that makes it possible to port floating-point applications to different processors and architectures. Currently a new revision is being developed (IEEE 754r) which has some very interesting functions and adds quad precision format – but this will be covered later on.

There are many details about the floating-point standard that are out of scope of this text, but some of them can be referred to as basics of floating-point definition, as they can cause one of mentioned things to happen: overflow or underflow. Overflow is a sample case, when a number is too large to fit into the data format (the same for integer operations), whereas underflow exists only in the floating-point world. It means, that the number is too small (negative exponent is too big) to fit into the data type. This is very important from the computational point of view. Why? Mainly, because current FP standard offers two data types which are known as float (single precision, 4 bytes) and double (double precision, 8 bytes). For medium and large sized two dimensional FFTs, proper data type can be beneficial, as then more data can fit into the cache – and this of course will minimize total number of cache miss events, which will result in faster execution (as it was mentioned before – from processor's point of view memory is slow). Not always is it possible to predict range of numbers that will form the input, and there are different demands concerning precision of calculations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>4 bytes (32 bits)</td>
<td>8 bytes (64 bits)</td>
</tr>
<tr>
<td>Range</td>
<td>$\pm10^{-38} - \pm3 \cdot 10^{38}$</td>
<td>$\pm2 \cdot 10^{-308} - \pm2 \cdot 2 \cdot 10^{308}$</td>
</tr>
<tr>
<td></td>
<td>($\pm1.17 \cdot 10^{-38} - \pm3.40 \cdot 10^{38}$)</td>
<td>($\pm2.25 \cdot 10^{-308} - \pm1.79 \cdot 10^{308}$)</td>
</tr>
<tr>
<td>Internals</td>
<td>1 bit for sign, 8 bits for the exponent and 23 bits for the fraction; implicit 1 added to the fraction gives in fact 24 bits</td>
<td>1 bit for sign, 11 bits of the exponent and 52 bits for the fraction; implicit 1 added to the fraction gives in fact 53 bits</td>
</tr>
<tr>
<td>Representation</td>
<td>$(-1)^s \cdot (1 + fraction) \cdot 2^{exponent}$</td>
<td></td>
</tr>
<tr>
<td>Exponent</td>
<td>Biased (127)</td>
<td>Biased (1023)</td>
</tr>
</tbody>
</table>

Both single and double precision formats support rounding schemes and special events like NaN or inf. These events as well as zero and denormal numbers (subnormal\(^{29}\)) are encoded in special ways, which makes the

\(^{29}\)Subnormal numbers are non-zero numbers that are smaller than the smallest normal
standard much more flexible.

As it was mentioned before, revision of the current floating-point standard will involve quad precision type (128bits) and one very important instruction, that currently is present in PowerPC and Itanium processors: **FMA**. Fused multiply-add or fused multiply-accumulate is instruction that makes both things much faster than multiplication followed by addition.

\[
FMA(a, b, c) = a \cdot b + c
\]

Whenever possible this instruction should be involved, as it greatly simplifies and speeds up computations on complex numbers, which of course are mandatory for FFT implementation (and can be almost directly used for calculating the so-called Sande-Gentleman butterfly).

From the design point of view, all FPUs are of course optimized for the double precision type, thus theoretically there should be no difference between time needed to compute the same thing on double or float type. From the other hand, there will be substantial difference in memory access time. 1Mpx picture in frequency domain will occupy 16 megabytes of RAM memory (for double, whereas single precision just half of it). There is no way for such array to fit into cache memory – thus everything has to be fetched from the “slow” memory. Good designs require good compromises: precision vs. speed.

**Limitations within the standard are more important in terms of the design, as they determine which type will be used etc.** For large transforms it might be beneficial to consider the float type, as it occupies half of the size that double arrays occupy – and this of course will result in faster access size.

**x87**

In the beginning floating-point math was available in form of a coprocessor. In current 64-bit implementations x87 is available to support execution of legacy 32-bit floating-point code. Of course there are also some advantages of running x87 code in the long mode, and in case of AMD64 these are:

- access to 64-bit address space
- RIP addressing mode is available.

RIP-relative addressing mode is a new mode offered by AMD64 processors, which makes loading of the PIC code more efficient, as instructions can reference data relative to the instruction pointer.

---

Quick note from gcc manual:

---mfpmath=unit

Where unit can be:

- 387 (default, will run everywhere)

numbers (numbers close to 0). Subnormal numbers were firstly implemented by Intel and then they became IEEE standard. On some hardware operations on subnormal numbers are not implemented in the hardware and rely on software solutions (or others) which makes computations much slower than on normal (normalized) numbers.

30 Name comes from Intel’s FP chip, whose name was ending with 87.
31 Position Independent Code.
- sse (utilize SSE extensions – needs additional options!)
- sse,387 (utilize both at the same time; experimental, should not be used for production code)

On x86-64 compiler SSE/SSE2 are enabled by default. On i386 compiler 'sse' option requires also –msse or –msse2 switches and –march=CPU has to be stated.

X87 contains 8 stack registers with the maximum precision equal to 80 bits.

As long as sources of the project are available it should be not a problem to compile a binary version that will run on a certain piece of hardware. Problems arise, when one is supposed to distribute binary version of the software. This of course means, that there should be several binary versions available or – at a cost of performance – floating-point code should use only x87 FPU.

To provide a real example: lame, an mp3 file encoder was used to encode files containing random data on two machines: one having Intel Xeon EM64T 3.4GHz and the other one Intel Pentium 4 HT 3.0GHz (both had 1024kB of cache memory). The second one was performing a bit better – just because it used MMX/SSE/SSE2 extensions.

(Marketing) Extensions

All these names have two purposes: to sell more processors and to give some overview about CPUs capabilities\textsuperscript{32}. Currently each processor contains so many extensions, that new “marketing” names are advertised only for revolutionary technologies (like HT for example). Since almost all CPUs have these extensions, then why bother? Knowledge is power and power can directly turn into performance of our code – even without writing a single line in assembly. The trick is to know how to utilize all power that comes with the compiler – for example with gcc.

Most of the extensions that will be presented belong to the group of SIMD instructions, which simply means Single Instruction – Multiple Data. The main purpose is to be able to process large amounts of data (applicable in DSP and graphics), which should more or less resemble in practice that what happens in a vector processor\textsuperscript{33}. The outcome is much higher level of parallelism than in case of an ordinary superscalar processor. Of course, SIMD requires more registers, but not always were they present (to lower the overall cost). That was the case of MMX extension (integer arithmetics), which added 8 new registers, that were in fact aliases of existing x87 registers. Thus, making at the same time floating-point operations will simply result in “register fight”. The other – maybe not disadvantage – but difficulty that comes with SIMD is the fact that data has to be perfectly aligned, and this is usually troublesome\textsuperscript{34}.

---

\textsuperscript{32} Sometimes extensions’ names are totally meaningless, like MMX.

\textsuperscript{33} Processor that operates on several data items at one time.

\textsuperscript{34} To make it even more complicated, on Pentium and Pentium PRO double and long double have to be aligned to 8-byte boundary, whereas on P3 they should be aligned to 16-byte boundary.
SSE
Streaming SIMD Extensions were a major enhancement when compared to MMX that offered only integer operations within limited scope (MMX registers were in fact FPU registers). SSE added 8 new 128-bit registers (XMM; each one packs 4 floats) that could be operated independently of the MMX set. In order to use SSE it has to be explicitly enabled. Of course, as the name states, XMM registers are SIMD-aware. SSE support was added later to AMD processors (starting from Athlon XP).

SSE2
At some point Intel enhanced SSE and created SSE2 – that was a major step forward. Below is summary of differences:
- reuses XMM registers, thus from the binary point of view operating vector is the same
- adds double precision floating-point operations
- enables to work with almost any integer type: 8/16/32/64 bits
Support for integer types made it possible to avoid switching to MMX mode – thus floating-point and integer operations can be mixed together.
As for now it is possible to use set of built-in calls (offered by gcc) and make use of both SSE and SSE2. It is also possible to automatically generate code for SSE/SSE2 extensions (supported by both gcc and icc).

SSE3
Newest set of SIMD instructions adds functions that can be used exclusively for DSP applications. Also operating on the register file is more flexible – with SSE3 it is possible to access the register horizontally (SSE/SSE2 – only vertically). There is also one function that improves efficiency of the pipeline – it enables to convert floating-point numbers to integers without explicit change of the rounding mode. SSE3 was firstly introduced by Intel – currently is present also in AMD CPUs (all AMD64 ones) and VIA C7 (some models only).

SSE instructions are the ones that can give big speed up for floating-point and integer oriented calculations. Whenever possible, they should be used.

3DNow!
Primarily 3DNow! was supposed to be MMX-enhancement (and AMD's corresponding MMX technology), as it added support for floating-point calculations, similar to SSE instructions that were later added by Intel to Pentium 3 processor. 3DNow! was also aliasing FPU registers (like MMX), but could pack only 2 floating-point numbers instead of four (in SSE). Another very interesting thing was that 3DNow! supported horizontal operations on a register file, which were added with SSE3.
Currently on AMD processors that support both SSE/SSE2 and 3DNow! theoretically it should be possible to execute at the same time both SSE and 3DNow! code. As it was mentioned before (notes about gcc), it is very hard to accomplish.

Later on AMD created enhanced 3DNow! Which was supposed to perform tasks like SSE extension. This was achieved later, when 3DNow! Professional was created (starting from Athlon XP).

**Similar situation is in case of 3DNow! In both cases (integer and floating-point math) if possible, it should be used.**

*Inside the memory hierarchy*

Many times it happens, that people become “gigahertz slaves” - it means, they measure performance of a computer by means of the clock frequency. Usually the higher the clocking frequency the better the performance – but not always as people expect it to be. One should not think that there is a linear dependency between both. Many things depend also on the most important link, and that is connection to the main memory (RAM). Since CPUs are much faster than RAM, it is obvious, that even the fastest processor will not perform good if every time it has to wait for data to arrive from the slow memory. There were many attempts that were trying to solve this problem, but the only solution for minimizing traffic on the bus is to add an intermediate stage, that will prevent the CPU from occupying the bus and accessing memory directly. This is the purpose of the cache memory. Its presence in the system is transparent to the programmer (from French, “caché” means “hidden, good place to hide”). Currently cache memory is put everywhere, where the cost of obtaining data is rather expensive. Capacity of cache memory is rather small, as they are very expensive (SRAM technology; access time around 0.5-5ns). Below there is the so called “memory pyramid”, which presents different memory hierarchies and defines differences between them, which are: access time, capacity and price.

There also other features that have impact on cache's performance, like organization, write or replacement modes.
Method

Cache is put in between the CPU and memory to minimize access time. This is due to the fact, that cache exploits two very important phenomena: spatial and temporal localities. If there is data that CPU wants from the memory, then it is likely that it would also like pieces of data that are nearby the referenced location – this is spatial locality. Temporal locality works according to the fact, that once a piece of data was referenced, it is likely that it will be referenced again, soon. To use localities efficiently, each block (or line) of cache contains more data than it was needed. Whole traffic between CPU and memory goes through cache memory. If an item is present in the cache memory, then we have a cache hit event – if it's not then it is cache miss, which unfortunately causes the following actions to be taken:

- since the entry was not present in the cache it should be fetched from the memory
- access procedure has to be started once again from the beginning

Bigger cache size or different organization can significantly decrease number of cache misses. These are the so called “cache friendly” programming rules, that are very strict, but in many cases can be applied:

1. Loops should be small – if they take large amounts of memory for the code then of course a loop won't fit into the cache. Amounts that can be put into cache are really small, around 2-4KB.
2. Calls to functions that reside outside current area of code should be performed less frequently than calls to functions that are close to the current area of code.
3. Memory should be accessed in the same areas. This might be of great interest when designing an object oriented application: code is cache friendly if one tries to access class’ fields together at one time (spatial). The same can be applied to arrays: when they are small and accessed sequentially then it is possible to notice performance increase.

Regarding point number 3 and arrays: there is also another way of storing two-dimensional arrays, which is the so-called set of of pointers to pointers (or in C words my_type ** variable). Referencing items from such a pointer is as easy as accessing a standard two-dimensional array, that is: variable[x][y]. It is a very bad design for time critical applications as:

- for array of size NxM we have N vectors of size M; in total “sizeof(my_type *) + N*M*sizeof(my_type)”
- deallocation corresponds to freeing firstly N vectors of size M and then the pointer to vectors
- rows of the array are in chunks, thus spatial locality will not work as it should
- it is much more difficult to align data in such form (if one wants to use SIMD extensions).

There are also bad habits, which can generate a lot of cache miss events, and thus will slow down execution of the program:
1. Large arrays that cannot fit into cache and that are accessed sequentially and several times will of course force CPU to access memory more often. A possible solution to this problem could be to split an array into smaller parts and process them one by one – but this cannot be done always.

2. In case of linked lists where one tries to access a piece of data in each node for each record – this can increase miss ratio. It is much better to process all fields from a record at a time and then move to the next node.

3. FIFO queues: the first-in element is usually the oldest entry in memory, thus according to LRU this will be the first discarded entry.

The most important is the first point. In our FFT problem, there is no way that we can void using large amounts of data. But there is another solution for this problem which was mentioned in the FFT algorithm section: row- or column-major formats. Whole matrix is packed into one dimensional vector which implies sequential access. This provides better usage of both localities.

There is still one more problem: how about writing to the memory? In general there are two strategies:

- write-back
- write-through

Write-through tries to keep both cache and memory consistent, and updates the block in both places. For write-through memory write operations should be collected to a write buffer and afterwards, when the buffer is full or other conditions are fulfilled, data should be written to the memory. On the contrary, write-back updates only block in the cache, which is further written to the main memory when the block is replaced.

Write-back scheme is harder in implementation than write-through, but can speed up write operations especially when they are generated by the CPU faster than they can be processed by the memory.

**Processors will have the best performance for highly complicated tasks – but they are not optimized for dealing with huge bandwidths or streams of data.**

**Currently it should not be possible to buy a PC computer that has a CPU without cache memory. Such CPUs have very attractive prices, but in turn offer really poor performance.**

**Sizes, levels, organization**

Due to high prices of SRAM technology, cache memories are usually small. Currently some CPUs (especially Intel Celeron series) are shipped with much smaller cache memories, and that of course has direct impact on the overall performance of a computer. Nowadays it is usually between 256kB and 1MB (most cases), but there are memories as small as 128kB (Intel Celeron, P4 based) and as large as 2MB (some laptops). In many cases, cache memories are put on the die together with CPU and are organized into levels (usually 2). The second, L2 (or other, higher level) cache level is used every time a cache
miss is encountered in the 1st level cache (L1 or corresponding lower level). Then the miss penalty is equal only to the access time to the L2 (or other level) cache memory which is of course much smaller than DRAM access time. Sometimes too many cache levels can increase the latency, like it was in case of the Itanium processor. Third (L3) level cache had such a big latency, that there was no big difference between accessing main memory and the third level cache (L2 was just 256KB and L3 was 9MB).

Usually L1 cache is split into two parts: one caches data and remaining part is dedicated for instruction caching (Intel calls it “micro op cache”, from the term micro operations). In case of Intel processors L1 cache is usually from 20 to 30KB (and this number is further split into data and instruction caches; proportions between both differ), whereas on most AMD processors L1 cache is rather big (Athlon, Duron, Opteron series) – 128KB (64KB for data and 64KB for instruction cache). For dual-core CPUs 128KB is per core which further connects both L1 caches to one L2 cache.

Apart from the cache size, there is one more thing that can influence performance of the cache subsystem: organization. It is said, that cache memory is n-way set associative if it is possible to map a block to n different places. A case, when a certain memory location can be placed only in one place of the cache is called direct-mapped cache. Opposite situation occurs, when a memory location can be placed anywhere in the cache – that is fully associative cache. Usually the best solution is a compromise, which in this case is some kind of n-way set associative organized structure. The more flexible placement of blocks, the more complicated search will be. For example, for a 2-way set associative cache each memory location has to be checked in two different places, for a 4-way in 4 places and so on. The best result can be achieved only if for n-way set associative cache there are n comparators, that can simultaneously check occurrence of referenced memory location. This means more hardware, and more hardware usually implies higher latency.

There is one question that still remains open: which block should be replaced on cache miss? It can be either a random block, which in this case might be wrong choice, as usually cache is small enough to implement LRU algorithm, that will replace mostly unused elements on a cache miss. Random mechanism might be interesting for TLB which is usually fully-associative, but anyway, newest Intel and AMD processors have either pseudo-LRU replacement or implement LRU with some other round-robin mechanisms.

For a machine that is supposed to deal with big bandwidths, and this is definitely our case, cache memory should be as big as possible. A CPU that has less than 1MB of cache should not be considered at all. 1MB or 2MB – these are values that should offer best performance for a decent price. There are examples of big multilevel caches, that unfortunately were not performing as good as they should. This is the case of Itanium processors, which has 256KB of L2 cache and 9MB (!) of L3 cache – but the latency involved by the third level is so high, that it makes the difference in access time between cache and memory very small.
One bus, different banks

There is another technique that can speed up memory fetch operations, which is called memory interleaving. Instead of having one big memory we have several memory banks. Of course, it is not enough to split the memory (after splitting it still can be used as segmented memory). Let say, that the memory we have is of N size and there are four available banks. Memory locations 1, 2, 3, 4, 5, 6 will belong to `addr modulo 4` bank, thus:

Address: 1 --> 1 mod 4 = bank 1  
Address: 2 --> 2 mod 4 = bank 2  
Address: 3 --> 3 mod 4 = bank 3  
Address: 4 --> 4 mod 4 = bank 0  
Address: 5 --> 5 mod 4 = bank 1  
Address: 6 --> 5 mod 4 = bank 2

This enables CPU to issue more commands than memory can process at a time. This is another example how it is possible to increase throughput by implementing some kind of pseudo-parallel operations. Memory interleaving can be viewed similarly as pipelining, which still needs a certain amount of time to finish one single operation.

Cache memory minimizes traffic on the bus between CPU and memory, but of course, the best results can be achieved for memories that have the highest clocking frequency. It is possible to connect memories to a Northbridge which have smaller frequency than FSB, but of course, this might not be a good idea for a fast research box. Big and fast memories are expensive, but in most cases it is worth investing in them.

Additionally, for such a purpose one should consider HyperTransport aware hardware, which is a very good, “glueless“ replacement for FSB in Northbridge.

Alternative solution – GPGPU

Popularity of gaming and constantly increasing requirements from both customers and graphic industry caused, that today we can buy a decent graphics board equipped with a GPU capable of doing several hundred billion\(^35\) floating-point operations per second for just a few hundred dollars. It sounds promising, especially that GPU performance increases much faster than in case of normal CPUs (Moore's law says about the ratio equal to 2, whereas for GPUs it is around 2.4). Currently almost all GPUs are programmable, thus it should be a bit easier to write general purpose programs for them. Nevertheless, GPU programming is not easy and the environment is highly constrained. Despite this fact it is worth trying, especially that most of image processing algorithms can be implemented on

\(^35\)To ensure, that it is not a type-error: several hundred billion floating-point operations per second.
GPUs.

**Introduction to GPU world**

GPU is not a general purpose processor, thus it was possible to use less control units and put more computational logic. Due to this fact, there are many functional units that introduce very high level of parallelism, because not only are they able to perform the same operations on multiple sets of data (SIMD - vector), but also they can perform different operations on different data at the same time (MIMD - vertex). Level of parallelism present on GPU much higher than the one present on contemporary CPUs (SIMD is the highest level of parallelism achievable on contemporary Intel and AMD CPUs).

**Streams**

Different architecture of GPUs has also put some requirements on the input data, thus it is said, that GPUs are stream processors. Stream is nothing else than a set of data of the same type. The larger the stream, the higher the probability, that it will be possible to use all functional units in parallel. To support this type of computing, all GPUs are optimized towards big throughputs.

As it was mentioned before, contemporary GPUs have fully programmable functional units. Before that, they were specialized in performing certain tasks. Specialization – not optimization – made it possible to achieve outstanding results, but of course, at a cost of no flexibility.

During computations streams are organized into a pipeline. Each computational stage is performed by a kernel (functional unit performing some kind of operation). These pipelines are usually deep and consist of several stages.

**Specialization vs. optimization - datapath**

As it was mentioned before, much better results can be obtained when computational unit is specialized in performing computations rather than optimized. Thus optimizations should be treated as subset of specialization. Conclusion is quite straightforward: a standard CPU is a purely sequential machine, that is in most cases supposed to deal with one data item at a time, and is mostly optimized for low-latency (not throughput) operations. Of course, CPUs contain mostly general-purpose hardware that is not specialized in performing certain tasks as CPUs have to provide certain level of flexibility. GPUs stand on the other side of barricade and are both specialized and optimized for parallel execution and maximum throughput. In other words – task- and data level parallelism are key features of GPU datapath.

---

36Standard CPUs in special conditions can sustain two floating-point operations at a time. This is easily achieved by GPUs, and even that was further beaten by GF 5800 (2.66) and finally by GF 6800, which offered 6 sustained floating-point operations at a time.
**FFT on GPU**

There are ready made implementations of FFT on GPU which rely on GPGPU available SDKs. The biggest advantage of FFT on GPU is that in the mean time CPUs can do other job. Nevertheless, it should be stressed out that FFT on GPU can be performed only in single precision (fully compatible wit IEEE single precision floating point standard).

Last part of the report presents results of measurements, which also include FFT on GPU. Measurements were performed under Windows on a demo program written by Kenneth Moreland and Edward Angel. Movies from these tests are present on attached CD.

GPGPU is not a brand new topic. It was introduced some time ago, but nowadays becomes more and more popular due to the fact, that it is possible to perform general-purpose computations on them. The biggest source of information about performing general purpose computations on graphics hardware can be found on GPGPU project webpage ([http://www.gpgpu.org/](http://www.gpgpu.org/)). Site contains also slides from GPGPU conferences, code samples and other useful resources (news, forum etc.).

Due to high complexity of the problem, no other work was done on GPGPU and FFT on GPU in this project (except testing ready-made FFT on GPU code).

All code samples and pieces of information come from GPGPU project's website.
Part III - “If you torture the data enough, it will confess.”

Ronald Coase

Introduction

This report is a summary of the whole project. It will heavily rely on the knowledge introduced in two previous parts. Last chapters will provide results from measurements of different algorithms and memory bandwidth tests.

Project formulation

“FFT in Image Processing: measurement, implementation, parallelization and computer architecture”

Goal of the project was to speedup 2D FFT calculations in MoInS software, being part of the PACO Project, developed also at Aalborg University Copenhagen. Changes should be transparent to programmers and should not interfere with already existing code, thus the task was mainly to create bodies of \texttt{fft()} and \texttt{invfft()} routines. One of the requirements was also to create a portable code, which could be run on most platforms, including x86_64.

Milestone plan

1. Introduction to theoretical aspects of the solution
   1. Mathematical background behind FFT
   2. Software issues (introduction to MoInS)
   3. Hardware issues
2. Analysis of possible solutions: either DIY or reusing available FFT libraries
   1. Performance vs. portability – complexity of creating optimized and portable code – SIMD memory alignment and other problems
   2. Implementing 2D FFT algorithms; creating algorithm tournaments/other solutions
   3. Benchmarking available libraries and focusing on implementing “neat and easy” interface for one of them
3. Measurements
   1. Methods, precision and implementation
   2. Current FFT implementation in MoInS
   3. Test case benchmarks
4. Demo implementations – non configurable FFT routines
Software development process was very similar to the Unified Process, which is a common practice when developing large, Object-Oriented applications. It is also suitable for smaller projects, as it enables to change requirements during the project development time. It also includes excessive testing after each milestone. To summarize, each development stage (milestone) consisted of:

- requirements phase (what should be implemented)
- analysis and design (possible solutions)
- implementation (demo and milestone implementations)
- testing
- deployment

This means, that development process of the project was incremental (successive parts of code were added to the main project) and iterative (many iterations within each milestone).

**Requirements**

The programming environment is constrained, as MoInS developers tend to create a monolithic software with its own set of libraries, so that external dependencies are as small as possible.

Other requirements as follows:

- portable and fast code that utilizes available CPU extensions (SIMD)
- 2D transforms of arbitrary sizes (power of two sizes)
- threads support (for SMP environments)
- run-time interface, so that it is possible to change library's behavior without recompiling the software
- mentioned before minimum number of external dependencies

To provide maximum portability and easy interface for compiling the library, it should utilize GNU building tools/process. This should also simplify further merging mfitt with MoInS software.

Important information about MoInS is that it is supposed to be a real-time system, thus code should be utilizing minimum resources. Since MoInS is also an image processing software, the code has to be optimized in terms of throughput (filtering in frequency domain, grouping etc., where each image has to be represented as a set of complex values).
This project is also one of the parts, that should help in achieving the first MoInS' milestone, which is “1Hz”.

**Problem solution**

Analysis of the complexity and time needed to implement fast and portable FFT routines proved, that more feasible solution would be to reuse one of the existing libraries. Selection method was rather easy and consisted of two steps:

- analyzing data available on the Internet (manuals, benchmarks, licenses etc.) - preselecting high performance libraries
- detailed analysis of preselected libraries: test implementations, benchmarks

As it will be shown in a moment, both steps can be merged into one, as there is FFT benchmarking software available that includes several FFT libraries. Software project is called benchFFT\(^{37}\) and was created by Matteo Frigo and Steven G. Johnson (both wrote also FFTW).

To be precise, only libraries that:
- are licensed under GPL
- have multi-threaded API
- are as much as possible vendor independent

can be taken into account (but it does not mean that they can't be benchmarked!). Below is a list taken from benchFFT of additional libraries, that should be installed by an user (if one wants to benchmark them).

<table>
<thead>
<tr>
<th>Free software:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFTW 2.x and/or FFTW 3.x (double and/or single precision): <a href="http://www.fftw.org">www.fftw.org</a></td>
</tr>
<tr>
<td>GNU Scientific Library: sources.redhat.com/gsl</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hardware/vendor-specific:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Math Kernel Library: <a href="http://www.intel.com/software/products/mkl">www.intel.com/software/products/mkl</a></td>
</tr>
<tr>
<td>Intel IPPS</td>
</tr>
<tr>
<td>AMD Core Math Library</td>
</tr>
<tr>
<td>Apple VDSP (Macintosh G4 and higher only)</td>
</tr>
<tr>
<td>IBM ESSL (AIX only)</td>
</tr>
<tr>
<td>sgimath (SGI/MIPS only)</td>
</tr>
<tr>
<td>SUNPERF (SPARC only)</td>
</tr>
<tr>
<td>DXML/CXML (Alpha only)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proprietary software:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical Recipes: copy .c and .f files into benchees/nr</td>
</tr>
<tr>
<td>NAG (Numerical Algorithms Group) Fortran Library</td>
</tr>
</tbody>
</table>

\(^{37}\)Project's website: http://www.fftw.org/benchfft/
After analysis of benchmark data, it was clear that FFTW is the library that should be used for further development. Similar results were presented on FFTW's website\textsuperscript{38}. In the next chapter there are results from running it on different machines (and results that are present on FFTW's website).

From the table presented above, only FFTW 3.x was benchmarked.

\textbf{BenchFFT}

\textbf{Method:} benchFFT was run on four different computers that were considered to be the most popular ones among people using MoInS. These were both 32 and 64-bit machines having Pentium and AMD processors. Because 2D FFT can be also viewed as performing 1D FFTs over both dimensions, results from one-dimensional test will be also presented. Non powers of 2 vectors are omitted, as they do not fit into performance requirements (besides, it is almost always possible to avoid using them).

Please note, that on SMP systems only one processor was used at a time!

\textbf{Florian}

\begin{verbatim}
uname -a: Linux florian-2 2.6.11.4-21.11-bigmp #1 SMP Thu Feb 2 20:54:26 UTC 2006 i686 i686 i386 GNU/Linux
arch: i686
Intel(R) Xeon(TM) CPU 3.40GHz
cache size : 1024 KB
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
        pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe lm pni monitor ds_cpl est cid cx16 xtpr
bogomips : 6733.82
(the same for processor number 1)
MemTotal:  4148604 kB

gcc --version: gcc (GCC) 3.3.5 20050117 (prerelease) (SUSE Linux)
gcc -v: Reading specs from /usr/lib/gcc-lib/i586-suse-linux/3.3.5/specs
g++ --version: g++ (GCC) 3.3.5 20050117 (prerelease) (SUSE Linux)
g++ -v: Reading specs from /usr/lib/gcc-lib/i586-suse-linux/3.3.5/specs
cc --version: cc (GCC) 3.3.5 20050117 (prerelease) (SUSE Linux)
cc -v: Reading specs from /usr/lib/gcc-lib/i586-suse-linux/3.3.5/specs
c++ --version: c++ (GCC) 3.3.5 20050117 (prerelease) (SUSE Linux)
c++ -v: Reading specs from /usr/lib/gcc-lib/i586-suse-linux/3.3.5/specs
\end{verbatim}

\textsuperscript{38}http://www.fftw.org/
uname -a: Linux mic 2.6.14-gentoo-r5 #1 SMP Fri Jan 20 17:06:25 CET 2006 i686 Intel(R) Pentium(R) 4 CPU 3.00GHz GenuineIntel GNU/Linux
arch: i686
model name : Intel(R) Pentium(R) 4 CPU 3.00GHz
cache size : 1024 KB
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe pni monitor ds_cpl cid xtpr
bogomips : 5991.37
(the same for processor 1)
MemTotal: 498368 kB
gcc --version: gcc (GCC) 3.3.6 (Gentoo 3.3.6, ssp-3.3.6-1.0, pie-8.7.8)
gcc -v: Reading specs from /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/specs
gcc --version: gcc (GCC) 3.3.6 (Gentoo 3.3.6, ssp-3.3.6-1.0, pie-8.7.8)
gcc -v: Reading specs from /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/specs
g77 --version: GNU Fortran (GCC) 3.3.6 (Gentoo 3.3.6, ssp-3.3.6-1.0, pie-8.7.8)
g77 -v: Reading specs from /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/specs
g++ --version: g++ (GCC) 3.3.6 (Gentoo 3.3.6, ssp-3.3.6-1.0, pie-8.7.8)
g++ -v: Reading specs from /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/specs
cc --version: gcc (GCC) 3.3.6 (Gentoo 3.3.6, ssp-3.3.6-1.0, pie-8.7.8)
cc -v: Reading specs from /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/specs
c++ --version: c++ (GCC) 3.3.6 (Gentoo 3.3.6, ssp-3.3.6-1.0, pie-8.7.8)
c++ -v: Reading specs from /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/specs
f77 --version: GNU Fortran (GCC) 3.3.6 (Gentoo 3.3.6, ssp-3.3.6-1.0, pie-8.7.8)
f77 -v: Reading specs from /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.6/specs
uname -a: Linux qtux 2.6.16-gentoo-r7 #2 SMP Fri May 19 18:48:58 CEST 2006 x86_64 Dual Core AMD Opteron(tm) Processor 270 GNU/Linux

arch: x86_64

Dual Core AMD Opteron(tm) Processor 270, 2GHz

cache size : 1024 KB

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt lm 3dnowext 3dnow pni lahf_lm cmp_legacy

bogomips : 4024.89

(the same for processors number 1, 2 and 3)

MemTotal: 4039724 kB

gcc --version: gcc (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
gcc -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
g77 --version: GNU Fortran (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
g77 -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
g++ --version: g++ (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
g++ -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
c --version: gcc (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
c -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
c++ --version: c++ (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
c++ -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
f77 --version: GNU Fortran (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
f77 -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
x86_64 Dual Core AMD Opteron(tm) Processor 285 GNU/Linux
arch: x86_64
model name : Dual Core AMD Opteron(tm) Processor 285, 2.6GHz
cache size : 1024 KB
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
        pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt lm 3dnowext 3dnow pni
        lahf_lm cmp_legacy
bogomips : 5232.45
(the same for processors number 1, 2 and 3)
MemTotal: 4039648 kB

gcc --version: gcc (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
gcc -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
g77 --version: GNU Fortran (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
g77 -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
g++ --version: g++ (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
g++ -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
c--version: gcc (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
c -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
c++ --version: c++ (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
c++ -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs
f77 --version: GNU Fortran (GCC) 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
f77 -v: Reading specs from /usr/lib/gcc/x86_64-pc-linux-gnu/3.4.5/specs

double-precision complex, 1d transforms
powers of two
double-precision real-data, 1d transforms
powers of two

double-precision complex, 2d transforms
powers of two
Results from FFTW's webpage

There are several different benchmarks available on the benchFFT website. Graphs are available on-line, thus most of them will not be presented here. We will focus our attention only on results coming from the same machine, but where source was compiled with two different compilers: gcc (GNU compiler) and icc (Intel Compiler). Results only for 1D and 2D transforms, powers of two.
Illustration 5: gcc, P4 2.4GHz

Illustration 6: icc, P4 2.4GHz
There are no big differences between both compilers, especially when they deal with high-quality source code. Also, there should be no doubt about the
fact, that Intel Compiler seems to be better in optimizing the code. In both cases if possible, it is advised to measure execution time of binaries generated by several different compilers.

**Project structure**

Main goal was to integrate FFTW with MoInS. Since FFTW gives lots of possibilities, the other aim was to design a fast and user-friendly interface for configuring FFTW without recompiling even a single piece of code. Created library was named **mfft** (there is also other library named mfft, but that is just coincidence), which should resemble "my fft", as the purpose of it was to give full control over the FFT calculation process.

Diagram below presents the structure and relations between different software parts. Dashed circle contains definitions that can be changed between program runs – solid lines represent objects, that have to be recompiled if modified.

MFFT does not encapsulate FFTW's types – this was done on purpose, as mfft is only an interface. It should be also relatively easy to modify mfft's source code and invoke different planner or add new options to the configuration file.

Since one of the requirements was to have minimum (or even no) external dependencies, some pieces of software were integrated with mfft, and these are:

<table>
<thead>
<tr>
<th>Software</th>
<th>Purpose</th>
<th>License</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCL</td>
<td>Configuration file parsing</td>
<td>GNU/GPL</td>
<td>Stephen F. Booth</td>
</tr>
<tr>
<td>libAVL</td>
<td>BST for wisdom manager</td>
<td>GNU/GPL</td>
<td>Ben Pfaff</td>
</tr>
</tbody>
</table>

FFTW was not integrated – that would be a big step backwards, since it is still being developed, thus it would be a substantial blockage for later updates (and with dynamic linking it is enough to update just the library, without recompiling depending software).
In case of CCL library it is assumed, that no further development was planned (I tried to contact the author by e-mail; new configuration feature was added to CCL just before it was integrated with mfft).

LibAVL does not require any further modifications and can be used “as is”.

How does FFTW work

FFTW provides extreme portability, while it is still able to maintain extra ordinary speed for all types of transforms, regardless the rank (number of dimensions) and size (can work even with prime types). It is very important to get acquainted with FFTW internals and way of working, because it will simplify a lot understanding mfft's documentation.

Portability and adaptivity – let's plan!

Calculation of FFTs in FFTW library is divided into two parts: planning and executing. Execution part only uses reference returned by the planner and in fact “calculates” the FFT, but as it will be revealed in a moment, it is not the hardest task.

As it was mentioned in previous parts, one of good attempts to solve the general DFT problem would be to run a tournament and benchmark available algorithms (and their flavors). For this process to be efficient (the more algorithms, the bigger the chance for faster solution), the space of available algorithms has to be big enough (this requires quite large set of algorithms). For solving this problem FFTW developers used dynamic programming approach, which strongly relies on expressing the problem.

In the FFTW-language available algorithms are called plans. To be able to use dynamic programming approach, each problem has to be broken down into subproblems which are further reused several times – these are the so called “overlapping subproblems”. Thus, expression of the problem is very important – also because it puts the upper-bound limit on the space of available plans. In dynamic programming approach, the space of available plans should be big enough to contain “good” algorithms, but of course it cannot be enormous, because then planning would take too much time.

Calculating DFT: overview

As it was mentioned before, planning is complex and the most time consuming task. What in fact is happening when a DFT is being calculated is:

- the planner is fed with information about the transform size and type
- the problem is split into several sub-problems until these become simple enough to use ready-made optimized pieces of code called codelets
- the planner measures execution time of different solutions (which are instantiated as solvers); a solver can return either a pointer to the plan or NULL pointer, when it could not create a plan for given input – in case when plans for sub-problems are needed, solvers call recursively planners
- the fastest algorithm is picked up and returned
- FFT is calculated according to the returned plan

Current release of FFTW (FFTW3) comes with about 150 ready-made codelets which should suit most needs. If one needs to have codelets that will deal with special cases – there is a special tool that comes with FFTW called **genfft**, which is a special-purpose “FFT compiler” which can generate optimized code from high-level mathematical description of the DFT algorithm.

**Implementation notes for multi-threaded environments**

FFTW provides multi-threaded API, which is the only optimal solution for SMP systems. Of course, it might happen, that one would include FFTW inside his/her own parallel program. Then it is very important to remember to properly synchronize threads, especially the planning procedure. This is due to the fact, that planners exchange and share a lot of data. In multi-threaded environment, the planning process should be treated as critical section and be performed only by one thread. Afterwards, other instances will use available planner data (or the so called accumulated wisdom).

There is one additional requirement coming from mfft: **mfft_shutdown()** functions has to be called from one thread only, too.

**Wisdom: using memoization**

FFTW heavily uses dynamic programming for finding the fastest algorithm. Recalculating the same thing every time would be inefficient and totally unacceptable, thus planners during runtime make use of memoization. It is a technique used in dynamic programming applications for speeding up the computations. It simply means, that results of functions are stored for further retrieval. In case of FFTW, planner does not hold the full result, as it might be big and thus memory consuming. Instead it stores only hash (strong MD5) of the problem and pointer to the solver that created the plan.

At some point developers of FFTW introduced the so called “wisdom”, which is nothing else than precomputed set of plans, which now can be exported and imported (FFTW provides set of universal calls, which look and act similarly to system calls). There is also additional utility called **fftw-wisdom**, which is nothing else than a wisdom generator which can be used to pre-compute plans. These can be stored either as system wisdom (available under /etc for all system users) or single files.

Wisdom data is cumulative, so that it is possible to export/import wisdom for several different transforms. It should be stressed out, that wisdom files should be always generated on the machine that will be used for calculations!

**Conclusions**

As it is visible on graphs coming from benchFFT, FFTW library offers fast execution and extreme portability. The interface with integrated multi-threaded API (configurable at compilation time) offers several different options that can affect the overall result, plus some additional features, as it was mentioned in the documentation, “for academic fun”. The **mfft** library is
nothing else, than an interface to the FFTW (for 2D transforms only), which gives possibility of configuring the “FFT-subsystem” without recompiling neither the library, nor software.

**MFFT**

MFFT is written in plain C and except some parts of the code (assembly snippets), it should be possible to use it on different architectures (as for now only x86 and x86_64 were tested). The library has built-in wisdom manager, which helps organizing accumulated wisdom, automatically imports it and stores human-readable wisdom information in a separate file. It also enables to configure threads, measure time with two different methods and many more.

**Development method**

To provide maximum portability and simplify building process of the library, for development and debugging I used Anjuta\(^39\) DevStudio, which is a GPL C/C++ IDE which was natively designed to run on Gnome Window Manager. Apart from nice interface, Anjuta offers also set of project maintenance functions, that can substantially simplify project management tasks. In this matter Anjuta is fully compatible with GNU standard tools like automake, autoconf and libtool.

Autoconf is a set of m4 macros that generate configure shell script which is further supposed to configure the software on a given system/platform. It works on most UNIX-like systems and can be also used for cross compilations (although, usually with some problems\(^40\)).

Automake generates Makefile.in files and requires autoconf as prerequisite. The last one, libtool is a set of commands that simplify building libraries, and can be used in all kinds of makefiles: Makefile, Makefile.in and Makefile.am.

**Configure.in**

This file is used to produce the configure script. It contains several macro definitions, that run several checks and try to adapt the source to the system. For mfft the following commands were added to the generic configure.in created by Anjuta:

```
AM_PROG_AS
Enables support for assembly sources. Needed for the rdtsc.S file, which contains routine mfft_tim which accesses TSC cycle counter of IA-32 processors (could be also solved as in-line assembly).

AC_CHECK_LIB(fftw3, fftw_execute,[LIBS="$LIBS -lfftw3"], AC_MSG_ERROR([Please install
```

39 For more information please visit http://www.anjuta.org/
40 This depends on how exotic the target platform is.

88 of 116
fftw3 (http://www.fftw.org/))

Checks if fftw3 library is available by compiling a simple program which calls fftw_execute function.

AC_CHECK_LIB(m, sin,[LIBS="$LIBS -lm"])

Comments are unnecessary.

AC_ARG_ENABLE(silent, [ --enable-silent supress all debugging messages (advanced)],
[    
    AC_DEFINE([HAVE_MFFT_SILENT], 1, [Get rid of debugging messages])
])

Compiles mfft without debugging messages. Should be used only by advanced users that already have wanted wisdom files. This option works on the preprocessor level and thus will decrease number of branch instructions, which can slightly improve overall performance.

AC_ARG_WITH(fftw3_threads, [ --with-fftw3-threads enable multi-threaded API for fftw3 (pthread only)],
[    
    AC_CHECK_LIB(fftw3_threads,fftw_init_threads,
    [        AC_DEFINE([HAVE_FFTW3_THREADS],1,[This means that
        pthread is present])
        
        LIBS="$LIBS -lfftw3_threads -lpthread"
    ],
    [        AC_MSG_ERROR([fftw3 was not compiled with --enable-threads. You can't use multi-threaded API.])
        ]
    ]
])

If fftw3 was compiled with thread support, then mfft should be compiled with it, too. Otherwise, the library will have hard coded number of threads and will not react to changes in configuration file.

Entries presented above were created by hand.

Summary of custom compilation options for mfft:

--with-fftw3-threads

Enables multi-threaded API in mfft. To use it, fftw3 has to be compiled with –enable-threads option.
--enable-silent
Suppresses all debugging messages. The library will not print anything to stdout.

Makefile.am
First version of this file was created by Anjuta. It is used by automake to generate Makefile.in files. Contains information about subdirectories in the project, documentation files and similar.

Each subdirectory in the project (in this case src/ and include/) contains its own Makefile.am which is further processed by automake to generate Makefile.in files. Below are contents of the Makefile.am from the src/ directory (comments in bold; -- example):

```plaintext
-- defines include directory
INCLUDES = -I../include
-- CFLAGS - C Compiler FLAGS
AM_CFLAGS =
    -Wall
    -g
-- name of the target object
lib_LTLIBRARIES = libmfft.la

-- list of source files
libmfft_la_SOURCES =
    mfft.c
    rdtsc.S
    ccl_get.c
    ccl_iterate.c
    ccl_parse.c
    ccl_release.c
    ccl_reset.c
    bst.c
-- linker flags
libmfft_la_LDFLAGS =
-- external libraries that should be linked with target object - this entry is empty
-- as $LIBS variable is setup during ./configure process (when particular library is detected, then proper information for the linker is appended to the $LIBS variable)
libmfft_la_LIBADD =
```

Iterations and milestones
Before mfft was released as it is, there were several other intermediate patches and small applications:
- first demo implementation calculating 1D FFT and measuring time of
execution (FFTW)
- second demo calculating 2D FFT; input was read from a file – this demo further used as a foundation of mfftest utility (reads data from a file, calculates forward and inverse transform – uses of course mfft) (FFTW)
- third demo implementation for MoInS; available in two versions (FFTW)
- mfft
There is also one more piece of software that uses mfft: FFTtoolbox. It was created just for academic purposes, to explore details of Fourier Transform and filtering in frequency domain. The program can:
- write image containing magnitude spectrum of the input image
- perform simple filtering on the input image; in this mode filter image has to be also specified (and is assumed to be in frequency domain; unfortunately, this puts certain limitation on the filter, which has only real values)
- adjust brightness of output images in several different ways

Source code documentation
Include file contains all prototypes of functions and is well documented. Code is documented only in places where it was absolutely necessary. Each mfft release contains also other files, and these are:
- AUTHORS (information about authors)
- COPYING (GPL2 License text)
- ChangeLog (changes among versions)
- INSTALL (general install instructions)
- NEWS (unmaintained)
- TODO (list of things to be done)
- README (detailed instructions about installation procedure)

AUTHORS, README, TODO and ChangeLog – these files are maintained by the author or mfft. Below only ChangeLog will be presented (other files are too big, please refer to the source code available on the CD):

```plaintext
0.2.4
* Fixed transform plans for non-symmetric images.
* x86_64 version available (excludes tsc ASM call)

0.2.3
* Fixed trailing slash in wisdom_directory (you can have a trailing slash now).
* Initialization debug information depends on MFFT_DEBUG environmental variable.
  If set, then init debug is active. After the library is configured configuration directives take over.
```

91 of 116
Program was developed to be as much as possible compliant with GNU Coding Standards.

Each distribution contains also well documented sample configuration file which will be covered in details in next chapters.

**Idea of mfft and key features**

The idea of mfft came after implementing several demo applications. It was a mixture of my own observations and feedback received from demo testers. The so called “wish list” is presented below, and should be considered as a starting point of mfft development:

- easy wisdom generation and maintenance
- configurable multi-threaded interface
- built-in precise time measurement
- configuration file

All these features are currently present in mfft. Generally speaking, most of them were merged together and now form monolithic configuration interface.

**Wisdom manager**

Wisdom manager is responsible for exporting and importing wisdom files. The key feature of it, is a possibility of maintaining a list of available wisdoms within the wisdom file. These files have .info extension and are in human readable format. Mfft parses .info file during configuration phase and if wisdom generation is enabled, can inform the user for example, that wisdom for specified transform is not present, and thus it might take some time before plan will be generated (when running in exhaustive planning modes). This might be useful piece of information, as during planning the application appears to be frozen.

Wisdom manager saves also other useful pieces of information about wisdoms like: number of threads, type (forward or inverse) and level (method of planning). Wisdom manager configures itself only once in the beginning when mfft_config function is called. During program run it maintains a set of available wisdoms, which is the Berkeley BST implementation (for fast, O(n logn) search).

```
-- Contents of a sample wisdom.info file
1024 1024 32 2 -1
1024 1024 32 2 1
1024 1024 64 2 -1
1024 1024 64 2 1
```

Behavior of wisdom manager can be controlled from the configuration file.
**Time engines**

32-bit version of mfft comes with two time engines – one uses available system call (and thus is fully portable among other platforms), whereas the second one offers higher precision but is limited only to x86 architecture.

**Gtod** or in other words gettimeofday is time engine that uses gettimeofday system call to measure time of execution of the fftw3_exec() function. This is the default time engine in mfft, because on the contrary to the **tsc** time engine, it is SMP-safe, which means, that it can be freely used on SMP systems, and measured time will always be correct – details will be discussed in tsc section. Of course, because of that gettimeofday is said to be of small accuracy, also because it is a system call, which puts additional overhead and therefore can result in inaccurate measurements.

Gettimeofday() is portable.

**Tsc** or in other words rdtsc is a mnemonic from x86 assembly language which means ReaD Time Stamp Counter. It returns registers EDX:EAX which contain number of ticks from processor reset. It is a very precise method of time measurement, but can be used only on x86 platforms. Other disadvantage of tsc engine is that it is not SMP-safe. To measure the time, function has to be called twice. There is no guarantee that values of tsc will come from exactly the same processor, because one never knows, if the task was switched to another processor.

It is very important to know how the timer functions are implemented, because this knowledge will usually give us more information about the accuracy of our measurements. Generally speaking, in case of gettimeofday(), the overhead is rather huge, also because it further uses a system call to get access to the timer. But this overhead can be compensated either by measuring large parts of code or by repeating the experiment several times (measured time should be smaller than in normal conditions, as running the same part of code repeatedly will result in less cache misses). As it was mentioned before, gettimeofday() is SMP-safe, and thus at this point, we might be sure that obtained result is correct. This might not be the case for tsc – but here there are lots of other tricks that can be used. This method is suitable for measuring short sequences of code (executing faster than 30ms) – but this case, we are on our own, because we never know if the task was switched to another CPU on SMP system (thus, it is very important to have proper scheduling policy!).

Default gtod time engine for mfft is a result of compromise between portability and accuracy.

**Configuration file and runtime settings**

The library checks environmental variable **MFFT_CONFIG** which should contain path to configuration file. No command line arguments, no hard coded values – everything is dynamic. In one of the versions, mfft was expanded and
checks also other environmental variable: **MFFT_DEBUG**. If this variable is set, then mfft will print on the screen initialization information (before configuration file is opened and parsed).

The library configures itself only once in the beginning (function `mfft_config`). Then it parses configuration file. All configuration values are kept in a structure which is used by almost every function within the library. If the configuration file is not set, there are hard coded default values to prevent the library from crashing. Configuration management functions were designed to be as easy as possible and thus can be easily expanded.

Default configuration values are set as macro definitions in the file `mfft.h` (prefix MFFT_CFG_).

Structure that keeps runtime values:

```c
struct mfft_conf_struct {
    char platform[MFFT_MAX];
    char wisdom_directory[MFFT_MAX];
    char hostname[MFFT_MAX];
    unsigned int generate_wisdoms;
    unsigned int wisdom_level;
    unsigned int fft_sign;
    unsigned int enable_debug;
    unsigned int enable_time;
    unsigned int time_engine;
    unsigned int enable_threads;
    unsigned int threads_number;
    unsigned int cpu_freq;
    struct bst_table *mfft_w;
    unsigned int bst_created;
    unsigned int enable_wisdom_manager;
    unsigned int enable_plan_info;
};
```

To add new entry it is enough to:

- create macro definition of the default value
- add entry to the `mfft_conf` structure
- add default value assignment to the function `mfft_set_defaults()`
- add parsing entry to the function `mfft_parse_entry()`

Configuration files has 'key = value' structure and supports comments after '#' signs. After that, new entry is ready to use and can be referenced from mfft configuration structure (that of course should be initialized).

Below are contents of sample configuration file, which is distributed with mfft:

```ini
# Configuration file for mfft
```
# '#' starts a comment
#
# Syntax of the file is as follows:
# key = value
# Please, read it carefully. If you have questions - contact me (see AUTHORS in mfft).
#
# MFFT CONFIG STARTS HERE
#
# Platform you are running now. Please, fill it in.
# Hints:
# - i686 - generic IA32
# - i586 - _why_ do you run it on such slow machine? :)
# - pentium3
# - pentium4
# - nocona - for x86_64 (Intel EM64T in 64bit mode)
# - k8 - for AMD64 in 64bit mode
# You can put whatever you want (even 'crap' if you work on Florian's computer :) - but it should
# correspond to the CPU/ARCH you have (please, no spaces in it). I suggest using values that can be used
# as arguments for '-march=' option in gcc/g++.
platform = pentium4

# Where do you keep your wisdom files?
# Remember, that you should be able to read and write inside this directory!
# Default: your $HOME

wisdom_directory = /nfs/staff/mic/Projects/wisdom

# Should I generate necessary wisdom files?
# This will take some time, but only once. Afterwards the engine
# will use built-in wisdom manager to load them.
# Default: no
# If you say no, then FFTs will be calculated with level = 1.
# You can always generate wisdoms - they are internally cumulated, so that once you generate
# a wisdom it will not be forgotten (if you delete both files, then it will :).
# Remember - generation involves I/O operations and additional checks, system calls and so on -
# it might have impact on the overall performance of mfft.

generate_wisdoms = no # yes / no
Wisdom manager is a tool that is responsible for managing information about wisdoms you have in your wisdom file. It keeps the data in human readable form in the file with .info extension. You can have dozens of wisdoms if you want - wisdom manager uses O(log n) search algorithm (BST).

If you don’t need it - disable it.

If you enabled 'generate_wisdoms', then wisdom manager will be enabled, too.

```bash
# enable_wisdom_manager = yes

# If you want to run immediately - no problem - but FFT calculation will take more time.
# 64 - heuristic (fast plan, slow execution)
# 32 - patient (~2-5 mins. for plan, fast execution)
# 8 - exhaustive (I got 30 mins for plan, very fast execution)

wisdom_level = 8
```

# Exponent sign when calculating the transform (for inverse (-1*fft_sign) is assumed)
# Default: 1

```bash
fft_sign = 1
```

# Show debug information (number of mul/div/add).
# Notice: this will slow down execution, it's just for fun.
# Default: no
# Does not depend on 'enable_debug'.

```bash
# enable_plan_info = yes
```

# Enable debugging. When it's set to 'yes' then mfft prints a lot to stdout.
# It might have huge impact if your graphics card/drivers are not friends with 2D acceleration. Generally speaking - it's useful to see it, because you get a general idea about internal processes, but then it's completely useless.

```bash
# enable_debug = no
```

# Enable time measurements.
# This enables time measurements for mfft. It will give you exactly the time spent on calculating FFT. Does not depend on 'enable_debug'.

```bash
# enable_time = yes
```

# Time engine: you can select one of available engines, they are listed below:
# - tsc
# - gtod
# Default: gtod (compatibility and SMP safe)
# If you use tsc on SMP system:
# it might happen that you will receive bogus results. The idea of tsc engine is that
# it uses rdtsc command to access TSC on CPU. If you have more CPUs... and if your task
# is rescheduled to another CPU, then you might read TSC from a different CPU which of
course
# has no sense.
# 'tsc' can be considered as high precision time engine.
# 'gtod' is safe, but involves some overhead (it's a system call)

time_engine = gtod

# Enable threads (library has to be linked against fftw3_threads and pthread)
# Default: no
# Remember: plans depend on this feature (please read description below).
enable_threads = yes

# Number of threads (wisdom files depend on it)
# If you enabled threads, then it should be equal to number of CPUs you have.
# Remember, that plans (wisdoms) are generated for a given number of threads.
# For example, you have a wisdom file generated for 2 threads, then you change
# number of threads to 4 - and of course - you need a plan for 4 threads!
threads_number = 2

Interface is clean and flexible. Some of the directives depend on compilation options, like `enable_threads` and `threads_number` – these options will have effect if and only of the library was compiled with `--with-fftw3-threads` option.

Configuration file is well documented and presents all features available in mfft, thus other comments are needless.

**In action: how to use the library**

Implementation is similar to the steps described in fftw3 manual. The standard procedure is expanded with two additional actions that are mfft specific: library's setup and shutdown. Below are details of implementation:

```
Used types:
fftw_complex i, o;
fftw_plan plan;
struct mfft_conf_struct cnf;
```
Structure of the body:

```c
i = (fftw_complex *) fftw_malloc(N*sizeof(fftw_complex));
o = (fftw_complex *) fftw_malloc(N*sizeof(fftw_complex));
mfft_config(&cnf); // configure the library
plan = mfft_fft_prepare(&cnf, i,o,1024,1024,1); // prepare plan

// initialize i with some data
mfft_fft_exec(&cnf, plan); // execute plan (calculate FFT)

[ repeat as many times as needed; put new data to i ]

mfft_shutdown(&cnf); // free memory taken by mfft
fftw_free(i); fftw_free(o);
```

It is very important to remember to use `fftw_malloc` and `fftw_free` instead of standard `malloc` and `free`. First set of functions aligns the data properly, so that SIMD extensions can be used.

For more details about implementation, please read sources of mfftest program and MoInS patches (available on the CD).

**Running mfftest: mfft output**

Mfftest program is a simple test program for mfft which can be used for measuring time of execution for both inverse and forward transforms. It takes two arguments:

- path to file containing data (each point equals one line); if we denote value as A, then mfftest will create complex input as follows: \( \Re(A) - \Im(A/2) \)

- optional argument with number of repetitions

fftw3 returns unscaled array of values – to avoid possible slow downs coming from floating point exceptions after forward transform the result is scaled (by the size of array). Mfftest is not suitable for making measurements for real-time systems, as it operates on the same set of data, thus exploits cache properties. Below is sample output from mfftest:

```
mfftest: scaling between fwd and inverse transform (by 1048576)
--------[1 of 2]--------
MFFT_TIME(gtod): execution time = 115 ms
MFFT_TIME(gtod): execution time = 99 ms
--------[2 of 2]--------
MFFT_TIME(gtod): execution time = 106 ms
MFFT_TIME(gtod): execution time = 100 ms
```

Output above does not contain any debugging messages. Below is output with...
all debugging messages present:

```c
mfft.c:630] mfft_wisdom_holder: bst_create
mfft.c:636] mfft_wisdom_holder: bst_create success
mfft.c:639] Looking for wisdom info file...

mfft.c:329] mfft_wisdom_name: Wisdom name
mfft.c:336] /nfs/staff/mic/Projects/wisdom/mic-pentium4.info
mfft.c:656] mfft_wisdom_holder: wisdom info finished.
    - size: 1024x1024, level: 32, threads: 2, type: -1
    - size: 1024x1024, level: 32, threads: 2, type: 1
    - size: 1024x1024, level: 64, threads: 2, type: -1
    - size: 1024x1024, level: 64, threads: 2, type: 1
mfft.c:403] mfft_fft_prepare: input size
mfft.c:404] - cols = 1024
mfft.c:405] - rows = 1024
mfft.c:408] mfft_fft_prepare: threaded, running plan
mfft.c:410] mfft_fft_prepare: initializing threads...
fft_init_threads() = 1
mfft.c:414] mfft_fft_prepare: number of threads = 2
mfft.c:288] mfft_wisdom_loaded = 0
mfft.c:312] mfft_wisdom_name: Wisdom name
mfft.c:319] /nfs/staff/mic/Projects/wisdom/mic-pentium4.fftw
mfft.c:296] mfft_load_wisdom: file found, importing
mfft.c:302] mfft_load_wisdom: finished
mfft.c:429] mfft_fft_prepare: INFO BODY!
fft_init_threads() = 1
mfft.c:431] mfft_fft_prepare: WISDOM MANAGER INFO ->
```

99 of 116
At highest verbosity level (output above) the library looks as very talkative – this was done on purpose. Good debugging can simplify both development and
Disabling debugging might be beneficial in terms of speed because:
- nothing is sent to stdout/stderr
- the following macro simply disappears from the resulting binary code

```c
#if HAVE_MFFT_SILENT

#define MFFT(msg);
#define MFFT_INT(msg,val);
#else
#define MFFT(msg) if(mfft->enable_debug) { printf("MFFT[%s:%.3d] %s\n", __FILE__, __LINE__, msg); fflush(stdout); }
#define MFFT_INT(msg,val) if(mfft->enable_debug) { printf("MFFT[%s:%.3d] %s = %d\n", __FILE__, __LINE__, msg,val); }
#endif
```

## Integration into MoInS

Use of mfft is optional and has to be explicitly enabled during compilation time. The following macro was added to the MoInS' configure.in file:

```
AC_ARG_ENABLE(mfft,[ --enable-mfft enable MFFT library (FFT)],
  [  
    AC_CHECK_LIB(mfft,mfft_config,[
      AC_DEFINE([HAVE_MFFT],1,[MFFT and FFTW are present])
    ])
  ])
```

When mfft is present and enabled, macro sets value of MFFTLIB variable which is further referenced in appropriate Makefile.am file:

```
libImageFkt_la_LIBADD = $(MFFTLIB)
```

It also defines preprocessor variable HAVE_MFFT, so that patch code is included.

Below is the patch (only for forward transform – they differ inly in the sign) that fills the body of the fft() routine:

```c
#if HAVE_MFFT
  fftw_complex *fout,*fin;
  double * inputArr,*outputArr;
  int _cols, _rows, _loop;
  fftw_plan p;

  inputArr = (double *) inp.pData();
  _cols = inp.cols();
```
Since for large transforms (like 1024x1024) there is no difference between in-place and out-of-place calculations, why not trying in-place algorithm? With in-place algorithm we might conserve about 16MB of memory space (for 1024x1024 image) and some overhead, because memory allocations means system call. Another thing is, that such allocated memory space has to be properly aligned).

One could say “why bother, when it changes so little“. That little can be everything on real-time systems, where everything counts.

**Measurements**

All source files coming from different benchmarks, including cache benchmarks can be found on attached CD.

The most important are of course measurements of FFT routines – before and after the project began.
**MoInS**

All measurements were performed on:

<table>
<thead>
<tr>
<th><strong>CPU</strong></th>
<th>Intel(R) Pentium(R) 4 CPU 3.00GHz HT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RAM</strong></td>
<td>512MB</td>
</tr>
<tr>
<td><strong>CPU FLAGS</strong></td>
<td>fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clfush dts acpi mmx fxsr sse sse2 ss ht tm pbe pni monitor ds_cpl cid xtr</td>
</tr>
<tr>
<td><strong>Kernel</strong></td>
<td>2.6.14-gentoo-r5 #1 SMP</td>
</tr>
<tr>
<td><strong>Linux distribution</strong></td>
<td>Gentoo; optimized for current hardware</td>
</tr>
</tbody>
</table>

In both cases MoInS was compiled with the following CFLAGS:
-Wno-deprecated -march=pentium4 -O2 -funroll-loops -mfpmath=sse -msse2 -msse -mmmx -ffast-math

Scheduling policy: No Forced Preemption
Source computations available on the CD (odt spreadsheet).

**If not stated then Out-of-place algorithm**

<table>
<thead>
<tr>
<th><strong>Impl.</strong></th>
<th><strong>Notes</strong></th>
<th><strong>Forward [avg]</strong></th>
<th><strong>Inverse [avg]</strong></th>
<th><strong>Forward [stdev]</strong></th>
<th><strong>Inverse [stdev]</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>In-place algorithm</td>
<td>1229ms</td>
<td>1167ms</td>
<td>76.87</td>
<td>14.1</td>
</tr>
<tr>
<td>MFFT</td>
<td>Exhaustive + 2 threads</td>
<td>105ms</td>
<td>104ms</td>
<td>0.46</td>
<td>0.82</td>
</tr>
<tr>
<td>MFFT</td>
<td>Estimate + 2 threads</td>
<td>360ms</td>
<td>359ms</td>
<td>1.6</td>
<td>0.82</td>
</tr>
<tr>
<td>MFFT</td>
<td>Estimate, no threading</td>
<td>367ms</td>
<td>365ms</td>
<td>2.23</td>
<td>3.51</td>
</tr>
</tbody>
</table>

**Best configuration – original and MFFT, ratio: 11.70 (fw), 11.22(inv)**

**Worst configuration – original and MFFT, ratio: 3.34 (fw), 3.19 (inv)**

**In-place algorithm**

<table>
<thead>
<tr>
<th><strong>Impl.</strong></th>
<th><strong>Notes</strong></th>
<th><strong>Forward [avg]</strong></th>
<th><strong>Inverse [avg]</strong></th>
<th><strong>Forward [stdev]</strong></th>
<th><strong>Inverse [stdev]</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MFFT</td>
<td>Exhaustive + 2 threads</td>
<td>93ms</td>
<td>92ms</td>
<td>1.28</td>
<td>0.5</td>
</tr>
</tbody>
</table>
### Implantation and Inverse

<table>
<thead>
<tr>
<th>Implantation</th>
<th>Notes</th>
<th>Forward [avg]</th>
<th>Inverse [avg]</th>
<th>Forward [stdev]</th>
<th>Inverse [stdev]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFFT</td>
<td>Estimate + 2 threads</td>
<td>366ms</td>
<td>367ms</td>
<td>4.5</td>
<td>5.44</td>
</tr>
<tr>
<td>MFFT</td>
<td>Estimate, no threading</td>
<td>382ms</td>
<td>377ms</td>
<td>7.18</td>
<td>8.5</td>
</tr>
</tbody>
</table>

Best configuration – original and MFFT, ratio: 13.21 (fw), 12.68 (inv)
Worst configuration – original and MFFT, ratio: 3.21 (fw), 3.09 (inv)

Best conf. - In-place vs. out-of-place, MFFT, ratio: 0.88 (fw), 0.88 (inv)
Worst conf. - In-place vs. out-of-place, MFFT, ratio: 1.04 (fw), 1.03 (inv)

Differences between out-of-place and in-place calculations in terms of “SpatialMonogenicTrafo” measurement point.
Tests performed on CGPTest demo program (Author: Morten Krogh Skov).

<table>
<thead>
<tr>
<th>Implantation</th>
<th>In-place</th>
<th>Out-of-place</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Left [s]</td>
<td>Right [s]</td>
</tr>
<tr>
<td>Original</td>
<td>15.1742</td>
<td>15.2217</td>
</tr>
<tr>
<td>MFFT (estimate, 2 threads)</td>
<td>8.24145</td>
<td>8.1763</td>
</tr>
<tr>
<td>MFFT (exhaustive, 2 threads)</td>
<td><strong>5.56497</strong></td>
<td><strong>5.54013</strong></td>
</tr>
<tr>
<td>MFFT (estimate, no threading)</td>
<td>8.17425</td>
<td>7.98708</td>
</tr>
<tr>
<td>Original (*)</td>
<td>17.7245</td>
<td>17.681</td>
</tr>
<tr>
<td>MFFT (exhaustive, 2 threads) (*)</td>
<td>5.67941</td>
<td>5.63868</td>
</tr>
</tbody>
</table>

(*) - fully preemptible kernel

It is clearly visible, that in-place algorithm performs much better and offers higher “calculation stability” (calculation time of successive transforms does not differ much).

In-place patch is currently present in MoInS.

Best configurations for in-place algorithms: fully preemptible kernel
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>In-place algorithm</td>
<td>1409ms</td>
<td>1483ms</td>
<td>94.11</td>
<td>148.42</td>
</tr>
<tr>
<td>MFFT</td>
<td>Exhaustive + 2 threads</td>
<td>89ms</td>
<td>89ms</td>
<td>1.6421</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Time for “SpatialMonogenicTrafo” points was measured with tictoc utility written by Morten Krogh Skov (uses TSC).

**MFFTEST**

Mfftest program for testing mfft library. 50 measurements per each test. No distinction between forward and inverse transforms. High standard deviation means, that there was a substantial difference in time of execution between forward and inverse transforms. Unless stated tests were performed for out-of-place algorithms.

**Florian**

| CPU       | Intel(R) Xeon(TM) CPU 3.40GHz HT EM64T, 1024KB cache |
| RAM       | 4GB                                                 |
| CPU Flags | fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe lm pni monitor ds_cpl est cid cx16 xtrp |
| Kernel    | 2.6.11.4-21.11-bigmp #1 SMP                        |
| Linux dist | SuSE Linux 9.3 (i586) (32-bit)                   |

<table>
<thead>
<tr>
<th>Model</th>
<th>Time engine</th>
<th>Average</th>
<th>StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive + 2 threads</td>
<td>gtod</td>
<td>107ms</td>
<td>13.01</td>
</tr>
<tr>
<td></td>
<td>tsc</td>
<td>108.5ms</td>
<td>12.9</td>
</tr>
<tr>
<td>Estimate + 2 threads</td>
<td>gtod</td>
<td>420ms</td>
<td>11.81</td>
</tr>
<tr>
<td></td>
<td>tsc</td>
<td>421ms</td>
<td>12</td>
</tr>
<tr>
<td>Estimate, no threads</td>
<td>gtod</td>
<td>420ms</td>
<td>11.78</td>
</tr>
<tr>
<td></td>
<td>tsc</td>
<td>421ms</td>
<td>13.9</td>
</tr>
</tbody>
</table>

**Mic**

| CPU       | Intel(R) Pentium(R) 4 CPU 3.00GHz HT, 1024KB cache |
| RAM       | 512MB                                               |
### CPU
Intel(R) Pentium(R) 4 CPU 3.00GHz HT, 1024KB cache

### CPU FLAGS
fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe pni monitor ds_cpl cid xtpr

### Kernel
2.6.14-gentoo-r5 #1 SMP

### Linux distribution
Gentoo; optimized for current hardware

<table>
<thead>
<tr>
<th>Model</th>
<th>Time engine</th>
<th>Average</th>
<th>StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive + 2 threads</td>
<td>gtod</td>
<td>103ms</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>tsc</td>
<td>103ms</td>
<td>9.3</td>
</tr>
<tr>
<td>Estimate + 2 threads</td>
<td>gtod</td>
<td>378ms</td>
<td>9.37</td>
</tr>
<tr>
<td></td>
<td>tsc</td>
<td>377ms</td>
<td>2</td>
</tr>
<tr>
<td>Estimate, no threads</td>
<td>gtod</td>
<td>381ms</td>
<td>2.98</td>
</tr>
<tr>
<td></td>
<td>tsc</td>
<td>381ms</td>
<td>3.3</td>
</tr>
</tbody>
</table>

### Qtux
Dual Core AMD Opteron(tm) Processor 270 2.0GHz (2x), 1024KB cache

### RAM
4GB

### CPU FLAGS
fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx mmxext fxsr_opt lm 3dnowext 3dnow pni lahf_lm cmp_legacy

### Kernel
2.6.16-gentoo-r7 #2 SMP

### Linux distribution
Gentoo; optimized for current hardware

<table>
<thead>
<tr>
<th>Model</th>
<th>Time engine</th>
<th>Average</th>
<th>StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive + 4 threads</td>
<td>gtod</td>
<td>48ms</td>
<td>2.92</td>
</tr>
<tr>
<td>Estimate + 4 threads</td>
<td>gtod</td>
<td>114ms</td>
<td>38.39</td>
</tr>
<tr>
<td>Estimate, no threads</td>
<td>gtod</td>
<td>239ms</td>
<td>73.5</td>
</tr>
</tbody>
</table>
Volks

<table>
<thead>
<tr>
<th>CPU</th>
<th>Dual Core AMD Opteron(tm) Processor 285 2.6GHz (x2), 1024KB cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>4GB</td>
</tr>
<tr>
<td>CPU FLAGS</td>
<td>fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt lm 3dnowext 3dnow pni lahf_lm cmp_legacy</td>
</tr>
<tr>
<td>Kernel</td>
<td>2.6.16-gentoo-r7 #3 SMP PREEMPT</td>
</tr>
<tr>
<td>Linux distribution</td>
<td>Gentoo; optimized for current hardware</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Time engine</th>
<th>Average</th>
<th>StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaustive + 4 threads</td>
<td>gtod</td>
<td>54ms</td>
<td>4.2</td>
</tr>
<tr>
<td>Estimate + 4 threads</td>
<td>gtod</td>
<td>110ms</td>
<td>34.62</td>
</tr>
<tr>
<td>Estimate + 2 threads</td>
<td>gtod</td>
<td>133ms</td>
<td>32.2</td>
</tr>
<tr>
<td>Estimate, no threads</td>
<td>gtod</td>
<td>220ms</td>
<td>62.92</td>
</tr>
</tbody>
</table>

In-place algorithms

Scheduling policy: no forced preemption.

<table>
<thead>
<tr>
<th>Host</th>
<th>Model</th>
<th>Average</th>
<th>StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>volks</td>
<td>MFFT (exhaustive, 4 threads)</td>
<td>49ms</td>
<td>2.69</td>
</tr>
<tr>
<td>volks</td>
<td>MFFT (estimate, 4 threads)</td>
<td>111ms</td>
<td>30.31</td>
</tr>
<tr>
<td>volks</td>
<td>MFFT (estimate, no threads)</td>
<td>221ms</td>
<td>62.31</td>
</tr>
<tr>
<td>mic</td>
<td>MFFT (exhaustive, 2 threads)</td>
<td>92ms</td>
<td>2.10</td>
</tr>
<tr>
<td>mic</td>
<td>MFFT (estimate, 2 threads)</td>
<td>372ms</td>
<td>0.85</td>
</tr>
<tr>
<td>mic</td>
<td>MFFT (estimate, no threads)</td>
<td>380ms</td>
<td>0.90</td>
</tr>
</tbody>
</table>

In-place, out-of-place with fully preemptible kernel

Tests performed only on machine “mic”.
<table>
<thead>
<tr>
<th>Model</th>
<th>Algorithm</th>
<th>Average</th>
<th>StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFFT (exhaustive, 2 threads)</td>
<td>out-of-place</td>
<td>99ms</td>
<td>3.32</td>
</tr>
<tr>
<td></td>
<td>in-place</td>
<td>89ms</td>
<td>0.62</td>
</tr>
<tr>
<td>MFFT (estimate, 2 threads)</td>
<td>out-of-place</td>
<td>392ms</td>
<td>18.21</td>
</tr>
<tr>
<td></td>
<td>in-place</td>
<td>380ms</td>
<td>3.88</td>
</tr>
<tr>
<td>MFFT (estimate, no threads)</td>
<td>out-of-place</td>
<td>401ms</td>
<td>19.15</td>
</tr>
<tr>
<td></td>
<td>in-place</td>
<td>380ms</td>
<td>0.69</td>
</tr>
</tbody>
</table>

2D FFT

Implementation that was introduced in the first part. Please have in mind the fact, that transposition is done out-of-place (thus, is inefficient)!
50 measurements per test.

<table>
<thead>
<tr>
<th>Hostname</th>
<th>Average</th>
<th>StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Florian</td>
<td>371ms</td>
<td>1.07</td>
</tr>
<tr>
<td>mic</td>
<td>427ms</td>
<td>12.48</td>
</tr>
<tr>
<td>qtux</td>
<td>320ms</td>
<td>1.03</td>
</tr>
<tr>
<td>volks</td>
<td>280ms</td>
<td>2.03</td>
</tr>
</tbody>
</table>

On each machine file 2d.c was compiled with the following CFLAGS:
-O2 -mfpmath=sse -msse -msse2 -funroll-loops -march={ARCH}

SciMark

Please compare results obtained on host natalie (only 128KB of cache). On each machine SciMark was compiled with the same CFLAGS as 2d.c.
Natalie's configuration:

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel(R) Celeron(R) CPU 2.80GHz, 128KB cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>512MB</td>
</tr>
<tr>
<td>CPU FLAGS</td>
<td>fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe cid xtr</td>
</tr>
<tr>
<td>Kernel</td>
<td>2.6.11.10 #7 SMP</td>
</tr>
<tr>
<td>Linux distribution</td>
<td>Debian testing/unstable</td>
</tr>
<tr>
<td>System</td>
<td>Using</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Florian</td>
<td>2.00 seconds min time per kernel.</td>
</tr>
<tr>
<td>mic</td>
<td>2.00 seconds min time per kernel.</td>
</tr>
<tr>
<td>qtux</td>
<td>2.00 seconds min time per kernel.</td>
</tr>
<tr>
<td>volks</td>
<td>2.00 seconds min time per kernel.</td>
</tr>
<tr>
<td>natalie</td>
<td>2.00 seconds min time per kernel.</td>
</tr>
</tbody>
</table>
**Cache benchmark**

Benchmarking program that generates 8 memory bandwidth curves. Project name: “cachebench”.

**Florian**
Volks

FFT on GPU

Test program was run on host “volks” under Windows XP Professional x64. Graphics board: (check exactly model --)

Sources can be found on attached CD (they come from the publication “The FFT on GPU”).

The test included: 512x512 low-pass filtering, 1024x1024 low-pass filtering and just rendering without FFT filtering plus original results obtained by K. Moreland and E. Angel.

FFT is performed on all four color channels of the “Utah teapot”.

Movies from tests are present on attached CD.

<table>
<thead>
<tr>
<th>Host</th>
<th>Vector size</th>
<th>Frame time [avg]</th>
<th>FPS</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>volks</td>
<td>512x512</td>
<td>0.0334s</td>
<td>29.94</td>
<td>Filtering</td>
</tr>
<tr>
<td>volks</td>
<td>1024x1024</td>
<td>0.1166s</td>
<td>8.57</td>
<td>Filtering</td>
</tr>
<tr>
<td>volks</td>
<td>512x512</td>
<td>0.0166s</td>
<td>60.24</td>
<td>No filtering</td>
</tr>
<tr>
<td>volks</td>
<td>1024x1024</td>
<td>0.0166s</td>
<td>60.24</td>
<td>No filtering</td>
</tr>
<tr>
<td>(*)</td>
<td>1024x1024</td>
<td>2.7027s</td>
<td>0.37</td>
<td>Filtering</td>
</tr>
<tr>
<td>Host</td>
<td>Vector size</td>
<td>Frame time [avg]</td>
<td>FPS</td>
<td>Notes</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>------------------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>(*)</td>
<td>512x512</td>
<td>0.625s</td>
<td>1.6</td>
<td>Filtering</td>
</tr>
</tbody>
</table>

(*) results come from “The FFT on GPU” publication (K. Moreland, E. Angel)

Performance increase in terms of FPS for 512x512: **18.71**
Performance increase in terms of FPS for 1024x1024: **23.16**

Results in the table exactly show how fast is FFT on GPU and how fast are GPUs themselves evolving (publication was released in 2003).

**Final conclusions**

There are clean cases, where it is possible to draw conclusions that will be always true. Unfortunately, they are minority. Currently the technology is so advanced, that it is practically impossible to examine possible outcome without performing any tests.

It is definitely worth using all rules/tricks that might affect performance: cache-friendly rules, optimal algorithms and so on – they will always pay back in better or the same performance (in most cases there is no risk). But there are situations, where the programmer should go even closer to the hardware, because that is the only way of getting power out of contemporary units. That is for sure the case of the “FFT problem”, where the mixture of SIMD extensions and proper algorithms can give very good results, unachievable for standard implementations.

Nowadays we work with machines that have gigabytes of RAM memory, gigahertz processors and so on. For real-time programming which additionally involves operations on large streams of data it is very important to remember at each stage of development process, that **every single** operation will be repeated billions of times, and thus even single memory allocation/deallocation, system call and similar actions might affect the overall result (please recall situation, when new patch for MoInS was created – it got rid of one memory allocation/deallocation plus the algorithm was changed from out-of-place to in-place; the outcome was of course faster execution and higher stability of the FFT routine). Another thing is, that even 1 millisecond is worth the effort, because the same millisecond if obtained in several places can speed up the execution a lot.

The final project and Engineering practice made it possible to draw also other conclusions, that are listed below:

1. Even when using ready made libraries one should posses enough knowledge, so that others work will not be wasted. **This is especially important when compiling different parts of software. Hardware and software knowledge is very important and will for sure be beneficial when using a compiler that supports different extensions/architectures – and that is definitely the gcc. Many software packages try to be as universal**
and portable as possible and thus offer many compilation-time options, which can adapt the library to the hardware one possesses. Without knowledge about the hardware and its capabilities, even the best code will not perform good, just because it will not utilize the full power that lies in the hardware.

2. One should not design his/her own FFT routines; there are several ready libraries, which in many cases are result of years of development and will provide much better results – and what is very important – in most cases, portability will not interfere with performance (best example: FFTW).

   This is the most controversial conclusion. Of course, everything depends on the project background, but in many cases it is not the best idea, because it might end up on “reinventing the wheel” - with better or worse results (in many cases, with worse). “Reuse of code” should be considered whenever possible (if the code has satisfactory quality and license allows to do so). In my project, selection of FFTW was a strike home and in similar cases, whenever FFT will be involved I will always encourage to test it first.

3. Each project has its own hardware and software demands; there is no golden middle, but there are hints how to test which configuration is the best. Benchmarks, reviews, results, test cases – everything run on computers that are going to use the software – and of course, analysis of that what can be found on the Internet. The more information you have, the higher chances for your success.

4. Never trust your code – even if it works reliably. You never know... Segmentation faults will always appear when you do not expect them to appear. In case of C – always think twice before you put a line. You do not have to be an expert in gdb or other debuggers (like Electric Fence) – it is just enough if you know how to obtain a backtrace, which will point you to the point where something went wrong. And always remember: pointers do not lie and memory leaks/buffer overflows are very difficult to spot!

5. Do not be afraid of experimenting; you will not waste your time – failed experiments are also important, because you know they will fail (I spent more than a week on implementing and testing network client for solving FFT problem on several machines – it failed, but I learned a lot during that time and now I know, that this is not a good solution, especially not for real-time systems and not for today's networking). Statistics lie, but you cannot change the numbers. Testing and experimenting will help a lot in determining optimal solution of your problem. Besides, it very rarely happens, that you can apply some changes to existing code and that will not require any other modifications or data conversions. This of course again will affect overall performance.

I am aware of the fact, that this thesis might not provide all necessary pieces
of information to solve the “FFT problem” in the best way. Nevertheless, it can be definitely treated as a good starting point for further research and even better, faster implementations.

Appendices
A CD is enclosed with the report that contains:
- source code of all programs, routines and projects (except MoInS) mentioned in this paper
- FFT image gallery (in PGM format)
- video footages from “FFT on a GPU” experiment
- results of tests and benchmarks plus their source code

Acknowledgments
Without these people my work would be much harder.

Supervisors:
Henning Haugaard, Engineering College of Copenhagen
Norbert Krueger, Aalborg University Copenhagen

Access to x86_64 machines:
Lars Frokjær Knudsen, AAUK IT (ideas and C/C++ hints)
Morten Krogh Skov, AAUK

Access to EM64T:
Florian Pilz, AAUK (introduction to MoInS)

The last, but not the least:
Søren Filtenborg Jensen, AAUK IT (Hardware)
Mikael Krøyer, AAUK IT (Hardware, blackboard and “happy quotes”)
Daria Barnaś (more than support)
Rolf Nordahl (feedback and personal advices)
Stefania Serafin (words of wisdom about experiments)
Tomasz Dudzisz (general support)
Tomasz Sterna (CPU related consultations)
Janne Anderson (patience)
Employees at Aalborg University Copenhagen, that let me spend last semester of my Bsc studies in nice and friendly atmosphere.
References


12.Multiple authors, “GPU Gems 2”, Pharr/Addison Wesley [chapters available in “Full Course Notes” document available on http://www.gpgpu.org/]
