DIPLOMARBEIT

Short Vector FFTs

ausgeführt am Institut für
Angewandte und Numerische Mathematik
der Technischen Universität Wien

unter der Anleitung von
Prof. Dr. Christoph Überhuber

durch

Franz Franchetti
Matrikelnummer: 9525993
J. Patzeltgasse 3
2700 Wiener Neustadt.

Wiener Neustadt, 11. September 2000 ——————

Für
Traude und Franz Franchetti

Preface

The *discrete Fourier transform* (DFT) is one of the principal algorithmic tools in the field of scientific computing as DFT methods can be used to solve various scientific and engineering problems. For example, the DFT is essential to the digital processing of analogous signals. DFT methods are also used to solve partial differential equations which arise, for example, in the field of computational fluid dynamics. Other practical utilization of DFT methods occurs in geophysical research, vibration analysis, radar and sonar signal processing, speech recognition and synthesis, image processing, etc.

The order of complexity of the DFT was long thought to be $O(N^2)$ (since the DFT requires the evaluation of a special matrix-vector product). In 1965 Cooley and Tukey [6] published an algorithm, the *fast Fourier transform* (FFT), which reduced the computational complexity of the DFT to $O(N \log N)$. Since then numerous studies have been published on how the FFT can be implemented efficiently on advanced computer systems. The first step was made by Pease [7] in 1968. He introduced an algorithm well suited for implementations on parallel computers. In his pioneering paper, which was not based on the Cooley-Tukey variant, Pease used the *Kronecker product* notation to describe the FFT. This formalism is particularly useful as mathematical formulas involving Kronecker product operations are easily translated into various programming constructs.

By algebraically manipulating Kronecker product formulas, different programs that achieve the same computation but have different performance characteristics can be obtained. In this way algorithms can be made *architecture adaptive* to better match specific computer architectures. Therefore Kronecker products are used in this thesis as a fundamental tool for the development and description of FFT algorithms.

Modern RISC processors (and high-end PC processors) provide a special instruction to perform $\pm a \pm b \times c$, i. e., a combination of a multiplication and an addition operation – called *multiply-add* or *fused multiply-add* (FMA) operation – in the same amount of time needed for a single floating-point addition or multiplication.

New generations of processors have a *short vector* FPU that provides two-way or four-way SIMD (single instruction, multiple data) instructions.

In this thesis FFT algorithms that utilize SIMD instructions are presented. Moreover, a special version of FFTW utilizing SIMD instructions and SIMD-FMA instructions on multiple platforms is introduced. The modifications to FFTW are easy to adapt to upcoming SIMD-capable processors.

Contents

Chapter 1

Introduction

History and Evolution of the FFT

The first use of trigonometric series in analysis can be found in the work of L. Euler (1707 – 1783). Euler presented the formulas for the coefficients of the Fourier series representation of a function of a real variable. Euler used trigonometric series to describe sound propagation in an elastic medium.

The stature of Euler in his own time assured that his work was read by his contemporaries, particularly a number of French mathematicians. Among them was A.-C. Clairaut (1713 – 1765), who published the earliest formula for the discrete Fourier transform in 1754. His formula was restricted, however, to a cosine-only finite Fourier series. J. L. Lagrange (1736 – 1813) published a formula for sine-only series in 1762. D. Bernoulli (1700 – 1782) expressed the form of a vibrating string as a series of sine and cosine terms with arguments of both time and distance in 1753. This implied that an arbitrary function could be expressed as an infinite sum of cosines.

Clairaut and Lagrange were concerned with orbital mechanics and the problem of determining the details of an orbit from a finite set of observations. C. F. Gauss (1777 – 1855) extended the work of Euler and Lagrange dealing with trigonometric interpolation to include periodic functions which are not necessarily odd or even. This was done while considering the problem of determining the orbit of certain asteroids from sample locations. Gauss developed an algorithm similar to the Cooley-Tukey *fast Fourier transform*, which was based on the reduction of one large Fourier transform to several smaller ones for computing the coefficients of a finite Fourier series. Gauss' treatise describing the algorithm appeared in his collected works as an unpublished manuscript. The presumed year of the creation of his algorithm is 1805. Though not influencing the work of Cooley and Tukey, the roots of the FFT algorithm date back to the early *nineteenth* century !

J. B. J. Fourier's (1768 – 1830) interest in heat conduction led him to begin work in 1807 on the "analytic theory of heat". Published in 1822, it shows how a mathematical series of sine and cosine terms can be used to analyze heat conduction in solid bodies. Fourier spent the rest of his life working on it and expanding it to include the Fourier integral.

Both the Fourier series and the Fourier integral allow to transform physically realizable time-domain waveforms to the frequency domain and vice versa. Today

many areas of science benefit from Fourier analysis. However the method did not gain acceptance in the time of J. B. J. Fourier. The reason behind this was the distrust in the use of infinite series. One influential mathematician in 1828 commented: "Divergent series are an invention of the devil, and it is shameful to base on them any demonstration whatsoever". Since then work by Dirichlet, Friedrich, Riemann, and others have resolved any doubts about the validity of the Fourier series.

The major setback in using the Fourier transform was its prohibitive computational effort. The discrete Fourier transform (DFT) required extensive computational time to evaluate. Techniques to reduce the computational effort were developed, for instance, by C. Runge in 1903, who which essentially described the FFT. Danielson and Lanczos in 1942 recognized certain symmetries and periodicities which reduced the number of operations.

Even with the advent of the digital computer the techniques to reduce the computational complexity of the DFT were unknown until 1965 when J. W. Cooley and J. W. Tukey published their DFT algorithm which became known as the *fast Fourier transform* (FFT).

The success story of this discovery began at a meeting of the US president's scientific advisory committee. R. L. Garwin, who was in desperate need of a fast means to compute Fourier transforms for his research with helium, noted that Tukey was writing algorithms for Fourier transforms and asked him to outline his techniques. Garwin then went to the IBM research center to have it programmed. Cooley, a relatively new member of the staff , was given the problem because to his own admission "had nothing important to do" and quickly worked it out. Thinking he would hear no more about it, Cooley "went back to doing some real work". This was obviously not the case since Garwin foresaw a wide range of applications and widely publicized the results.

The FFT idea has been re-discovered several times in the past two centuries, before becoming a well-known standard. This is partly due to differing languages and notations.

The *Kronecker product* formalism offers a unifying basis for the description of FFT algorithms. This approach makes it easy to modify an FFT algorithm by exploiting the underlying algebraic structure of its matrix representation. This is in contrast to the usual signal flow approach where no well defined methodology for modifying FFT algorithms is available.

Van Loan used this technique for a state of the art presentation of FFT algorithms in his remarkable book "Computational Frameworks for the Fast Fourier Transform".

In the twenty-five years between the publications of Pease and Van Loan, only a few authors used this powerful technique: Temperton [12] and Johnson et

al. [13] for FFT implementations on classic vector computers and Norton and Silberger [14] on parallel computers with MIMD architecture.

Recently, Gupta and Pitsianis [15] used the Kronecker product formalism to synthesize FFT programs. As a consequence, the Kronecker product approach to FFT algorithm design antiquates more conventional techniques like signal flow graphs. Signal flow graphs rely on the spatial symmetry of a graph representation of FFT algorithms, whereas the Kronecker product exploits matrix algebra.[1]

Applications

Seismic Observation. One specific application of the discrete Fourier transform was to test, whether a seismic activity was caused by an earthquake or by a nuclear bomb. These events can be distinguished in the frequency domain, because the respective spectra have strongly differing characteristics.

There are more seismic applications of the FFT. The oil and gas industry, for example, uses the FFT as a fundamental exploration tool. With the so-called "seismic reflection method" it is possible to map sub-surface sedimentary rock layers.

Filtering. Filters are used to attenuate or enhance certain frequency components of a given signal. There are low pass, high pass and band pass filters. But there are even more filter issues that can be addressed with FFTs, for example the complexity reduction of finite impulse responses by providing filter convolution.[2]

Image Processing. Filters can also be applied to two- and more-dimensional signals like digital pictures. They can be smoothed, sharp edges can be enhanced as well as disturbing background noise can be reduced. Especially for high-noise medical images (like some X-ray pictures) the FFT can be used to enhance quality and visibility. Furthermore, the spectral representation of a digital picture can be used to gain valuable information for pattern-recognition purposes.

Advanced image processing techniques are used also for fingerprint analysis. Using FFTs, the frequencies corresponding to the regular lines of a check can be found and deleted, minimizing distortion. With the background removed, the lines that make up the fingerprint become visible.[3]

Data Communication. The FFT is usually associated with low level aspects of communication. For instance, to understand how a signal will behave when

[1] Introduction to the Fourier theory is given at
`http://aurora.phys.utk.edu/ forrest/papers/fourier/`.

[2] General information about the FFT and many FFT applications can be found at
`http://www.spd.eee.strath.ac.uk/ interact/FFT/applicns/comms.html`.

[3] Information on fingerprint analysis available at
`http://www.mediacy.com/notes/an130.htm`.

sent through communication channels, amplifiers etc. Especially the degree of distortion can be modeled easily knowing the bandwidth of the channel and the spectrum of the signal. For example, if a signal is made up of certain frequency components but only part of them pass through the channel, adding up only those components passing results in a good approximation to the distorted signal.

Astronomy. Sometimes it is not possible to get all the required information from a "normal" telescope. In such situations radio waves or radar are used instead of light. The radar signals are treated just like any other time varying voltage signal and can be processed digitally. For example, the satellite Magellan, released in 1989 and sent to earth's closest planet, Venus, was equipped with modern radar and digital signal processing capabilities and provided excellent data. These data were used to create a computer-generated virtual flight above planet Venus.[4]

Optics. In optical theory the signal is the oscillatory electric field at a point in space where light passes by. The Fourier transform of the signal is equivalent to breaking up the light into its components by using a prism. The Fourier transform is also used to calculate the diffracted intensity with the experiments of light passing through narrow slits (even Young's famous double slit experiment made use of the Fourier Transform). These ideas can be applied to all kinds of wave analysis applications like acoustic, X-ray, and microwave diffraction.

Speech Recognition. The field of digital speech processing and recognition is a multi-million Euro business by now. Advances in computational speed and new language models have made speech recognition possible on simple PCs. Today, speech recognition is a very wide field, consisting of isolated word recognition for highly specialized fields as medicine as well as connected word and even conversational speech recognition.

Speech recognition is investigated for many reasons. Automatic typewriters are the most obvious ones, but the easiness of use (of an eventually working system) and its possible speed over other information input are worth mentioning as well. Speech is rather independent from other activities involving hands or legs. Many tasks could be performed by computers, when the recognition process were reliable and cheap. For achieving both goals the FFT is an important tool.

The difficulties of speech recognition can be summarized as follows:

So called phonemes, the parts building the words, have to be extracted and recognized from a sampled input. Unfortunately, they do not seem to have invariant characteristics, which would allow for an easy classification. Instead, phonemes influence each other in consequence of coarticulation, background noise can influence the recording, and any speaker has very distinct characteristics. Different

[4] National Aeronautics and Space Administrations pages at
`http://www.nasa.gov` and
`http://www.nasa.gov/gallery/photo/index.html`.

phonemes can even have the same spectral characteristics when spoken by two distinct speakers.

But these are just low-level problems. While a listener can usually predict the next word according to several linguistic constrains and can compensate incomplete phonetic information, digital speech processing does not have this ability. Education, mood, as well as time of day may influence human speaking and recognition. Communication with people you know very well can be done efficiently using only a few words. But a computer can utilize this background information only by analyzing the whole process.

Because of all these problems, methods from distant fields as neural networks, digital signal processing and knowledge databases are currently being combined to allow for the apparently simple task of getting information out of some sounds and words.

The FFT plays a small, but important role in the recognition process. It is used to transform the signal, usually after a first filtering process, into the frequency domain to obtain its spectrum, because the critical features for perceiving speech by the human ear are mainly included in the spectral information, while the phase information does not play a crucial role.

Not only can the FFT transform the signal into an easier-to-handle form, it is even cheap in time and makes real-time recognition economically affordable.

The whole process consists of a number of small steps: (1) Speech (sound signal) omitted by speaker. (2) Low-pass filtering, sampling and quantization. (3) Frame extraction and windowing. (4) Spectral analysis, usually done with the FFT. (5) Feature extraction and classification of phonemes. (6) Use of probability models like hidden Markov models to recognize words out of phoneme sequences. (7) Use of linguistic models to choose the correct words among some that sound identical.

Usually this real-time process yields better results when the vocabulary is small, when the speaker is known, and when the system was trained to recognize his voice. Recent developments constantly improved continuous recognition quality, and there are useful software packages commercially available.[5]

Other Applications. The Fourier transform (especially the discrete Fourier transform) is also used for solving partial differential equations and is therefore, for example, essential to the field of computational fluid dynamics. Other practical applications include vibration analysis, meteorology and speech synthesis. Some compression algorithms use the Fourier transform to save storage space—they just store the relevant parts of the spectrum.

[5] More on speech recognition at
`http://coral.lili.uni-bielefeld.de/Classes/Summer96/Acoustic/acoustic2/acoustic.html`.

1.1 The Fast Fourier Transform

1.1.1 The Fourier Transform

The Fourier transform (FT) essentially decomposes a waveform, signal or function into sinusoids of different frequency whose sum reproduces the original function.

The *Fourier transform* F of a function f is defined as

$$F(\omega) := \int_{-\infty}^{\infty} f(t) e^{2\pi i \omega t} dt \tag{1.1}$$

provided the integral exists as a Cauchy principal value[6] for any $\omega \in \mathbb{R}$. This transform maps the function $f(t)$, a signal taken as a function of time, into a complex valued function $F(\omega)$ which represents the signal as a function of frequency.

The inverse operation to (1.1) is

$$f(t) = \int_{-\infty}^{\infty} F(\omega) e^{-2\pi i \omega t} \, d\omega, \tag{1.2}$$

i. e., the *inverse Fourier transform*. Using (1.2) signals can be (re-)transformed from the frequency domain into the time domain.

The representation of a signal as a function of time is also said to be a representation in the *time domain*; the representation of a signal as a function of frequency is said to be a representation in the *frequency domain*. Both representations portray the same signal. Often the representation in the time domain appears "more natural" (in particular when the signal is taken or measured in this form), whereas the representation in the frequency domain is more suitable for filtering purposes and for any kind of manipulation of the spectrum.

Continuous functions f do not appear often in technical applications, instead individual observations (*samples*) of such functions at certain time points occur. In the following it is assumed that the sampling of f is carried out at equidistant time points whose distance is the sampling interval Δ. The quantity $1/\Delta$ is the *sampling rate*; it gives the number of discrete values of f per time unit.

Definition 1.1 (Nyquist Frequency) *For any sampling interval* Δ,

$$\omega_c := \frac{1}{2\Delta}$$

denotes the Nyquist frequency.

[6] The Cauchy principal value of $\int_{-\infty}^{\infty} x(t)\,dt$ is defined as $\lim_{A\to\infty} \int_{-A}^{A} x(t)\,dt$.

original function	Fourier transform
real and even	real and even
real and odd	imaginary and odd
imaginary and even	imaginary and even
complex and even	complex and even
complex and odd	complex and odd
real and asymmetric	complex and asymmetric
imaginary and asymmetric	complex and asymmetric
real even plus imaginary odd	real
real odd plus imaginary even	imaginary
even	even
odd	odd

Table 1.1: Symmetry properties of the Fourier transform.

The importance of this quantity is made clear in the following section.

1.1.2 The Discrete Fourier Transform

The following considerations deal with the Fourier transform of discrete (sampled) data sequences. Assuming that there are N data points, where, for sake of simplicity, it is presupposed that N is even (although all considerations remain valid when N is odd):

$$f_k := f(t_k), \qquad t_k := (k_0 N + k)\Delta, \qquad k = 0, 1, \ldots, N-1\,.$$

The form $k_0 N$ has been chosen because of notational simplifications.

For a piecewise continuous functions f with

$$\int_{-\infty}^{\infty} |f(t)|\, dt \;<\; \infty,$$

the Fourier transform F always exists. In this case the infinite integral (1.1) can be approximated by a finite sum (for a suitable selection of N, Δ and k_0) using numerical integration:

$$F(f) = \int_{-\infty}^{\infty} f(t) e^{2\pi i \omega t} dt \;\approx\; \sum_{k=0}^{N-1} f_k e^{2\pi i \omega t_k} \Delta.$$

Moreover, f can be interpolated, according to Theorem 1.1, using a continuous function g with

$$\lim_{t\to\pm\infty} g(t) = 0$$

and

$$G(\omega) = 0 \qquad \text{for} \quad \omega \notin (-\omega_c, \omega_c)$$

at the points t_k, i. e.

$$f(t_k) = g(t_k), \qquad k = 0, 1, \ldots, N-1\,.$$

The Fourier transform F of f, however, is now represented approximately by estimates of its function values at the points

$$\omega_n := \frac{n}{N\Delta}, \qquad n = -\frac{N}{2},\, -\frac{N}{2}+1,\, \ldots,\, \frac{N}{2}-1\,:$$

$$F(\omega_n) \;\approx\; \Delta \sum_{k=0}^{N-1} f_k e^{2\pi i \frac{n}{N\Delta}(k_0 N+k)\Delta} = \Delta \sum_{k=0}^{N-1} f_k e^{2\pi i k n/N}. \tag{1.3}$$

Using this formula, N discrete function values f_k lead to N discrete frequencies ω_n. Thus, instead of determining the Fourier transform $F(\omega)$ in the range $[-\omega_c, \omega_c]$, $F(\omega)$ is only determined for the discrete frequencies

$$\omega_n := \frac{n}{N\Delta}, \qquad n = -\frac{N}{2},\, -\frac{N}{2}+1,\, \ldots,\, \frac{N}{2}-1.$$

The formula (1.3) is the *discrete Fourier transform* (DFT), and in this thesis it is always denoted F_n (which leaves out the factor Δ):

$$F_n := \sum_{k=0}^{N-1} f_k e^{2\pi i k n/N}\,. \tag{1.4}$$

This transform is periodic in n with the period N, i. e., $F_{N+n} = F_n$ is valid for any $n \in \mathbb{Z}$.

The connection between the continuous Fourier transform F of the function f and the discrete Fourier transform F_n of the data f_k, obtained by sampling f with a sampling interval Δ is represented as follows:

$$F(\omega_n) \approx \Delta F_n. \tag{1.5}$$

In (1.5) $n = 0$ corresponds to the frequency $\omega = 0$; positive frequencies $0 < \omega < \omega_c$ correspond to $1 \le n \le N/2-1$; and negative frequencies $-\omega_c < \omega < 0$ correspond to $N/2+1 \le n \le N-1$. For $n = N/2$, n corresponds both to the frequencies ω_c and $-\omega_c$.

The *inverse discrete Fourier transform* (inverse DFT), which can be derived from (1.2) analogously to (1.3), is given by:

$$f_k = \frac{1}{N} \sum_{n=0}^{N-1} F_n e^{-2\pi i k n/N}. \tag{1.6}$$

The only differences between (1.6) and (1.4) are the negative sign of the exponent and the division by N. Therefore, for the calculation of the discrete Fourier transform and its inverse, the same routines can be used to a large extent.

For a sinusoidal wave the Nyquist frequency ω_c is the highest frequency that is still reconstructable at a fixed length of the sampling interval Δ because, on the one hand, the argument of the expression of the form $\sin(2\pi\omega_0 t)$ with a frequency $\omega_0 \leq 1/(2\Delta)$ can move 2 quadrants, at the most, from one sampling to the next on the unit circle; and if, on the other hand, for the given sampling points, there exists a sinusoidal wave with a frequency larger than $1/(2\Delta)$, then there must always be one sinusoidal wave with a frequency smaller than $1/(2\Delta)$, since

$$\begin{aligned} &\sin\left[2\pi\left(\frac{1}{2\Delta}+\varepsilon\right)(t_0+k\Delta)\right] \\ &= -\sin\left[2\pi\left(\frac{1}{2\Delta}-\varepsilon\right)\left(\frac{t_0}{\varepsilon\Delta-\frac{1}{2}}+(t_0+k\Delta)\right)\right] \end{aligned}$$

for all $t_0, \epsilon \in \mathbb{R}$ and $k \in \mathbb{Z}$. Thus, at least two sample points per period are required in order to reconstruct a sine wave correctly.

Another restriction on the sampling of a function with the frequency ω appears if $N\omega/2\omega_c$ is not an integer. Then, in the discrete spectrum none of the discrete frequencies $\omega_n = 2\omega_c n/N$ appears. The frequency in the discrete spectrum which is next to the original frequency ω is clearly the largest; however, all other frequencies also appear more or less strongly in the spectrum. This phenomenon is called *leakage*.

Theorem 1.1 (Shannon's Sampling Theorem) *Let f denote a function with $\int_{-\infty}^{\infty} |f(t)|^2\, dt < \infty$, i. e., a signal with finite energy which is sampled at the rate $1/\Delta$. If f is band limited in its continuous frequency spectrum by the Nyquist frequency $\omega_c = 1/(2\Delta)$, i. e., if the Fourier transform F satisfies $F(\omega) = 0$ for any ω with $|\omega| \geq \omega_c$, then the function f can be recovered exactly from its sample values using the interpolation function*

$$g(t) = \frac{\sin(2\pi\omega_c t)}{2\pi\omega_c t}. \tag{1.7}$$

Thus, f may be expressed as

$$f(t) = \sum_{n=-\infty}^{\infty} f_n g(t - n\Delta), \tag{1.8}$$

where $f_n := f(n\Delta)$ are the samples of f.

If the function f is not band-limited, then in the Fourier transform of f all components of the frequency spectrum that lie outside of the range $[-\omega_c, \omega_c]$ are moved into this frequency range. This spectral overlap is called *aliasing*. In this case the function can not be completely reconstructed using its sample values.

Thus, the length Δ of the sampling interval has to be selected so that the critical frequency ω_c is higher than all frequencies appearing in the spectrum of the data. Whether this condition applies to certain data can be seen from the frequency spectrum $F(\omega)$ of the data decreasing to zero, when the frequency ω tends to ω_c. If this is not the case, then a remedial precaution can be taken by shortening the sampling interval or by restricting the signal in its frequency spectrum (for instance, by using low-pass filtering) before sampling.

Another approach to understanding the DFT is to consider the transform as a simple change of coordinates. The column vectors of the transformation matrix are orthogonal and represent different frequency components. When multiplying the system matrix with the inverse matrix, the components of the solution vector indicate, how these column vectors compose the input.

When analyzing real-world data in form of a signal, this signal is represented (exactly or approximately) by a finite sequence of numbers. Computers are then used to calculate transformations of those signals. This is done by applying the *discrete Fourier transform (DFT)*.

The following matrix notation is used to increase readability and to understand the FFT idea better. It will lead finally to the powerful Kronecker notation. Furthermore, the variables t and ω are replaced by indexing the vectors (arrays) x and y.

The DFT vector

$$y = (y_0, \ldots, y_{N-1})^\top \in \mathbb{C}^N$$

of the data vector

$$x = (x_0, \ldots, x_{N-1})^\top \in \mathbb{C}^N$$

is defined by

$$y_k := \sum_{j=0}^{N-1} \omega_N^{kj} x_j, \quad k = 0 : N-1, \tag{1.9}$$

where

$$\omega_N := \cos(2\pi/N) - i\,\sin(2\pi/N) = e^{-2\pi i/N}, \quad i = \sqrt{-1}.$$

The powers of ω_N are called *twiddle-factors* .

In matrix-vector terms, the DFT can be written as the matrix-vector product

$$y = F_N x.$$

The elements of of the matrix $F_N \in \mathbb{C}^{N\times N}$ are given by

$$[F_N]_{k,j} := \omega_N^{kj} = e^{-2\pi ikj/N}, \quad k, j = 0 : N-1.$$

Example: The DFT matrices F_1, F_2, and F_4 are given by

$$F_1 = (1), \quad F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}.$$

The evaluation of a matrix-vector product needs $O(n^2)$ floating-point operations, resulting in quadratically increasing calculation times.

1.1.3 The Fast Fourier Transform

The DFT is one of the most important tools in modern engineering. Therefore the algorithm was continuously optimized over the years. The key improvement, which reduced computation time and costs dramatically, was to exploit the intrinsic symmetry of the DFT matrix. The breakthrough algorithm of Cooley-Tukey enabled a reduction of the number of operations to be carried out to $\text{const}\times N\log N$—the constant varying between 3 and 5 depending on the specific variant of the algorithm used.

In applications where the DFT has to be performed many times (for example, in meteorology) and in applications where large transformation lengths are needed (especially in seismic applications) the speed-up is invaluable.[7]

[7]For the vector length $N = 2^{22} = 4,194,304$ the execution time can be reduced from 5 hours to 100 milliseconds! Note, however, that there are factors which influence ideal execution

N	N^2	$N \log_2 N$	DFT (sec)	FFT (sec)	Speed-up
4	1.60×10^1	8.00×10^0	1.60×10^{-8}	8.00×10^{-9}	2
16	2.56×10^2	6.40×10^1	2.56×10^{-7}	6.40×10^{-8}	4
64	4.10×10^3	3.84×10^2	4.07×10^{-6}	3.84×10^{-7}	11
256	6.55×10^4	2.05×10^3	6.55×10^{-5}	2.05×10^{-6}	32
1,024	1.05×10^6	1.02×10^4	1.05×10^{-3}	1.02×10^{-5}	102
4,096	1.68×10^7	4.92×10^4	1.68×10^{-2}	4.92×10^{-5}	341
16,384	2.68×10^8	2.29×10^5	2.68×10^{-1}	2.29×10^{-4}	1,170
65,536	4.29×10^9	1.05×10^6	4.30×10^0	1.05×10^{-3}	4,096
262,144	6.87×10^{10}	4.72×10^6	6.87×10^1	4.72×10^{-3}	14,564
1,048,576	1.10×10^{12}	2.10×10^7	1.10×10^3	2.10×10^{-2}	52,429
4,194,304	1.76×10^{13}	9.23×10^7	1.76×10^4	9.23×10^{-2}	190,650

Table 1.2: Theoretical execution times of the "classical" DFT and the FFT on a 1 GHz processor running at peak performance.

The key idea behind the Cooley-Tukey algorithm is to use the divide and conquer paradigm. This idea can be explained by means of the 8×8 DFT matrix

$$F_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{pmatrix},$$

where $\omega := \omega_8 = e^{-2\pi i/8}$.

Any matrix F_N, N even, can be rearranged by the perfect shuffle permutation Π_N which groups the even-indexed columns first and then the odd-indexed columns second:

$$F_N \Pi_N = (F_N(:, 0 : N-2 : 2) | F_N(:, 1 : N-1 : 2)).$$

Rearranging the columns of F_8 in this manner

times. These factors will be subject of a more detailed discussion in later chapters of this thesis. For instance, the operations of an FFT algorithm are *complex* multiplications and additions, requiring a larger number of real floating-point instructions. Then, there are dependencies between the instructions, which may inhibit the completion of one operation at every cycle. Finally, there is overhead caused by the function calls and there are memory latencies due to cache misses, which may increase ideal execution times substantially.

$$F_8\Pi_8 = \left(\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline 1 & 1 & 1 & 1 & \omega^4 & \omega^4 & \omega^4 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^5 & \omega^7 & \omega & \omega^3 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^6 & \omega^2 & \omega^6 & \omega^2 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^7 & \omega^5 & \omega^3 & \omega \end{array}\right)$$

establishes a connection between F_8 and F_4. Using the fact that $\omega_8^2 = \omega_4$ is a 4th root of unity, and therefore

$$\left(\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 \\ 1 & \omega_8^4 & 1 & \omega_8^4 \\ 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 \end{array}\right) = \left(\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & 1 & \omega_4^2 \\ 1 & \omega_4^3 & \omega_4^2 & \omega_4 \end{array}\right) = F_4,$$

leads to the partition

$$F_8\Pi_8 = \left(\begin{array}{cc} F_4 & \Omega_4 F_4 \\ F_4 & \omega_8^4 \Omega_4 F_4 \end{array}\right),$$

where $\Omega_4 = \mathrm{diag}(1, \omega_8, \omega_8^2, \omega_8^3)$. Given the fact that $\omega_8^4 = -1$, the factorization

$$F_8\Pi_8 = \left(\begin{array}{cc} I_4 & I_4 \\ I_4 & -I_4 \end{array}\right) \left(\begin{array}{cc} I_4 & \\ & \Omega_4 \end{array}\right) \left(\begin{array}{cc} F_4 & \\ & F_4 \end{array}\right)$$

is obtained.

Now, both of the smaller transforms can be split up again into two transforms of half the original size. This kind of splitting up can be performed $\log_2 N$ times. Each of these steps involves the whole input vector, on which $O(N)$ operations are performed. Therefore the total arithmetic complexity is $O(N \log_2 N)$.

The Cooley-Tukey idea can be programmed easily using a recursive approach, which in pseudo-code, would have the following form (if the permutation of the input vector is already done):

Algorithm 1.1 (Recursive FFT)

if *length of input vector equals* 1 **then return**
$u :=$ **fft** *of upper part of input*
$l :=$ **fft** *of lower part of input*
$l := l \times$weight_vector
overwrite upper part of output with $u + l$
overwrite lower part of output with $u - l$
return output

Unfortunately, such recursive algorithms have some unfavorable characteristics. For example, each recursive call needs additional memory. Therefore these algorithms are normally not used for real applications, but they are important tools to understand the Cooley-Tukey concept. Later on a non-recursive version of the above algorithm will be given together with the respective C code.

1.1.4 Definitions

Definition 1.2 (Stride Permutation) *For a vector* $x \in \mathbf{C}^{mn}$ *the stride permutation* L_n^{mn} *is defined by*

$$L_n^{mn} x := \begin{pmatrix} x(0 : mn-1 : n) \\ x(1 : mn-1 : n) \\ \vdots \\ x(n-1 : mn-1 : n) \end{pmatrix}.$$

The permutation operator L_n^{mn} sorts the components of x according to their index modulo n. Thus, components with indices equal to $0 \bmod n$ come first, followed by the components with indices equal to $1 \bmod n$, and so on.

The permutation $y := L_2^n x$ (n even) is called an *even-odd sort permutation*, because it groups the even-indexed and odd-indexed components together.

The permutation $y := L_{n/2}^n x$ (n even) is called a *perfect shuffle permutation*, since its action on a deck of cards could be the shuffling of two equal piles of cards so that the cards are interleaved one from each pile. Because of its importance, the perfect shuffle permutation $L_{n/2}^n$ is denoted in short by Π_n.

Example: $y := L_4^8 x = \Pi_8 x$ is given by

$$\Pi_8 x = (x_0\ x_4\ x_1\ x_5\ x_2\ x_6\ x_3\ x_7)^\top.$$

Stride permutations have many important algebraic properties:

Property 1.1

$$(L_n^{mn})^{-1} = L_m^{nm}$$

Example:

$$(L_2^{2^i})^{-1} = L_{2^{i-1}}^{2^i} = \Pi_{2^i}.$$

Definition 1.3 (Complex Arithmetic Complexity) *Let $\mu_C \in \mathbb{N}$ ($\alpha_C \in \mathbb{N}$) denote the number of complex multiplications (additions) needed to perform a given numerical computation. The complex arithmetic complexity $\pi_C \in \mathbb{N}$ is defined by*

$$\pi_C := \mu_C + \alpha_C.$$

Definition 1.4 (Real Arithmetic Complexity) *Let $\mu_R \in \mathbb{N}$ ($\alpha_R \in \mathbb{N}$) denote the number of real multiplications (additions) needed to perform a given numerical computation. The real arithmetic complexity $\pi_R \in \mathbb{N}$ is defined by*

$$\pi_R := \mu_R + \alpha_R.$$

A complex multiplication needs 4 real multiplications and 2 real additions (4 + 2 method) and a complex addition needs 2 real additions. Thus, a fixed connection between complex arithmetic and real arithmetic complexity can be established:

$$\begin{aligned} \pi_R &= 6\mu_C + 2\alpha_C, \\ \mu_R &= 4\mu_C, \\ \alpha_R &= 2\mu_C + 2\alpha_C. \end{aligned}$$

A complex multiplication can alternatively be done with 3 real multiplications and 5 real additions (3 + 5 method), according to

$$(a + ib) \cdot (c + id) = [(a + b) \cdot (c - d) + a \cdot d - b \cdot c] + i[a \cdot d + b \cdot c].$$

In this thesis the "4 + 2 method" for performing complex multiplications is used.

As already mentioned, modern RISC processors provide a special instruction to perform a combination of a multiplication and an addition operation (*fused multiply-add*)—the operation $(a \times b) + c$—within the same time as needed by a single floating-point addition or multiplication—with the additional advantage that only *one* rounding error occurs.

Definition 1.5 (Multiply-Add Arithmetic Complexity) π_{fma} *denotes the number of floating-point operations executed by an algorithm. Any fused multiply-add operation counts as a single instruction.*

On a computer with FMA architecture a nontrivial complex multiplication requires 4 FMA instructions (2 multiplications and 2 multiply-adds) and a complex addition requires 2 FMA instructions.

Note, that there is no simple relation between the complex arithmetic complexity and the multiply-add complexity π_{fma}, like the following example illustrates:

$$\begin{aligned} c_1 &= c_2 \times c_3 + c_4, \\ c_1 &= c_2 \times c_3, \quad c_4 = c_5 + c_6. \end{aligned}$$

The statements of both lines have the same complex arithmetic complexity, but the first line requires the execution of 4 fused multiply-add instruction, the second line needs 6 FMA instructions.

There are two interesting special cases, where operations are combined without increasing the overall multiply-add complexity.

Definition 1.6 (Cheap Operations) *On a multiply-add processor the addition of certain arithmetic operations does not increase* π_{fma}*:*

$$\begin{aligned} c_1 &= c_2 \times c_3 \quad \longrightarrow \quad c_1 = c_2 \times c_3 + c_4, \quad c_i \quad \textit{complex} \\ c_1 &= c_2 + c_3 \quad \longrightarrow \quad c_1 = r \times c_2 + c_3, \quad r \quad \textit{real.} \end{aligned}$$

Such operations are called cheap.

Compilers are quite successful in finding out the first case of cheap operations, and in rearranging the corresponding sequences of operations.

Although the multiply-add complexity is not affected by combining cheap operations, program performance might decrease as FMA instructions may have higher latencies due to an increased number of memory accesses, which may entail an increased number of secondary cache misses.

Some complex operations of FFT algorithms involve trivial operands or arguments, like 0 or 1. Multiplication by such factors can be avoided at different levels:

- At the compiler level, references to constant trivial operands can be detected and operations eliminated. Most compilers perform this task automatically, some only at higher optimization levels.
- Functions of some libraries do not perform any operation at all when called with trivial arguments, like sin(0).
- At the hardware level, trivial operations can be detected, though this is unusual, as trivial operations are not the common case and the increased chip complexity has a negative performance impact.

- At the algorithm level, clever programs avoid predictable operations with trivial factors. This kind of algorithm design enhances the performance of FFT algorithms considerably.

There is a special trivial complex operand, i, with a special characteristic: Multiplication operations involving i can be done by exchanging the real and imaginary part of a number with an additional negation. Thus, only one register swap and one floating-point operation are needed.

Definition 1.7 (Memory Accesses) *Let $\lambda \in \mathbb{N}$ ($\sigma \in \mathbb{N}$) denote the number of loads (stores) needed to perform a given numerical computation. The total number of memory accesses, $\gamma \in \mathbb{N}$, is given by*

$$\gamma := \lambda + \sigma.$$

Definition 1.8 (Normalized Execution Time) *The normalized execution time $\overline{T}$ is given by*

$$\overline{T} := \frac{T}{N \log N}$$

where T denotes the runtime.

Definition 1.9 (Efficiency) *The efficiency is the ratio of arithmetic operations performed to the number that could be performed. Thus, the efficiency is given by*

$$E := \frac{\pi_R}{T \times \text{PeakFP}} \times 100\,[\%]$$

where π_R denotes the number of floating-point operations, T the runtime, and PeakFP *the theoretical peak performance.*

1.1.5 Kronecker Products

Definition 1.10 (Kronecker Product) *The Kronecker product of the matrices $A \in \mathbb{C}^{m_1 \times n_1}$ and $B \in \mathbb{C}^{m_2 \times n_2}$ is the block structured matrix*

$$A \otimes B := \begin{pmatrix} a_{0,0}B & \dots & a_{0,n_1-1}B \\ \vdots & \ddots & \vdots \\ a_{m_1-1,0}B & \dots & a_{m_1-1,n_1-1}B \end{pmatrix} \in \mathbb{C}^{m_1 m_2 \times n_1 n_2}.$$

Kronecker products have the following properties.

Property 1.2 (Associativity) *If A, B, C are arbitrary matrices, then*

$$(A \otimes B) \otimes C = A \otimes (B \otimes C).$$

Thus, the expression $A \otimes B \otimes C$ is unambiguous.

Property 1.3 (Multiplicative Property) *If A, B, C, D are arbitrary matrices, then*

$$(A \otimes B)\,(C \otimes D) = A\,C \otimes B\,D,$$

provided the products AC and BD are defined.

A consequence of this property is the following factorization.

Corollary 1.1 (Decomposition) *If $A \in \mathbb{C}^{m_1 \times n_1}$ and $B \in \mathbb{C}^{m_2 \times n_2}$, then*

$$A \otimes B = AI_{n_1} \otimes I_{m_2}B = (A \otimes I_{m_2})(I_{n_1} \otimes B),$$

$$A \otimes B = I_{m_1}A \otimes BI_{n_2} = (I_{m_1} \otimes B)(A \otimes I_{n_2}).$$

Property 1.4 (Transposition) *If A, B are arbitrary matrices, then*

$$(A \otimes B)^\top = A^\top \otimes B^\top.$$

Property 1.5 *If $A \in \mathbb{C}^{r \times r}$ and $x \in \mathbb{C}^n$ with $n = rc$, then*

$$y = (I_c \otimes A)x \quad \Leftrightarrow \quad y_{r\times c} = Ax_{r\times c}.$$

Property 1.6 *If $A \in \mathbb{C}^{r \times r}$ and $x \in \mathbb{C}^n$ with $n = rc$, then*

$$y = (A \otimes I_c)x \quad \Leftrightarrow \quad y_{r\times c} = x_{r\times c}A^\top.$$

1.1.6 FFT Algorithms in Kronecker Product Notation

Using

$$\Omega_{N/2} := \mathrm{diag}(1, \omega_N, \ldots, \omega_N^{N/2-1}), \qquad N \text{ even},$$

the four symmetry conditions $(k, j = 0 : N/2)$

$$
\begin{array}{llll}
[F_N \Pi_N]_{k,j} & = \omega_N^{k(2j)} & = \omega_{N/2}^{kj} & = [F_{N/2}]_{k,j} \\
[F_N \Pi_N]_{k+N/2,j} & = \omega_N^{(k+N/2)(2j)} & = \omega_{N/2}^{(k+N/2)j} & = [F_{N/2}]_{k,j} \\
[F_N \Pi_N]_{k,j+N/2} & = \omega_N^{k(2j+1)} & = \omega_N^k \omega_{N/2}^{kj} & = [\Omega_{N/2} F_{N/2}]_{k,j} \\
[F_N \Pi_N]_{k+N/2,j+N/2} & = \omega_N^{(k+N/2)(2j+1)} & = -\omega_N^{k(2j+1)} & = [-\Omega_{N/2} F_{N/2}]_{k,j}
\end{array}
$$

imply the *radix-2 splitting*:

$$
F_N \Pi_N = \begin{pmatrix} F_{N/2} & \Omega_{N/2} F_{N/2} \\ F_{N/2} & -\Omega_{N/2} F_{N/2} \end{pmatrix}.
$$

These relations can be easily established with simple computations due to the fact that $\omega_N^2 = \omega_{N/2}$ and $\omega_N^{N/2} = -1$.

The term "radix-2 splitting" indicates that this relation establishes a connection between the full-sized DFT matrix F_N and the half-sized DFT matrix $F_{N/2}$. Recursive application of this splitting process is the heart of all radix-2 FFT algorithms. More generally, if p divides N, it is possible to relate F_N to $F_{N/p}$.

Thus, F_N can be factorized as

$$
F_N = \begin{pmatrix} I_{N/2} & I_{N/2} \\ I_{N/2} & -I_{N/2} \end{pmatrix} \begin{pmatrix} I_{N/2} & \\ & \Omega_{N/2} \end{pmatrix} \begin{pmatrix} F_{N/2} & \\ & F_{N/2} \end{pmatrix} L_2^N.
$$

This factorization can be expressed in terms of Kronecker products.

Theorem 1.2 (Cooley-Tukey Factorization) *For $N \geq 2$*

$$
F_N = (F_2 \otimes I_{N/2}) T_N (I_2 \otimes F_{N/2}) L_2^N,
$$

$$
T_N = (I_{N/2} \oplus \Omega_{N/2}),
$$

where $\oplus$ denotes the direct matrix sum operator.

For $N = 2^n$, repeated application of Theorem 1.2 (Cooley-Tukey factorization) leads to the following factorization (Johnson et al. [13]).

Theorem 1.3 (Cooley-Tukey Radix-2 FFT) *It holds that*

$$
F_{2^n} = \Big[\prod_{i=1}^{n} (I_{2^{i-1}} \otimes F_2 \otimes I_{2^{n-i}})(I_{2^{i-1}} \otimes T_{2^{n-i+1}}) \Big] R_{2^n}, \tag{1.10}
$$

where

$$
R_{2^n} := \prod_{i=2}^{n} (I_{2^{n-i}} \otimes L_2^{2^i}).
$$

The permutation matrix R_{2^n} is called a *bit reversal matrix*.

By using the Kronecker product property 1.3 (multiplicative property) the expression

$$(I_{2^{i-1}} \otimes F_2 \otimes I_{2^{n-i}})(I_{2^{i-1}} \otimes T_{2^{n-i+1}})$$

can be written as

$$I_{2^{i-1}} \otimes ((F_2 \otimes I_{2^{n-i}})(I_{2^{n-i}} \oplus \Omega_{2^{n-i}})),$$

where

$$(F_2 \otimes I_{2^{n-i}})(I_{2^{n-i}} \oplus \Omega_{2^{n-i}}) = \begin{pmatrix} I_{2^{n-i}} & \Omega_{2^{n-i}} \\ I_{2^{n-i}} & -\Omega_{2^{n-i}} \end{pmatrix} =: B_{2^{n-i+1}}.$$

The matrix $B_{2^{n-i+1}}$ is called a *radix-2 Butterfly matrix*.

Thus, (1.10) can be expressed as

$$F_{2^n} = \Big[\prod_{i=1}^{n} (I_{2^{i-1}} \otimes B_{2^{n-i+1}})\Big] R_{2^n}. \tag{1.11}$$

A major drawback of the Cooley-Tukey approach is the need of the bit reversal. This operation, represented in Kronecker notation by the bit reversal matrix, does not perform well on vector processors. The design of such processors enables them to calculate multiple FFTs concurrently, but the bit reversal step still performs badly.

The *transposed Stockham factorization*

$$F_{2^n} = \prod_{q=n}^{1} \Big[(I_{2^{n-q}} \otimes B_{2^q})(\pi_{2^{n-q+1}} \otimes I_{2^{q-1}})\Big] \tag{1.12}$$

avoids bit reversal and is therefore suitable for vector processors. One of the most used implementations of the transposed Stockham algorithm was developed by P. N. Swartztrauber [16], and implemented by R. Sweet as part of the VFFTpack library.

1.1.7 Radix-4 Kernels

Theorem 1.4 (Cooley-Tukey Radix-4 Factorization)

$$F_{4^n} = \Big[\prod_{i=1}^{n}(I_{4^{i-1}} \otimes F_4 \otimes I_{4^{n-i}})(I_{4^{i-1}} \otimes T_{4^{n-i}}^{4^{n-i+1}})\Big] R_{4^n}, \tag{1.13}$$

where

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix},$$

$$T_{4^{n-i}}^{4^{n-i+1}} = \operatorname{diag}(I_{4^{n-i}}, \Omega_{4,4^{n-i}}, \Omega_{4,4^{n-i}}^2, \Omega_{4,4^{n-i}}^3),$$

and

$$R_{4^n} := \prod_{i=1}^{n-1}(I_{4^{i-1}} \otimes L_4^{4^{n-i+1}}).$$

By using (1.3) the expression

$$(I_{4^{i-1}} \otimes F_4 \otimes I_{4^{n-i}})(I_{4^{i-1}} \otimes T_{4^{n-i}}^{4^{n-i+1}})$$

can be written as

$$I_{4^{i-1}} \otimes ((F_4 \otimes I_{4^{n-i}}) \operatorname{diag}(I_{4^{n-i}}, \Omega_{4,4^{n-i}}, \Omega_{4,4^{n-i}}^2, \Omega_{4,4^{n-i}}^3)),$$

where

$$((F_4 \otimes I_{4^{n-i}}) \operatorname{diag}(I_{4^{n-i}}, \Omega_{4,4^{n-i}}, \Omega_{4,4^{n-i}}^2, \Omega_{4,4^{n-i}}^3)) =: B_{4,4^{n-i+1}}.$$

$B_{4,4^{n-i+1}}$ is the *radix-4 butterfly matrix*. Using this matrix, (1.13) can be expressed as

$$F_{4^n} = \Big[\prod_{i=1}^{n}(I_{4^{i-1}} \otimes B_{4,4^{n-i+1}})\Big] R_{4^n}. \tag{1.14}$$

If $x \in \mathbb{C}^N$ and $N = 4^n$, then $x := F_{4^n} x$ can be implemented as

Algorithm 1.2 ($x := F_{4^n} x$)

$x := R_{4^n} x$
do $i = 1 : n$
  $L := 4^i$
  $r := N/L$
  do $k = 0 : r - 1$
    $x(kL : (k+1)L - 1) := B_{4,L} x(kL : (k+1)L - 1)$
  end do
end do

Radix-4 Butterfly Computation. The radix-4 butterfly update can be calculated using the following algorithm

Algorithm 1.3 ($x := B_{4,L} x$)

do $j = 0 : L/4 - 1$

$a := x(j);$	$b := \omega_L^j x(j + L/4)$
$c := \omega_L^{2j} x(j + L/2);$	$d := \omega_L^{3j} x(j + 3L/4)$
$\tau_0 := a + c;$	$\tau_1 := a - c$
$\tau_2 := b + d;$	$\tau_3 := b - d$
$x(j) := \tau_0 + \tau_2;$	$x(j + L/4) := \tau_1 - i\tau_3$
$x(j + L/2) := \tau_0 - \tau_2;$	$x(j + 3L/4) := \tau_1 + i\tau_3$

end do

The radix-4 butterfly computation involves 3 complex multiplications ($\mu_C = 3$) and 8 complex additions ($\alpha_C = 8$), i. e., 34 real floating-point operations ($\pi_R = 34$). On an FMA architecture the computation of the radix-4 butterfly involves 6 multiply-add operations, 6 multiplications, and 16 additions. Accordingly $\pi_{\text{fma}} = 28$ and the FMA utilization amounts to a mere 21 %.

Assuming that the twiddle-factors are pre-computed, the elementary radix-4 butterfly computation involves 14 loads and 8 stores, leading to 1.54 flops per memory access.

The arithmetic complexity of the entire radix-4 FFT algorithm is

$$\pi_R = 4.25 N \log N$$

or, on a computer with an FMA processor,

$$\pi_{\text{fma}} = 3.5 N \log N.$$

1.1.8 Radix-p Kernels

This splitting idea is not restricted to dividing $N = 2^2$ and $N = 2^4$, it can be used for all $N = N_1 \cdot N_2$.

Theorem 1.5 (Fundamental Splitting) (Johnson et al.[13])
For $N = pk \geq 2$

$$F_N = (F_p \otimes I_k)T_k^{pk}(I_p \otimes F_k)L_p^{pk},$$

with

$$T_k^{pk} = \mathrm{diag}(I_k, \Omega_{p,k}, \ldots, \Omega_{p,k}^{p-1}),$$

where

$$\Omega_{p,k} := \mathrm{diag}(1, \omega_N, \ldots, \omega_N^{k-1}).$$

For $N = p^n$, repeated application of Theorem 1.5 leads to the following factorization of the DFT matrix F_{p^n}.

Theorem 1.6 (Fundamental Single-Radix Factorization)

$$F_{p^n} = \Big[\prod_{i=1}^{n}(I_{p^{i-1}} \otimes F_p \otimes I_{p^{n-i}})(I_{p^{i-1}} \otimes T_{p^{n-i}}^{p^{n-i+1}})\Big]R_{p^n}. \tag{1.15}$$

This factorization makes it possible to split a p^n-point DFT into n DFTs of size p. The number p is called the *radix* of the FFT algorithm. The permutation matrix R_{p^n} is the index reversal matrix responsible for the permutation of the input data sequence (see Van Loan [17]).

By using (1.3), i. e., the mixed-product property of the Kronecker product, the expression

$$(I_{p^{i-1}} \otimes F_p \otimes I_{p^{n-i}})(I_{p^{i-1}} \otimes T_{p^{n-i}}^{p^{n-i+1}})$$

can be written as

$$I_{p^{i-1}} \otimes ((F_p \otimes I_{p^{n-i}})\mathrm{diag}(I_{p^{n-i}}, \Omega_{p,p^{n-i}}, \ldots, \Omega_{p,p^{n-i}}^{p-1})).$$

The matrix

$$B_{p,p^{n-i+1}} := (F_p \otimes I_{p^{n-i}})\mathrm{diag}(I_{p^{n-i}}, \Omega_{p,p^{n-i}}, \ldots, \Omega_{p,p^{n-i}}^{p-1})$$

is said to be a *radix-p butterfly matrix*. Using this matrix, (1.15) becomes

$$F_{p^n} = \Big[\prod_{i=1}^{n}(I_{p^{i-1}} \otimes B_{p,p^{n-i+1}})\Big]R_{p^n}. \tag{1.16}$$

If $x \in \mathbb{C}^N$ and $N = p^n$, the FFT computation $x := F_{p^n}x$ can be implemented as

Algorithm 1.4 (Radix-p FFT)

$x := R_{p^n} x$
do $i = 1 : n$
  $L := p^i$
  $r := N/L$
  do $k = 0 : r - 1$
    $x(kL : (k+1)L - 1) := B_{p,L} x(kL : (k+1)L - 1)$
  end do
end do

The computationally most intensive part of any radix-p FFT algorithm is $x := B_{p,L}x$, i. e., the butterfly update which occurs in the innermost loop. Thus, the arithmetic complexity of any radix-p FFT algorithm depends primarily on the design and implementation of the butterfly kernel.

1.2 Hardware Implications

1.2.1 The Processor

The Clock Rate

The clock rate in one of the key performance factors of every processor, though not the only one. It is an indicator for the processors peak performance. This peak performance is hardly ever reached in real-world applications, therefore one should be careful to judge a processor only by its clock rate. Clock rates increased during the last years up to 1 GHz in modern processors. This is already very close to physical limits, like heat dissipation.

One of the reasons, that made this increase possible, was the use RISC computers. They have less instructions and a simpler design than CISC processors and can therefore be run at a faster pace.

Pipelining

The development of the pipeline concept has played an even more important role in the performance boost of processors. The idea behind it is to start with a division of each instruction into small, simple parts. A typical instruction, for example, consists of five basic steps:

1. *Instruction fetch phase*: The next instruction to be performed is read from memory to an instruction register;

2. *Decoding phase*: The instruction is interpreted and memory operand addresses are extracted;
3. *Operand fetch phase*: The operands are moved to the registers;
4. *Execution phase*: The instruction is carried out;
5. *Save phase*: The result of the operation is stored into main memory.

What at first sight might look like making easy things (one instruction) complicated (five steps) is a powerful way to let different instructions execute concurrently.

When an instruction's first step is finished, and its second starts, the next instruction can already begin with its first step and so on. Ideally, up to five instructions can be executed concurrently.

Today's *superpipelined* computers have even more than five stages. Thus, the stages can be made simpler, allowing for a higher clock rate and for a higher computational speed. The number of stages does not affect execution time, except in the start-up phase during the first filling of the pipeline.

Unfortunately, there are instructions like jumps (or even conditional jumps) in the program, which make it difficult to start the following instruction. There are latencies when accessing the slow levels of the memory hierarchy. Interdependent instructions may occur, where the second instruction needs the result of the previous one. In such a case, the execution of the instructions in the pipeline has to be delayed. This phenomenon is called a *pipeline stall*.

It is the task of the compiler to arrange instructions in a way to avoid as many pipeline stalls as possible. On the other hand, the programmer is responsible to design a concrete implementation of an algorithm in such a way that the optimizer can do its job.

This performance factor has another effect: it is quite difficult to establish the effectiveness of a pipeline if the compiler does not provide an explicit output. Fortunately, some modern compilers do not require the algorithm developer to take a deep look into assembler code to find stalls, because they provide so-called *scheduling reports*.

Multiple Execution Units

The *superscalar* approach is based on the fact that there are fundamentally different instructions like integer instructions, floating-point instructions and memory access instructions. The execution unit can therefore be divided into several highly optimized units which can even work concurrently. There are even several units of the same class in most of the modern microprocessors to increase throughput.

Of course, as in a single pipeline, data dependencies and bandwidth issues must be considered that could reduce the benefit of multiple execution units.

Most of the time the assignment of instructions to a unit is done statically at compile time by an optimizing compiler. Such processors are called *very long instruction word processors (VLIW)*. However, there are processors which assign the units at run time without any previous compiler support, so-called *superscalar processors*.

1.2.2 The Memory

The Memory Hierarchy

While processor speed increased significantly and predictably over the last decades, memory access time did not evolve at the same pace. Instead, memory capacity increased faster, basically due to economic considerations, as it is cheaper to achieve fast access times for small memory. To combine the advantages of fast, small memory and slow, large memory, so-called *caches* are introduced. These are fast intermediate memory levels between the processor and the main memory.

Parts of the data stored in main memory are (slowly) transferred to the cache levels where the program can access them rapidly. The idea is, that any memory reference will be followed very likely by a reference to an address near the first one soon afterwards. This concept is called *temporal locality of reference*.

The effective speed gains depend on how great a part of the transferred data can be used before other data has to replace the one previously loaded.

This concept can be applied to instructions as well as to data. Moreover, more than one intermediate cache level can be introduced, resulting, for instance, in primary cache and secondary cache. When analyzing execution times of an algorithm depending on different problem sizes, one can easily notice a performance breakdown both when primary cache misses happen more frequently and later when secondary cache misses start to predominate. It is one goal of an efficient algorithm to minimize the negative impact of those cache effects.

The concept of caches can even be extended to the slowest storage media like magnetic tapes, by using the main memory (which is astonishingly fast and expensive compared to a tape) as a big cache for the tape storage. On the other side, registers are super-fast and super-small memory level compared to the next lower level of the memory hierarchy. Finally the internet with its mirror sites and replicated data servers is the bottom level of a memory hierarchy.

Strictly speaking, any memory hierarchy consists of several memory levels. Each level is of lower capacity but faster accessibility than the level below it. Generally,

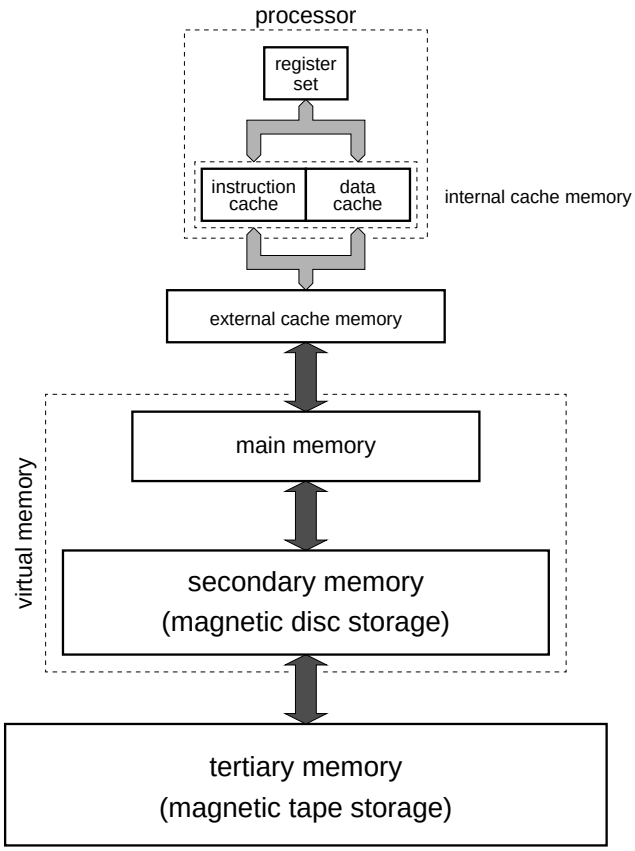

Figure 1.1: A typical memory hierarchy consisting of six levels.

each level contains a full copy of the data stored in the level above it. A machine word is stored in the highest level once it has been referenced. Each word remains in the memory level it is stored in as long as the capacity of the level is not exhausted.

Modern processors try to minimize performance degradation by pre-loading parts of the content of the memory to upper levels, whenever they are likely to be accessed next (for example, in a typical program loop).

Modern computers use several levels of a memory hierarchy, typically six. When writing algorithms for such architectures, one has to keep in mind that an effective cache utilization is crucial to approach peak performance.

There are only few exceptions to such hierarchies. There are (very expensive) examples of computers (NEC SX) which have a main memory with already cache-like characteristics.

Memory Accesses and the Unit Stride Issue

Memories organized in banks are called "interleaved". This is done to raise the performance of memory access. Memory consists typically of 2 or 4 banks, for example the odd and even memory locations.

When performing, for example, a single load instruction, this results in an access to one of the banks and there a certain memory latency occurs. But the next bank can provide the following word immediately afterwards without latency. Of course, the program can only benefit from this feature as long as its access pattern requires data near to each other, it is best if the data immediately follow each other. Therefore algorithms should try to access data in a *unit stride* way, i. e., demanding subsequent memory accesses.

The easiest form of an FFT algorithm consists of two loops. They can be arranged differently, so that unit stride data access may or may not be achieved. This has a big impact on the floating-point performance of the program.

There is one more important point about unit stride. When accessing more-dimensional fields in different languages like C and Fortran, the same code might result in unit stride code in C and in a fatal access pattern in Fortran. This is because arrays are arranged differently by those languages.

Especially with SIMD algorithms, the unit stride issue becomes important. The only efficient way to load data into a SIMD register is to load the complete register from an proper aligned memory address. For the Pentium III and the Motorola G4 AltiVec data can only be loaded efficiently into the SIMD register, if loaded in 16-byte chunks aligned at 16 byte.

Cache Considerations

A *cache miss* happens whenever a requested memory word is not found in the cache but has to be fetched from a lower level of the memory hierarchy. A cache miss is usually the cause of a high latency and therefore entails pipeline stalls.

The cache itself consists of many blocks. When memory is accessed a whole block is transferred from main memory and replaces one of the cache blocks.

When each memory block has only one specific place where it may be loaded into the cache, the cache is said to be *direct mapped*. If the block can be placed anywhere in the cache, the cache is called *fully associative*. This cache type is more demanding on chip logic and could restrict clock rates. Therefore most of the actual caches used are *set associative*. In this case, a block from memory can only be stored in a restricted set of places in the cache. A set is a group of blocks in the cache. If there are n blocks in a set, the cache is said to be *n-way set associative*.

When storing a block in the cache, it replaces another one. Which block should be replaced in a set? There are different approaches, like *random replacement*, but most of the time the most obvious method is used, the *least recently used (LRU)* method, where that block is replaced, which has not been accessed for the longest period of time.

What happens, when writing data to the cache? There are two basic options to handle stores: the first is *write through* (or *store through*), where the information is written both to the cache and to the lower memory levels. The second is called *write back* (*copy back*, *store in*). Here, the data is written only to the cache. The modified cache block is written to main memory only when it is replaced. Write through is easier to implement and read misses do not result in writes to the lower level. Write back stores are performed at the speed of the cache, multiple writes require only one write to the lower level. Also, less memory bandwidth is used, making write back attractive in multi-processor systems. On the other hand, consistency must be preserved.

There are different types of cache misses:

- *Compulsory cache misses (cold start misses, first reference misses)* occur during the first access to a block which is not yet stored in the cache.
- *Capacity misses* occur when the cache is too small to store all the blocks needed during the program execution.
- *Conflict misses* occur in set associative or direct mapped caches when too many blocks are mapped to the same set.

The last two cases must be considered carefully when implementing high performance algorithms. Especially conflict misses can be a critical factor in FFT algorithms, as the transformation lengths and the cache sizes usually are powers of 2. When unit stride access occurs to three memory locations (which is typical for an FFT algorithm with pre-computed weight factors) and they are mapped to the same block set of a 2-way set associative cache, one block has to be replaced at every step. This can easily happen at some step of the execution for data organized at power of 2 memory locations. Note, unit stride access—normally considered to be the best access pattern—leads to a performance degradation.

A concept to minimize cache misses is pre-fetching. This means, that a operand is fetched into a given cache-level before it is used by an instruction. This operation should be completed before the code actually accesses that data. Normally the compiler inserts that instructions on the optimal place. But on difficult data access patterns it is favorable that the programmer can issue that instructions explicitly.

Sometimes data should be loaded and stored without polluting the cache. This is called data streaming. The PentiumIII and the Motorola G4 AltiVec both offer this mechanisms.

The Virtual Memory

The first computers made an explicit distinction between main memory and slower memory levels, like hard disks. Usually, only programs that would fit into main memory along with their data could be executed. Then, hard disks were used to serve as one level of the memory hierarchy. Only the currently executed part of a program and the data needed were loaded from disk to main memory. Suddenly, programs were able to address Gigabytes of memory, even when the main memory was much smaller. This concept is known as *virtual memory*. It has many advantages: first, larger programs can be written and they can handle larger amounts of data even on small computer systems. Software became more independent of the underlying hardware and (without modification) will benefit from larger main memories of future hardware generations.

Nearly every modern operating system provides virtual memory. Thus, for most users the difference between hard disks and main memory vanishes, which made computers a lot easier to understand for the beginner.

But when developing algorithms which handle such amounts of data, that the virtual memory must be used, i. e., data has to be transferred from or to disk, one must be aware that there are two layers of the memory hierarchy involved and that any disk write needs much more time than a main memory access. FFT algorithms sometimes have to deal with such long transformation lengths. When FFT algorithms are not designed to deal with virtual memory issues, performance will collapse completely and the operating system may fall into some sort of coma.

1.2.3 Special Instructions

Floating-Point Operations

Real numbers of mathematics are usually represented in a computer as *floating-point numbers*. Those numbers may use different numbers of bytes, resulting in different precision, the most common being *float, double* and *long double precision*, using 32, 64 and 128 bits respectively.

Many operations applicable to floating-point operands are provided in modern processors. These processors also provide the basic arithmetic operations like addition, subtraction, multiplication, division, as well as a range of more complex functions like inversion, absolute value, power, square root, sine, exponential and logarithmic functions. Depending on their complexity, their implementation in hardware or software, and the values of the arguments, these functions require different numbers of cycles to complete. Moreover, some functions require two or even more operands, resulting in more than one memory access, which might stall the pipeline even before the instruction can be executed.

For FFT algorithms with pre-computed twiddle-factors only addition and multiplication operations are needed. The final division of the FFT output is normally not made by the FFT routine itself. When a re-transformation to the time domain is not required, the division operation may indeed be omitted. Additionally, division is the most expensive arithmetic operation, while multiplication and addition operations usually have a repeat rate of only one cycle.

Most numerical software products require a large number of floating-point operations. Therefore the number of floating-point instructions executed in one second is used to assess both hardware performance and software effectiveness.

Fused Multiply-Add Operations

Modern RISC processors provide a special instruction, a combination of addition and multiplication, the most important arithmetic operations. The *fused multiply-add (FMA)* instruction allows to perform the instruction $a + b \times c$ in one single step (one cycle). On some architectures only one rounding error occurs.

Depending on the type of algorithm, typically up to 20 percent of the floating-point instructions can be combined that way by the compiler.

The FMA instruction has some interesting characteristics. First, it is ternary, and if none of the operands is found in the registers, it requires *three* loads and one store. So the execution of this single instruction may entail several cache misses.

Second, while the repeat rate of an FMA instruction usually is one cycle, the latency can be higher. Therefore it is not advisable to expect a speed-up proportional to the usage of FMA operations.

SIMD Operations

Current desktop computer processors like the Pentium III, the Motorola G4 AltiVec, and the AMD K7 have a special short vector FPU with 2 or 4 single precision elements per vector. Some of the corresponding SIMD operations are like ordinary FPU operations (addition, multiplication, subtraction) others are like FMA instructions.

SIMD extensions are meant to accelerate 3D animation (shading, lighting, ...) and multi media application (IP telephony, Digital TV, ...). But a short vector unit unit can also be used for single precision mathematical algorithms like the FFT. As FFT algorithms cannot be implemented applying unit stride memory access patterns only, the use of SIMD operations in this context turns out to be tricky.

Chapter 2

SIMD Technology

Since multi media applications (like 3D geometry, real-time digital video and audio processing) will be among the most important computing tasks in the next years, major vendors of general purpose microprocessors have added *single instruction, multiple data* (SIMD) extensions to their *instruction set architectures* (ISA) to improve the performance in these applications by exploiting the subword level parallelism available in most multi media kernels.

All SIMD extensions are currently based on the packing of large registers with smaller 8-bit, 16-bit or 32-bit datatypes . Operations are then performed in parallel on the separate datatypes within the register. Although initially the new data types did not include single-precision floating-point numbers, more recently, new instructions have been added to deal with single-precision floating-point SIMD parallelism. For example, Motorola's AltiVec and Intel's SSE can operate on four single-precision floating-point numbers in parallel.

If single-precision is sufficient, SIMD extensions can be utilized to significantly speed-up general scientific numerical computations [24] [25]. However, if the restriction to single-precision floating-point numbers is too strong for a particular scientific application, a recently announced Intel processor (code name "Willamette"), able to operate on double-precision floating-point numbers in parallel, could be used.

The intra-vector parallelism of SIMD extensions form a contrast to the inter-vector parallelism of vector supercomputers such as those manufactured by Cray Research, Inc., Fujitsu or NEC. Vector lengths in such machines reached up to hundreds of elements. For example, the Cray Y-MP vector registers contain 64 elements while the Cray C90 vector registers hold 128 double-precision, floating-point elements.

Microprocessor SIMD extensions have many similarities with vector processors, with some notable differences. The basic similarity is that they are all based on operating in parallel on small data items packed into long data units (vectors). Operations are performed on multiple data elements by one single instruction. This is often referred to as *(short) vector* SIMD parallel processing. This technique also differs from the parallelism achieved through multiple pipelined parallel execution units within superscalar RISC processors in that the programmer explicitly specifies parallel operations using instruction set extensions.

2.1 Introduction to SIMD Extensions

Two classes of processors supporting SIMD extensions can be distinguished: (*i*) processors supporting only integer SIMD instructions, and (*ii*) processors supporting both integer and floating-point SIMD instructions. Processors supporting integer SIMD instructions are covered in the following since those instructions can be used to accelerate the bitreversal permutation of FFT algorithms.

2.1.1 Integer SIMD Extensions

With the PA-7100LC, Hewlett-Packard introduced a small set of multi media acceleration extensions, MAX-1, which performed parallel subword arithmetic. Though the design goal was to support all forms of multi media applications, the most convincing application was real-time MPEG-1, which achieved performance gains using high-level C software which applied specific macros to directly invoke MAX-1 instructions.

Next, Sun introduced VIS, a much larger set of multi media extensions for UltraSparc microprocessors. In addition to the parallel arithmetic instructions, VIS provides novel instructions specifically designed for reducing memory latency for algorithms that manipulate visual data. In addition, it includes a special-purpose instruction to compute the sum of absolute differences of eight pairs of pixels, similar to an instruction found in multi media coprocessors such as Philips' Trimedia.

Then, Hewlett-Packard introduced MAX-2 with its 64-bit PA-RISC 2.0 microprocessors. MAX-2 has some new instructions for subword data alignment and rearrangement to further support subword parallelism. MAX-2 is an interesting attempt to provide a minimalistic set of general-purpose multi media acceleration primitives.

The MMX technology is a set of multi media extensions for the Intel x86 family of processors. It lies between MAX-2 and VIS in terms of both the number and complexity of new instructions. The MMX designers have skillfully integrated a useful set of multi media acceleration instructions within the somewhat constrained register structure of the x86 architecture. MMX shares some characteristics of both MAX-2 and VIS, and also introduces some new instructions like the parallel 16-bit multiply-accumulate instruction.

VIS, MAX-2, and MMX all have the goal to provide high-performance multi media processing on a general-purpose microprocessor. They all provide a full set of subword-parallel instructions operating on 16-bit subwords, and having a parallelism of four subwords per 64-bit register word. Differences exist in the type and amount of support they provide, some of which are driven by the needs of their target market. For example, support is provided for 8-bit subwords when the

target market includes lower end multi media applications (for example, games) in addition to high end multi media applications (for example, workstations used in the field of medical imaging).

2.1.2 Floating-Point SIMD Extensions

Efficient floating-point computation is a must for any FFT algorithm. Thus, speeding up floating-point computation is essential to their overall performance.

The MIPS-3D application specific extension (ASE) is a 3D graphics extension to the MIPS64 architecture. This extension adds new instructions to the existing integer and floating-point instructions (primarily indented for higher performance in 3D geometry processing). The MIPS-3D ASE is designed for 64-bit processors that implement the MIPS64 architecture—a superset of the MIPS V instruction set architecture enhanced for fixed-point and floating-point DSP-type operations. These fixed-point and floating-point instructions are intended to increase the performance of audio, video, 3D graphics, and multi media applications on MIPS64 based processors. The floating-point instructions of the MIPS64 architecture operate on single (32-bit), double (64-bit), or paired-single data types. The paired-single operations provide 2-way SIMD capability by operating on two 32-bit floating-point values packed into a single 64-bit register.

The AltiVec SIMD architecture [26] [27] of Motorola's G4 generation of PowerPC microprocessors, the MPC7400, expands the current PowerPC architecture through the addition of a 128-bit vector execution unit, which operates concurrently with the existing integer and floating-point units. This new execution unit provides for highly parallel operations, allowing for the simultaneous execution of up to 16 (4) arithmetic operations in a single clock cycle for 1 byte integer (single-precision floating-point) data values. AltiVec technology employs a short vector parallel architecture. Depending on data size, vectors lengths are 4, 8, or 16.

In the Pentium III processors streaming SIMD extensions (SSE) [19] [20] [21] Intel added 70 new instructions to the IA-32 architecture. The programming model is similar to the MMX technology model except that instructions now operate on new packed floating-point data types, which contain four single-precision floating-point numbers. The SSEs of the Pentium III processor introduce new general purpose floating-point instructions, which operate on a new set of eight 128-bit SSE registers. In addition to these instructions, SSE technology also provides new instructions to control cacheability of all data types. These include ability to stream data into the processor while minimizing pollution of the caches and the ability to prefetch data before it is actually used. Both 64-bit integer and packed floating-point data can be streamed to memory.

Intel's Willamette Processor[1] is the first member of a new family of processors that are the successors to the Intel P6 family of processors, which includes the Intel Pentium Pro, Pentium II, and Pentium III processors. New SIMD instructions introduced in the Willamette processor architecture include floating-point SIMD instructions, integer SIMD instructions, and conversion of packed data between XMM registers and MMX registers.

The new added floating-point SIMD instructions allow computations to be performed on packed double-precision floating-point values (two double-precision values per 128-bit XMM register). Both the single-precision and double-precision floating-point formats and the instructions that operate on them are 100 % compatible with the IEEE *Standard* 754 *for Binary Floating-Point Arithmetic*.

2.1.3 Software Support

Currently, application developers have three common methods for accessing media-processing hardware within a system. They can invoke vendor-supplied, media-processing libraries; rewrite key portions of the application in assembly language using the media-processing instructions; or code in a high-level language and use vendor-supplied macros that make available the functionality of the media-processing primitives through a simple function-call like interface.

The simplest approach to improving media-processing application performance is to rewrite the system libraries to employ the media-processing hardware. The clear advantage of this approach is that existing applications can immediately take advantage of the new hardware without recompilation. However, the restriction of media-processing hardware to the system libraries also limits potential performance benefits. An application's performance will not improve unless it invokes the appropriate system libraries, and the overheads inherent in the general interfaces associated with system functions will limit application performance improvements. Even so, this is the easiest approach for a system vendor, and vendors have announced or plan to provide media-processing-enhanced libraries.

At the other end of the programming spectrum, an application developer can benefit from media-processing hardware by rewriting key portions of an application in assembly language. Though this approach gives a developer great flexibility, it is generally tedious and error prone. In addition, it does not guarantee a performance win (over code produced by an optimizing compiler), given the complexity of today's microarchitectures.

Recognizing the tedious and difficult nature of assembly coding, most media-processing hardware vendors have developed programming-language abstractions. These give an application developer access to the low-level media-processing prim-

[1] Code name for the next generation of IA-32 processors

itives without having to actually write assembly language code. Typically, this approach results in a function-call-like abstraction that represents one-to-one mapping between a function call and a media-processing instruction.

There are several benefits to this approach. First, the compiler not the developer performs machine-specific optimizations such as register allocation and instruction scheduling. Second, this method integrates media-processing operations directly into the surrounding high-level code without an expensive procedure call to a separate assembly language routine. Third, it provides a degree of transportability and isolation from the specifics of the underlying hardware implementation. If the media-processing primitives do not exist in hardware on the particular target machine, the compiler can replace the media-processing macro with a set of equivalent operations.

The most common language extension for specifying media-processing primitives is to provide function-call like macros within the C programming language. C compilers for the HP MAX-2, Intel's MMX and Streaming SIMD, Motorola's AltiVec, and Sun's VIS architectures support this approach.

Each macro directly translates to a single media-processing instruction, and the compiler allocates registers and schedules instructions. This approach would be even more attractive to application developers if the industry agreed to a common set of macros, rather than having a different set from each vendor.

While macros may be an acceptable and even efficient solution for invoking multi media instructions within a high-level language, subword parallelism could be further exploited with automatic compilation from high-level languages to multi media instructions. Some research directions toward automatic compilation techniques for media-processing architectures are currently under investigation. Pacific Sierra Research Corporation recently announced VAST-C/AltiVec, a C compiler that automatically create vector code for Motorola AltiVec PowerPC microprocessors.

2.1.4 Data Streaming

One of the keys to a fast media-processing application is the efficient streaming of data into and out of the processor. Multi media programs such as video decompression stress the data memory system in ways that the multilevel cache hierarchies of many general-purpose processors cannot handle efficiently. These programs are data intensive with working sets bigger than many first-level caches. Streaming memory systems and compiler optimizations aimed at reducing memory latency (for example, via prefetching) have the potential to improve these applications' performance. Current research in data and computation transformations for parallel machines may provide starting points for further gains in this area.

2.2 Motorola's AltiVec Technology

Motorola's AltiVec technology expands the current PowerPC architecture through the addition of a 128-bit vector execution unit, which operates concurrently with the existing integer and floating-point units. This new engine provides for highly parallel operations, allowing for the simultaneous execution of up to 16 operations in a single clock cycle.

AltiVec technology is a short vector parallel architecture. Depending on data size, vectors are 4, 8 or 16 elements long.

AltiVec technology offers support for

- 16-way parallelism for 8-bit signed and unsigned integers and characters,
- 8-way parallelism for 16-bit signed and unsigned integers, and
- 4-way parallelism for 32-bit signed and unsigned integers and IEEE floating-point numbers.

AltiVec technology also includes a separate register file containing 32-entries, each 128-bits wide. These 128-bit wide registers hold the data sources for the AltiVec technology execution units. The registers are loaded and unloaded through vector store and vector load instructions that transfer the contents of a single 128-bit register to and from memory.

AltiVec technology can be most accurately thought of as a set of registers and execution units added to the PowerPC architecture in an analogous manner to the addition of floating-point units. Floating-point units were added to most mainstream microprocessor architectures several years ago to provide better support for high-precision scientific calculations. AltiVec technology is being added to the PowerPC architecture to dramatically accelerate the next level of performance driven, high-bandwidth communications and computing applications.

Each AltiVec instruction specifies up to three source operands and a single destination operand. All operands are vector registers, with the exception of the load and store instructions and a few instruction types that provide operands from immediate fields within the instruction. 162 new unique instructions are defined for the AltiVec technology.

2.2.1 Software Support

Motorola has defined extensions to the C language unique to AltiVec technology. Use of Motorola's extensions provides high performance and allows portability between PowerPC compilers.

Pacific Sierra Research Corporation recently announced VAST-C/AltiVec, a C compiler that automatically create vector code for Motorola AltiVec chips. VAST-C/AltiVec is a vectorizing preprocessor that automatically replaces loops in C programs with inline AltiVec vector extensions.

VAST-C/AltiVec features:

- Optimization of entire loop nests, not just inner loops.
- Loop fusion, outer loop unrolling, loop collapse, loop interchange.
- Unrolled vector loops.
- Vectorization of reduction loops.
- Vectorization of conditional loops.
- Non-aligned vectors can be vectorized.
- Loops with non-unit stride vectors can be vectorized.
- 32-bit float and 8, 16 and 32-bit integer vectorization.
- ALIGNED pragma so that the user can inform VAST-C about arrays that are aligned on 16-byte boundaries.
- DISJOINT, NODEPCHK pragmas for disambiguating data dependencies.
- -L parameter for assertion levels to allow vectorization in the presence of pointer arguments.

GNU has released a version of their development tools (including `gcc`), that supports Motorola's AltiVec extensions. This Package runs on Linux G4 Systems and is also available as a cross compiler for SunOS and the Windows "Linux portation" `cygwin`.

2.2.2 Used Features

All experiments on the Motorola G4 AltiVec were carried out using the PPC Linux GNU `gcc-vec` AltiVec enabled compiler. The following operations were used within the AltiVec FFTW SIMD version.

The Data Types `vector float`, `vector unsigned char`.

The `vector float` type refers to a 128 bit quantity that is 16 byte aligned. It is a 4-element vector of `float` elements denoted `[z, y, x, w]`. Variables of type `vector float` should be declared, for instance as

```
vector float var = (vector float) (1.0, 2.0, 3.0, 4.0)
```

The `vector unsigned char` type refers to a 128 bit quantity that is 16 byte aligned. It is a vector of 16 `unsigned char` elements. and used for addressing subelements in the `vect_perm` instruction.

The Arithmetic Operations `vec_add`, `vec_sub`, `vec_madd`, `vec_nmsub`.

These intrinsics operate on 2 or 3 parameters of type `vector float` and return the result of type `vector float` denoted as `[r3, r2, r1, r0]`.

```
vector float vec_add(vector float a, vector float b)
r0:=a0+b0
r1:=a1+b1
r2:=a2+b2
r3:=a3+b3

vector float vec_sub(vector float a, vector float b)
r0:=a0-b0
r1:=a1-b1
r2:=a2-b2
r3:=a3-b3

vector float vec_madd(vector float a, vector float b,
                      vector float c)
r0:=a0*b0+c0
r1:=a1*b1+c1
r2:=a2*b2+c2
r3:=a3*b3+c3

vector float vec_nmsub(vector float a, vector float b,
                       vector float c)
r0:=-(a0*b0-c0)
r1:=-(a1*b1-c1)
r2:=-(a2*b2-c2)
r3:=-(a3*b3-c3)
```

The Vector Reordering Operations `vec_perm`, `vec_mergel`, `vec_mergeh`, `vec_splat`.

These intrinsics recombine elements from their 1 or 2 arguments of type `vector float` into one result of type `vector float`.

```
vector float vec_perm(vector float a,vector float b,
                      vector unsigned char i)
do i=0 to 15
  j:=c{i}[4:7]
```

```
  if c{i}[3]=0 then
    r{i}:=a{j}
  else
    r{i}:=b{j}
  end if
end do

vector float vec_splat(vector float a, int b);
r0:=a[b]
r1:=a[b]
r2:=a[b]
r3:=a[b]

vector float vec_mergel(vector float a, vector float b)
r0:=a2
r1:=b2
r2:=a3
r3:=b3

vector float vec_mergeh(vector float a, vector float b)
r0:=a0
r1:=b0
r2:=a1
r3:=b1
```

The Memory Access Functions `vec_ld`, `vec_ste`.

The function `vec_ld` loads a `vector float` from an 16 byte aligned address. `vec_ste` stores one element of a `vector float` at an 4 byte aligned address. The index of the stored element is chosen according to the alignment of the address.

```
vector float vec_ld(int a, float *b)
p:=AND(a+b,0xFFFFFFFFFFFFFFF0)
r0:=*p0
r1:=*p1
r2:=*p2
r3:=*p3

void vec_ste(vector float a, int b, float *c)
i:=MOD(b+c,0x10)
*c:=a[i]
```

The Alignment Support Functions `vec_lvsl`, `vec_lvsr`.

These intrinsics return a `vector unsigned char` that can be used with the `vec_perm` to load and store missaligned data. Here these intrinsics are used to swap the upper and the lower two elements of a `vector float` if needed.

```
vector unsigned char vec_lvsl(int a, float *b)
EA:=(a+b)
sh:= EA[28..31]
if sh = 0x0 r:=0x000102030405060708090A0B0C0D0E0F
if sh = 0x1 r:=0x0102030405060708090A0B0C0D0E0F10
if sh = 0x2 r:=0x020304050607080 90A0B0C0D0E0F1011
...
if sh = 0xF r:=0x0F101112131415161718191A1B1C1D1E

vector unsigned char vec_lvsr(int a, float *b)
EA:=(a+b)
sh:= EA[28..31]
if sh = 0x0 r:=0x101112131415161718191A1B1C1D1E1F
if sh = 0x1 r:=0x0F101112131415161718191A1B1C1D1E
if sh = 0x2 r:=0x0E0F101112131415161718191A1B1C1D
...
if sh = 0xF r:=0x0102030405060708090A0B0C0D0E0F10
```

2.3 The MIPS-3D ASE Extensions

The MIPS-2D ASE extensions to the MIPS V instruction set introduces a new datatype called the paired single. The paired single is the packing of two 32-bit single-precision floating-point values into one 64-bit register. Thus creating a new set of two element vector operations. The paired single datatypes are intended to be used in computer graphics to perform 3-D transforms, shading, clipping and lighting calculations. The paired single operations can also be used to accelerate signal processing applications where fixed-point arithmetic is inadequate.

The MIPS-3D ASE extensions add four and eight element vector capability to integer arithmetic. The MIPS-3D ASE registers share the 32 existing 64-bit floating-point registers and are loaded with existing floating-point load and store double-word operations. In each MIPS-3D ASE instruction is a format specifier. In such a format specification, the registers are interpreted as *quad half* (QH) and *oct byte* (OB). In QH the 64-bit FPR is interpreted as a vector of 4 signed 16-bit integers, and in the OB format it is interpreted as a vector of 8 unsigned 8-bit integers. The rest of the MIPS-3D ASE architecture is centered around a vector accumulator.

The MIPS-3D ASE has a private 192-bit accumulator register. The format of the accumulator is determined by the format of the elements accumulated. In case of the OB, the accumulator is formatted as 8 unsigned 24-bit slices, in the case of QH, it is formatted as 4 48-bit slices. As the product of two n-bit integers can grow to $2n$ bits, the use of an accumulator that is $3n$ bits wide permits the additional additive accumulation of $2n$ $n \times n$ multiplies with a single shift or rounding operation at the end of the accumulation cycle. Because the 192-bit

accumulator is used, MIPS-3D ASE does not have 32-bit integer vector datatypes.

Finally, the MIPS-3D ASE extensions include a method that allows one to select the second operand of a vector operation to be either a vector or a scalar element of the second vector register or an immediate scalar value.

2.4 The Intel Pentium III Processor

The Intel Pentium III processor is Intel's first processor that offers the *streaming SIMD extensions* (SSE). SSE instructions are Intel's floating-point and integer single instruction multiple data extensions to the P6 core. That means this instructions can operate on a short vector (of size 4) of single precision floating-point numbers. Each cycle one SIMD instruction (e. g., vector addition, vector shuffle operation, find maximum, comparison instruction, ...) can be issued. So this processor is qualified for short vector algorithms.

The Intel Pentium III integrates the following features:

- P6 dynamic execution microarchitecture.
- Dynamic branch prediction.
- Out of order execution.
- Intel MMX media enhancement technology.
- Streaming SIMD extensions.
- Program monitor counters (PMC) for performance analysis.
- Available at clock rates ranging from 450 MHz to 1.13 GHz.
- 100MHz and 133 MHz frontsidebus versions.
- 16 KB L1 data cache and 16 KB L1 instruction cache (non-blocking).

The Pentium III comes in two versions, Katami (0.25 micron, 512 KB off-die half speed cache) and Coppermine (0.18 micron, 256 KB on-die full speed cache).

2.4.1 The Intel Architecture (IA)

The Intel Pentium III processor is a member of the P6 family. It has a three-way superscalar, pipelined architecture. Thus, the Pentium III is able to decode, dispatch and complete execution of three instructions per clock cycle. The P6

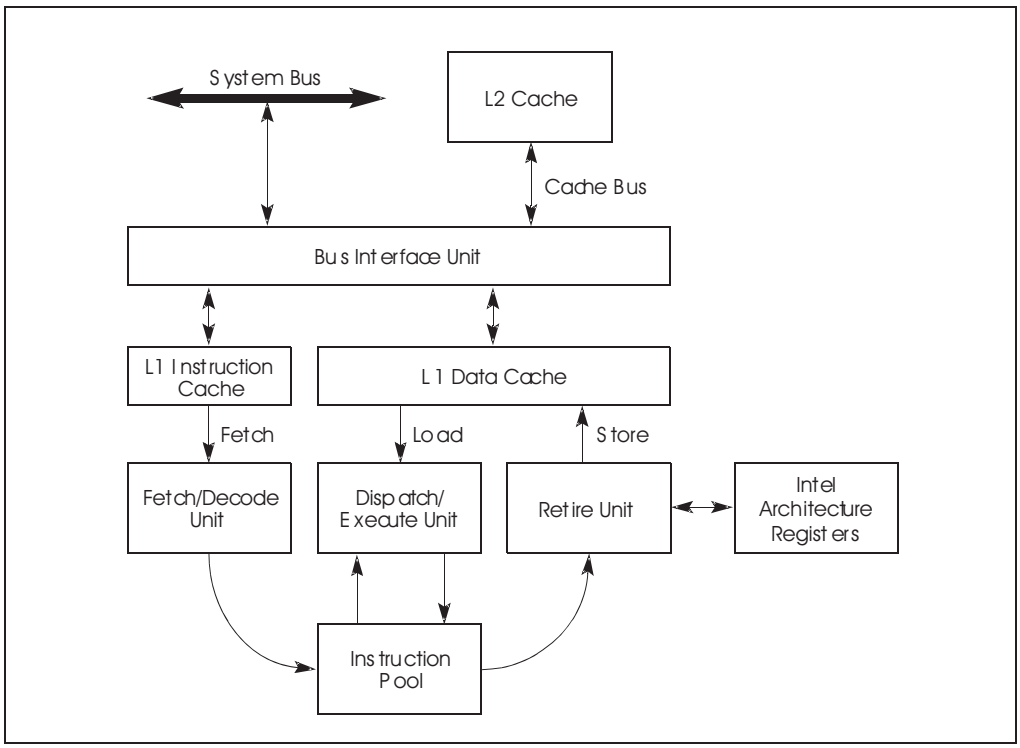

Figure 2.1: The processing units in the P6 microarchitecture.

family uses a 12-stage superpipeline that supports out-of-order instruction execution. Figure 2.1 shows a conceptual view of this pipeline, with the pipeline divided into four processing units (the fetch/decode unit, the dispatch/execute unit, the retire unit, and the instruction pool). Instructions and data are supplied to these units through the bus interface unit. The centerpiece of the P6 processors' microarchitecture is the out-of-order "dynamic execution" mechanism.

Dynamic execution incorporates three concepts: (*i*) deep branch prediction, (*ii*) dynamic data flow analysis, and (*iii*) speculative execution.

The Floating-Point Unit

The processors of the P6 family include a floating-point unit (FPU). In the FPU the registers are treated as as stack of eight 80 bit values (ST(0) ... ST(7)). The calculations are performed like

```
FADD ST,ST(2)
```

Here the value in the stack register 2 above the actual top is added to the top of the stack. The FPU provides the floating-point data types

- Single precision (24-Bits),

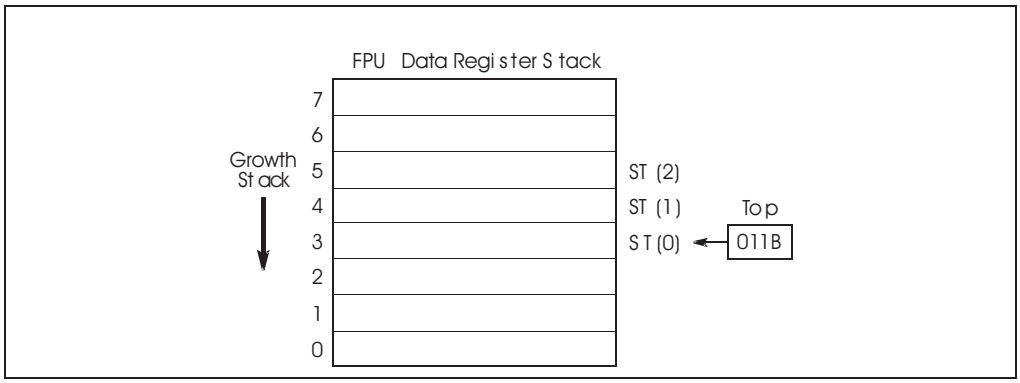

Figure 2.2: The FPU data register stack

- Double precision (53-Bits),
- Extended precision (64-Bits),

and the usual integer types. Figure 2.2 shows the FPU data register stack.

The MMX Technology

The MMX technology uses the single instruction, multiple data (SIMD) paradigm to perform arithmetic and logical operations on the bytes, words, or doublewords packed into 64-bit MMX registers. For example, the PADDSB instruction adds 8 signed bytes from the source operand to 8 signed bytes in the destination operand and stores 8 byte-results in the destination operand.

Values in MMX registers have the same format as a 64-bit quantity in memory. MMX registers have a 64-bit access mode and a 32-bit access mode.

The 64-bit access mode is used for 64-bit memory access, 64-bit transfer between MMX registers, all pack, logical and arithmetic instructions, and some unpack instructions.

The 32-bit access mode is used for 32-bit memory access, 32-bit transfer between integer registers and MMX registers, and some unpack instructions.

The MMX instruction set consists of 57 instructions, grouped into the following categories: (*i*) data transfer instructions, (*ii*) arithmetic instructions, (*iii*) comparison instructions, (*iv*) conversion instructions, (*v*) logical instructions, (*vi*) shift instructions, (*vii*) empty MMX state instruction (EMMS).

The MMX instructions use the same eight registers for the computation as the FPU. So after using MMX instructions the state must been emptied by the (computational expensive) EMMS instruction.

The Streaming SIMD Extensions

The streaming SIMD extensions (SSE) offer general purpose floating-point instructions, that operate on a set of eight 128-bit SIMD floating-point registers. Each register is a vector of four single-precision floating-point numbers. The streaming SIMD extensions registers are not aliased onto the floating-point registers as are the MMX registers. This set enables the programmer to develop algorithms that can intermix packed, single-precision, floating-point and integer using both streaming SIMD extensions and MMX instructions respectively. In addition to these instructions, streaming SIMD extensions also provide new instructions to control cacheability of all MMX technology and 32-bit IA data types. These instructions include the ability to stream data to memory without polluting the caches, and the ability to prefetch data before it is actually used. The streaming SIMD extensions provide the following extensions to the IA programming environment:

- Eight SIMD floating-point registers (XMM0 through XMM7).
- SIMD floating-point data type: 128-bit, packed floating-point.
- The streaming SIMD extension (SSE) set.

The streaming SIMD extensions consist of 70 instructions, grouped into the following categories:

- Data movement instructions,
- Arithmetic instructions,
- Comparison instructions,
- Conversion instructions,
- Logical instructions,
- Additional SIMD integer instructions,
- Shuffle instructions,
- Management instructions, and
- Ability control instructions.

SSE instructions operate on either all (see Figure 2.3) or the least significant (see Figure 2.4) pairs of packed data operands in parallel. In general, the address of a memory operand has to be aligned on a 16-byte boundary for all instructions.

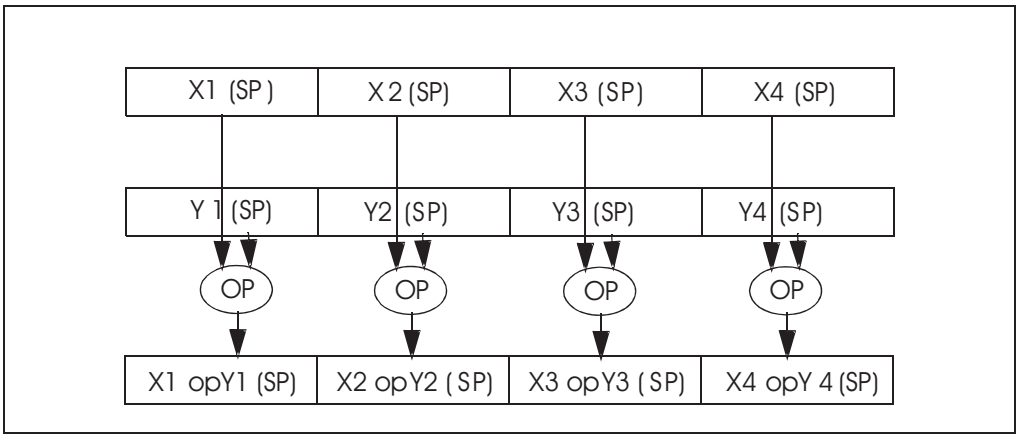

Figure 2.3: Packed SSE operations.

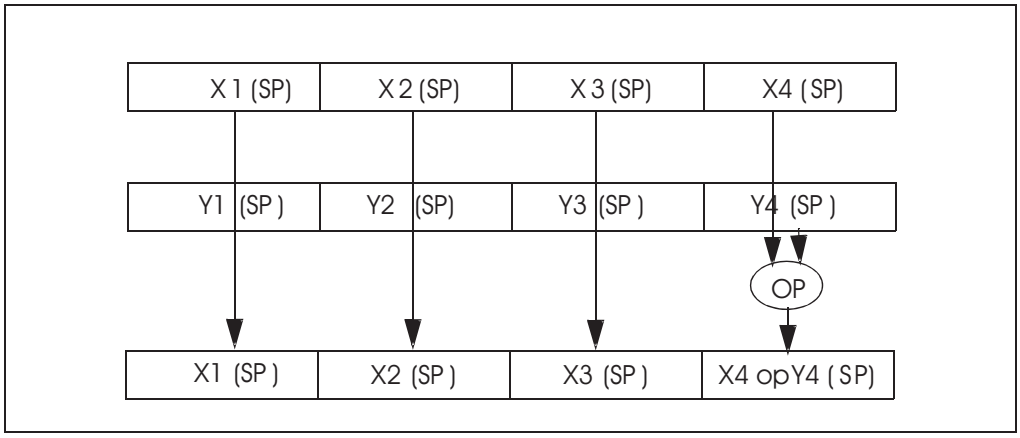

Figure 2.4: Scalar SSE operations.

The data movement instructions include pack/unpack instructions and data shuffle instructions that enable to "mix" the indices in the vector operations. The SHUFPS (shuffle packed, single-precision, floating-point) instruction is able to shuffle any of the packed four single-precision, floating-point numbers from one source operand to the lower two destination fields; the upper two destination fields are generated from a shuffle of any of the four single-precision floating-point numbers from the second source operand (Figure 2.5). By using the same register for both sources, SHUFPS can return any combination of the four single-precision floating-point numbers from this register.

Data referenced by a programmer can have temporal locality (data will be used again) or spatial locality (data will be in adjacent locations). Thus, the programmer does not want the applications cached code and data to be overwritten by this non-temporal data. The cacheability control instructions enable the programmer to control caching so that non-temporal accesses will minimize cache pollution. In addition, the execution engine needs to be fed such that it does not become

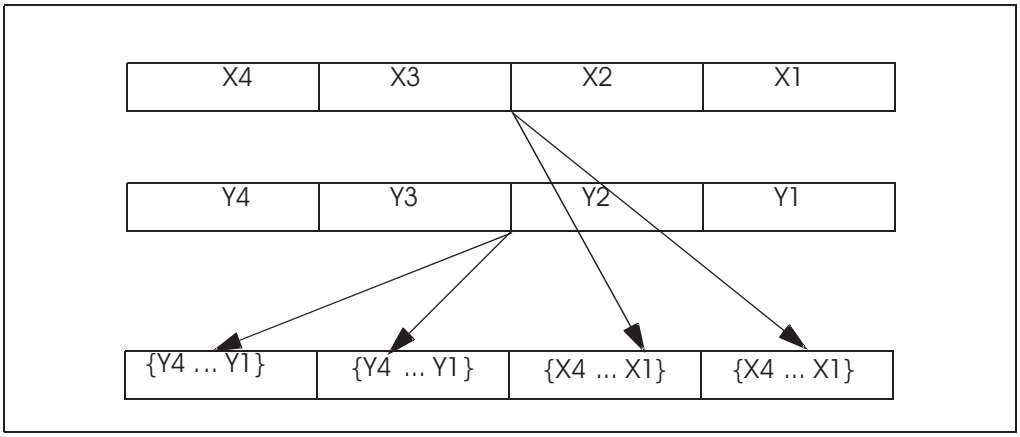

Figure 2.5: Packed shuffle SSE operations.

stalled waiting for data.

The streaming SIMD extensions allow the programmer to prefetch data long before its final use. These instructions are not architectural since they do not update any architectural state and are specific to each implementation. These instructions merely provide a hint to the hardware, and they will not generate exceptions or faults. Excessive use of prefetch instructions may degrade processor performance due to resource allocation.

2.4.2 Development Tools

The usual compiler support for new processor technologies is often insufficient. No commercial compiler supports the SSE instructions except the Intel C/C++ compiler. Intel offers this compiler as a snap-in into Microsofts Visual Studio.

The Microsoft Visual Studio

The Microsoft Visual Studio is a Win32 (Windows 95/98/2000 or WindowsNT) platform product. It is a Windows-based integrated development environment (IDE). That means, all steps needed for building an application can be done in it. It includes an multi-document editor, source code management, an integrated debugger, and support for arbitrary external compiler and other tools. It is therefore an optimal environment for that platform.

The Intel C/C++ Compiler

The Intel C/C++ Compiler [18] is a snap-in into the Microsoft Visual Studio. It uses the Visual Studio's include files, libraries, tools and the Microsoft macro

assembler (MASM). The compiler can either produce executables, object files or assembler language output.

Support for MMX and SSE Extension. The Intel C/C++ Compiler supports the MMX Technology and the streaming SIMD extensions via four different approaches:

- Compiler intrinsics (macros for C),
- Short vector classes (for C++),
- Inline assembler, and
- Automatic vectorization.

For the MMX technology a special `__m64` data type is used to represent a MMX register. The following MMX technology intrinsic groups are supported:

- General support intrinsics,
- Packed arithmetic intrinsics,
- Shift intrinsics,
- Logical intrinsics, and
- Compare intrinsics.

The streaming SIMD extensions are supported via

- The intrinsics API which maps the new instructions to C macros,
- The `__m128` data type, that represents a SSE register (a vector of 4 floats),
- Data alignment support [23] (essentially for SSE), and
- Assembly language support.

Types of Optimization. The Intel C/C++ Compiler has the following optimization capabilities [22]:

- Standard optimization: -O1, -O2 using loop unrolling, global register allocation, constant propagation, ... ,
- Processor specific optimizations (e.g. code especially for the P6 family processors),

- Automatic processor dispatch support (the best code for the actual processor is taken at runtime),
- Automatic prefetching (only for the Pentium III),
- Floating-point optimizations (e.g., division replaced by multiplication by the reciprocal),
- Interprocedural optimization,
- Profile guided optimization, and
- Automatic vectorization.

The VTune Performance Analyzer

The VTune performance analyzer is a tool that provides a graphical view of the performance of an application in a number of different ways, and it gives feedback on tuning the application.

In the following a brief description of each of these methods is given.

Event Based Sampling. Event-based sampling is the most commonly used method for analyzing application performance with the VTune analyzer. It allows the user to select any of a number of different events implemented in the processor. These events allow the user to hone in on specific aspects of an applications use of the processor from clocktick events or time, to specific types of operations that retired, to penalties that occur.

This information is displayed for each module or each line of code. So the user has a very powerful tool to performance evaluate his program. For SIMD code, e.g., a clocktick counter and a flop-counter can be used and cache misses can be recorded to see in which part of the algorithm the memory performance should be improved.

Code Coach. The analyzer's *code coach* analyzes the source code using some information from the compiler and offers advice on ways to tune the application. The advice ranges from tips on restructuring search algorithms to advice on using the intrinsics or the performance library suite.

Dynamic Analysis. Dynamic analysis uses the same software simulator the Pentium II and Pentium III processor architects used in designing the processors. It is useful for honing in on specific micro-architectural information for hotspots identified by event-based sampling such as penalties and retirement time. It is especially useful for analyzing branch mispredictions and cache utilization.

2.4.3 Used Features

All algorithms discussed here were developed in the VisualStudio98/Intel C/C++ compiler environment. The Intrinsic-API of the Intel C/C++ Compiler has been used to code all SIMD algorithms. The following features were used:

The Data Types `__m64`, `__m128`.

`__m64` is a 64 bit quantity that is 8 byte aligned. It is used as a pair of `float` to access complex numbers.

`__m128` is a 128 bit quantity that is 16 byte aligned. It is a vector of 4 `float` elements. It will be denoted as `[z, y, x, w]`. Variables of type `float[4]` should be declared, for instance, as

```
__declspec(align(16)) float[4] var={1.0, 2.0, 3.0, 4.0};
__m128 *pvar=&var;
```

so that they can be accessed as a `__m128` variable.

The Arithmetic Operations `_mm_add_ps`, `_mm_sub_ps`, `_mm_mul_ps`.

These intrinsics operate on 2 parameter of type `__m128` and return the result of type `__m128`. The result is denoted as `[r3, r2, r1, r0]`.

```
__m128 _mm_add_ps(__m128 a, __m128 b)
r0:=a0+b0
r1:=a1+b1
r2:=a2+b2
r3:=a3+b3

__m128 _mm_sub_ps(__m128 a, __m128 b)
r0:=a0-b0
r1:=a1-b1
r2:=a2-b2
r3:=a3-b3

__m128 _mm_mul_ps(__m128 a, __m128 b)
r0:=a0*b0
r1:=a1*b1
r2:=a2*b2
r3:=a3*b3
```

The Vector Reordering Operations `_mm_shuffle_ps`, `_mm_unpacklo_ps`, `_mm_unpackhi_ps`.

These intrinsics recombine elements from their two arguments of type `__m128` into one result of type `__m128`.

```
__m128 _mm_shuffle_ps(__m128 a, __m128 b, int i)
r0:=a[i AND 3]
r1:=a[(i>>2) AND 3]
r2:=b[(i>>4) AND 3]
r3:=b[(i>>6) AND 3]

__m128 _mm_unpacklo_ps(__m128 a, __m128 b)
r0:=a0
r1:=b0
r2:=a1
r3:=b1

__m128 _mm_unpackhi_ps(__m128 a, __m128 b)
r0:=a2
r1:=b2
r2:=a3
r3:=b3
```

The Memory Access Functions `_mm_loadl_pi`, `_mm_loadh_pi`, `_mm_storel_pi`, `_mm_storeh_pi`.

These intrinsics provide the ability to load and store the upper or lower half a register to a `__m64` memory location.

```
__m128 _mm_loadl_pi(__m128 a, __m64 *p)
r0:=*p0
r1:=*p1
r2:=a2
r3:=a3

__m128 _mm_loadh_pi(__m128 a, __m64 *p)
r0:=a0
r1:=a1
r2:=*p0
r3:=*p1

void _mm_storel_pi(__m64 *p, __m128 a)
*p0=a0
*p1=a1

void _mm_storeh_pi(__m64 *p, __m128 a)
*p0=a2
*p1=a3
```

2.5 Motorola's AltiVec Extensions vs. Intel's Streaming SIMD Extensions

2.5.1 Register Usage

Intel's SSE architecture only supports 8×128 bit registers (as compared to AltiVec's 32×128 bit registers). 8 registers are very few for modern compilers and algorithms which can use more registers to optimize algorithms. Algorithms that require more than 8 registers with the Intel x86 architecture require to unload and reload registers more, which wastes time—time that the processor could do more useful things. Additionally, each load/store may cause a cache stall that costs dozens of cycles (or more) waiting for memory access.

Intel's SSE instructions use the same 2 source registers and 1 destination register format as the x86 architecture, but the destination register is one of the source registers—thereby destroying the source-data. Destroyed source-data means that data have to be either reload or a copy of the original data have to be made before an SSE instruction is executed (both of which waste time).

AltiVec registers have a 2 source, 1 filter/modifier and 1 destination register format (and the destination is separate from the source). The $3 \rightarrow 1$ format of AltiVec makes the instruction set more versatile and doesn't cause source-data destruction (and reloads). (Intel can not use the $3 \rightarrow 1$ instruction format because there aren't enough registers).

2.5.2 Floating-Point Instructions

Motorola's AltiVec technology support multiply-add instructions. Intel's SSE extensions doesn't have multiply-add instructions (partly because it is not able to do the $3 \rightarrow 1$ instruction encoding), so with Intel's SSE extension a multiply followed by an add instruction have to be carried out. This two step execution may require an extra register (which the processor is already starved for), and it may force another load/store operation.

Intel's Pentium III processor (the first Intel Processor that implements SSE extensions) can't multiply all 4 vector components (an entire register) at once (nor add them)—it can only do 2 vector components (half the register) at once. Then there is large latency in Intel's SSE extensions to do multiplies (5 cycles), and there is a long latency for add operations as well (3 cycles).

Chapter 3

SIMD FFT Algorithms

3.1 Radix-4 Triple Loop SIMD Algorithms

These algorithms are vectorized variants of the inplace Radix-4 Van Loan algorithm (Algorithm 3.1). The vectorlength is 4. So either 4 real or imaginary parts or 2 complex numbers can be processed simultaneously. The data vector of complex numbers is stored either as a vector of the real parts and a vector of the imaginary parts or as vector of complex numbers. Vectorization in this context means, that more than one pass of a loop is computed simultaneously.

The target loop for vectorization is chosen, that whenever possible, unit stride access is achieved. There are three different vectorization methods required. One for the first stage, one for the last stage and one for the intermediate stages. The twiddle factors are stored in memory so that they can be loaded with unit stride, too.

Algorithm 3.1 ($x := F_n x$, $n = 4^t$)

do $q = 1 : t$
  $L := 4^q$
  $r := n/L$
  $L_* := L/4$
  do $j = 0 : L_* - 1$
    $\omega := e^{-2\pi ij/L}$
    do $k = 0 : r - 1$
      $\alpha := x(kL + j)$
      $\beta := \omega \cdot x(kL + L_* + j)$
      $\gamma := \omega^2 \cdot x(kL + 2L_* + j)$
      $\delta := \omega^3 \cdot x(kL + 3L_* + j)$
      $\tau_0 := \alpha + \gamma$
      $\tau_1 := \alpha - \gamma$
      $\tau_2 := \beta + \delta$
      $\tau_3 := \beta - \delta$
      $x(kL + j) := \tau_0 + \tau_2$
      $x(kL + L_* + j) := \tau_1 - i\tau_3$
      $x(kL + 2L_* + j) := \tau_0 - \tau_2$
      $x(kL + 3L_* + j) := \tau_1 + i\tau_3$
    end do

end do
end do

In stage $q = 1$, the j-loop degenerates. That means, the only loop, that can be vectorized is the k-loop. The twiddle factors are all trivial ($\omega = 1$). Within the k-loop the elements are accessed at unit stride. This results in Algorithm 3.2

Algorithm 3.2 (stage $q = 1$)

$r := n/4$
do $k = 0 : r - 1$
$\alpha := x(4k)$
$\beta := x(4k + 1)$
$\gamma := x(4k + 2)$
$\delta := x(4k + 3)$
$\tau_0 := \alpha + \gamma$
$\tau_1 := \alpha - \gamma$
$\tau_2 := \beta + \delta$
$\tau_3 := \beta - \delta$
$x(4k) := \tau_0 + \tau_2$
$x(4k + 1) := \tau_1 - i\tau_3$
$x(4k + 2) := \tau_0 - \tau_2$
$x(4k + 3) := \tau_1 + i\tau_3$
end do

In stages $q = 2$ to $q = t - 1$, L_* is a multiple of 4 and therefore the j-loop can be vectorized. This results in unit stride access in the k-loop and therefore the vectorization is straight forward. Algorithm 3.3 describes this subproblem.

Algorithm 3.3 (Stage $q = 2$ to $q = t - 1$)

do $q = 2 : t - 1$
$L := 4^q$
$r := n/L$
$L_* := L/4$
do $j = 0 : L_* - 1$
$\omega := e^{-2\pi ij/L}$
do $k = 0 : r - 1$
$\alpha := x(kL + j)$
$\beta := \omega \cdot x(kL + L_* + j)$
$\gamma := \omega^2 \cdot x(kL + 2L_* + j)$
$\delta := \omega^3 \cdot x(kL + 3L_* + j)$
$\tau_0 := \alpha + \gamma$
$\tau_1 := \alpha - \gamma$
$\tau_2 := \beta + \delta$

$\tau_3 := \beta - \delta$
$x(kL + j) := \tau_0 + \tau_2$
$x(kL + L_* + j) := \tau_1 - i\tau_3$
$x(kL + 2L_* + j) := \tau_0 - \tau_2$
$x(kL + 3L_* + j) := \tau_1 + i\tau_3$
end do
end do
end do

In stage $q = t$ the k-loop degenerates and the vectorization works as for stage $q = 2$ to $q = t - 1$. This results in Algorithm 3.4.

Algorithm 3.4 (Stage $q = t$)
$L_* := n/4$
do $j = 0 : L_* - 1$
$\omega := e^{-2\pi ij/n}$
$\alpha := x(j)$
$\beta := \omega \cdot x(L_* + j)$
$\gamma := \omega^2 \cdot x(2L_* + j)$
$\delta := \omega^3 \cdot x(3L_* + j)$
$\tau_0 := \alpha + \gamma$
$\tau_1 := \alpha - \gamma$
$\tau_2 := \beta + \delta$
$\tau_3 := \beta - \delta$
$x(kL + j) := \tau_0 + \tau_2$
$x(L_* + j) := \tau_1 - i\tau_3$
$x(2L_* + j) := \tau_0 - \tau_2$
$x(3L_* + j) := \tau_1 + i\tau_3$
end do

3.1.1 The Interleaved Complex Algorithm

In this variant, all complex data is stored in the *interleaved complex* format. That means a vector of complex numbers is stored as a vector of dupels consisting of the real part and of the imaginary part. This format is natural for applications. FFTW works with this format.

In memory the input vector x is stored that way:

$$x = ((x_{0,r}, x_{0,i}), (x_{1,r}, x_{1,i}), (x_{2,r}, x_{2,i}), \ldots, (x_{n-1,r}, x_{n-1,i}))$$

In a 4-element vector 2 complex numbers can be stored. So in the k-loop 2 oper-

ations are computed simultaneously. The instructions

$$\begin{aligned}\alpha &:= 1 \cdot x(kL+j)\\ \beta &:= \omega \cdot x(kL+L_*+j)\\ \gamma &:= \omega^2 \cdot x(kL+2L_*+j)\\ \delta &:= \omega^3 \cdot x(kL+3L_*+j)\end{aligned}$$

translate into 4 complex multiplications.

If 4 complex multiplications have to be conducted, 8 complex numbers have to be loaded into 4 registers. After loading (this involves some data rearrangement in stages $q = 2$ to $q = t$) data resides in 6 register that way:

$$\begin{array}{lcll} x_{01} & = & (a_0, b_0, a_1, b_1) & = & (x_{0,r}, x_{0,i}, x_{1,r}, x_{1,r}) \\ x_{23} & = & (a_2, b_2, a_3, b_3) & = & (x_{2,r}, x_{2,i}, x_{3,r}, x_{3,r}) \\ \omega_0 & = & (c_0, d_0, c_1, d_1) & = & (1, 1, \omega_r, \omega_i) \\ \omega_1 & = & (c_2, d_2, c_3, d_3) & = & (\omega_r^2, \omega_i^2, \omega_r^3, \omega_i^3) \\ \omega_{0,r} & = & (d_0, c_0, d_1, c_1) & = & (1, 1, \omega_i, \omega_r) \\ \omega_{1,r} & = & (d_2, c_2, d_3, c_3) & = & (\omega_i^2, \omega_r^2, \omega_i^3, \omega_r^3) \end{array}$$

The complex multiplications to be computed are:

$$x_i \cdot \omega_k = (a_i + ib_i) \cdot (c_k + id_k) = (a_i c_i - b_i d_i) + i(a_i d_i + b_i c_i)$$

Now 4 vector-multiplications can be computed:

$$\begin{array}{lcll} tmp_0 & = & x_{01} \cdot_{vec} \omega_{0,r} & = & (a_0 d_0, b_0 c_0, a_1 d_1, b_1 c_1) \\ tmp_1 & = & x_{23} \cdot_{vec} \omega_{0,r} & = & (a_2 d_2, b_2 c_2, a_3 d_3, b_3 c_3) \\ tmp_2 & = & x_{01} \cdot_{vec} \omega_0 & = & (a_0 c_0, b_0 d_0, a_1 c_1, b_1 d_1) \\ tmp_3 & = & x_{23} \cdot_{vec} \omega_0 & = & (a_2 c_2, b_2 d_2, a_3 c_3, b_3 d_3) \end{array}$$

Via data rearrangement the corresponding values have to be grouped together before the final addition and subtraction:

$$\begin{array}{lcl} tmp_0 & = & (a_0 c_0, a_1 c_1, a_2 c_2, a_3 c_3) \\ tmp_1 & = & (b_0 d_0, b_1 d_1, b_2 d_2, b_3 d_3) \\ tmp_2 & = & (a_0 d_0, a_1 d_1, a_2 d_2, a_3 d_3) \\ tmp_3 & = & (b_0 c_0, b_1 c_1, b_2 c_2, b_3 c_3) \end{array}$$

Now the vector of real the parts and the vector of the imaginary can be computed by a vector-addition and a vector-subtraction:

$$\begin{array}{lcl} real & = & tmp_0 -_{vec} tmp_1 \\ & = & (a_0 c_0 - b_0 d_0, a_1 c_1 - b_1 d_1, a_2 c_2 - b_2 d_2, a_3 c_3 - b_3 d_3) \\ imag & = & tmp_2 -_{vec} tmp_3 \\ & = & (a_0 d_0 + b_0 c_0, a_1 d_1 + b_1 c_1, a_2 d_2 + b_2 c_2, a_3 d_3 + b_3 c_3) \end{array}$$

In the context of Algorithm 3.1 this is the intermediate result

$$\begin{aligned} real &= (\alpha_r, \beta_r, \gamma_r, \delta_r) \\ imag &= (\alpha_i, \beta_i, \gamma_i, \delta_i) \end{aligned}$$

The next step in the computation is

$$\begin{aligned} \tau_0 &:= \alpha + \gamma \\ \tau_1 &:= \alpha - \gamma \\ \tau_2 &:= \beta + \delta \\ \tau_3 &:= \beta - \delta \end{aligned}$$

Again, data rearrangement is needed first:

$$\begin{aligned} tmp_0 &= (\alpha_r, \alpha_i, \beta_r, \beta_i) \\ tmp_1 &= (\gamma_r, \gamma_i, \delta_r, \delta_i) \end{aligned}$$

Then τ_0, τ_1, τ_2 and τ_3 can be computed with a vector-addition an a vector-subtraction:

$$\begin{aligned} tmp_0 +_{vec} tmp_1 &= (\alpha_r + \gamma_r, \alpha_i + \gamma_i, \beta_r + \delta_r, \beta_i + \delta_i) &= (\tau_{0,r}, \tau_{0,i}\tau_{2,r}\tau_{2,r}) \\ tmp_1 -_{vec} tmp_1 &= (\alpha_r - \gamma_r, \alpha_i - \gamma_i, \beta_r - \delta_r, \beta_i - \delta_i) &= (\tau_{1,r}, \tau_{1,i}\tau_{3,r}\tau_{3,r}) \end{aligned}$$

The last step in the inner loop of Algorithm 3.1 computes the result for one butterfly update.

$$\begin{aligned} x(kL + j) &:= \tau_0 + \tau_2 \\ x(kL + L_* + j) &:= \tau_1 - i\tau_3 \\ x(kL + 2L_* + j) &:= \tau_0 - \tau_2 \\ x(kL + 3L_* + j) &:= \tau_1 + i\tau_3 \end{aligned}$$

To compute the final value, again data rearrangement and an negation of $\tau_{3,r}$ is needed:

$$\begin{aligned} tmp_0 &= (\tau_{0,r}, \tau_{0,i}, \tau_{1,r}, \tau_{1,i}) \\ tmp_1 &= (\tau_{2,r}, \tau_{2,i}, \tau_{3,i}, -\tau_{3,r}) \end{aligned}$$

Now the result x_0, x_1,x_2 and x_3 can be computed with a vector-addition and a vector-subtraction:

$$\begin{aligned} x_{01} &= tmp_0 +_{vec} tmp_1 &= (\tau_{0,r} + \tau_{2,r}, \tau_{0,i} + \tau_{2,i}, \tau_{1,r} + \tau_{3,i}, \tau_{1,i} - \tau_{3,r}) \\ x_{23} &= tmp_2 -_{vec} tmp_3 &= (\tau_{0,r} - \tau_{2,r}, \tau_{0,i} - \tau_{2,i}, \tau_{1,r} - \tau_{3,i}, \tau_{1,i} + \tau_{3,r}) \end{aligned}$$

So the result is

$$\begin{aligned} x_{01} &= (x_{0,r}, x_{0,i}, x_{1,r}, x_{1,i}) \\ x_{23} &= (x_{2,r}, x_{2,i}, x_{3,r}, x_{3,i}) \end{aligned}$$

As one load or store operation can only access a vector of length 4, two subsequent complex numbers must be loaded at once. So the algorithm outlined above has to be applied to the first part of the loaded vectors (the first complex number) and to the second part of the loaded vectors (the second complex number) which implies again data rearrangement and a special case for stage $q = 1$.

3.1.2 The Split Complex Algorithm

In this variant, all complex data is stored in the *split complex* format. That means a vector of complex numbers is stored as a vector of the real parts and a vector of the imaginary parts. This format is natural for a SIMD FFT algorithm, because real parts and imaginary parts have to be treated in different ways (due to complex multiplications), but as the algorithm is a loop, subsequent real parts and imaginary parts can be calculated simultaneously. The twiddle factors are stored that they can be loaded with an vector-load instruction, too.

In memory the input vector x is stored that way:

$$\begin{aligned} x_r &= (x_{0,r}, x_{1,r}, x_{2,r}, x_{3,r}, x_{4,r}, x_{5,r}, x_{6,r}, \ldots, x_{n-1,r}) \\ x_i &= (x_{0,i}, x_{1,i}, x_{2,i}, x_{3,i}, x_{4,i}, x_{5,i}, x_{6,i}, \ldots, x_{n-1,i}) \end{aligned}$$

with

$$\begin{aligned} x(0) &= x_{0,r} + ix_{0,i} \\ x(1) &= x_{1,r} + ix_{1,i} \\ &\vdots \\ x(n-1) &= x_{n-1,r} + ix_{n-1,i} \end{aligned}$$

Stage $q = 1$

Algorithm 3.2 shows the stage $q = 1$ of Algorithm 3.1. within the k-loop the elements $x(4k)$, $x(4k+1)$, $x(4k+2)$ and $x(4k+3)$ are used for different computations. So only the k-loop can be vectorized. That means the pass $k = 0$, $k = 1$, $k = 2$, $k = 3$ is done simultaneously and so

$$\begin{aligned} \alpha &:= (x(4l), x(4(l+1)), x(4(l+2)), x(4(l+3))) \\ \beta &:= (x(4l+1), x(4(l+1)+1), x(4(l+2)+1), x(4(l+3)+1)) \\ \gamma &:= (x(4l+2), x(4(l+1)+2), x(4(l+2)+2), x(4(l+3)+2)) \\ \delta &:= (x(4l+3), x(4(l+1)+3), x(4(l+2)+3), x(4(l+3)+3)) \end{aligned}$$

and so forth, $l = 0 : n/16 - 1$ is used instead of

$$\begin{aligned} \alpha &:= x(4k) \\ \beta &:= x(4k+1) \\ \gamma &:= x(4k+2) \\ \delta &:= x(4k+3) \end{aligned}$$

$k = 0 : n/4 - 1$.

Applying this rule yields Algorithm 3.5 for the computation of stage $q = 1$.

Algorithm 3.5 (Vectorized Stage $q = 1$)

$r := n/16$
do $l = 0 : r - 1$
$\alpha := (x(4l), x(4(l+1)), x(4(l+2)), x(4(l+3)))$
$\beta := (x(4l+1), x(4(l+1)+1), x(4(l+2)+1), x(4(l+3)+1))$
$\gamma := (x(4l+2), x(4(l+1)+2), x(4(l+2)+2), x(4(l+3)+2))$
$\delta := (x(4l+3), x(4(l+1)+3), x(4(l+2)+3), x(4(l+3)+3))$
$\tau_0 := \alpha +_{vec} \gamma$
$\tau_1 := \alpha -_{vec} \gamma$
$\tau_2 := \beta +_{vec} \delta$
$\tau_3 := \beta -_{vec} \delta$
$(x(4l), x(4(l+1)), x(4(l+2)), x(4(l+3))) := \tau_0 +_{vec} \tau_2$
$(x(4l+1), x(4(l+1)+1), x(4(l+2)+1), x(4(l+3)+1)) := \tau_1 -_{vec} i\tau_3$
$(x(4l+2), x(4(l+1)+2), x(4(l+2)+2), x(4(l+3)+2)) := \tau_0 -_{vec} \tau_2$
$(x(4l+3), x(4(l+1)+3), x(4(l+2)+3), x(4(l+3)+3)) := \tau_1 +_{vec} i\tau_3$
end do

Now the elements of $\alpha = (\alpha_0, \alpha_1, \alpha_2, \alpha_3), \beta, \gamma$ and δ are not continous in memory. For the computation the 4-element vectors α_r, α_i ($\alpha_0 = \alpha_{0,r} + i\alpha_{0,i}$, $\alpha = \alpha_r +_{vec} i\alpha_i$) and so forth are needed. Every operation in Algorithm 3.5 is a complex operation that translates into 2 real vector operations.

α, β, γ and δ can be loaded with 4 vector load operations and a transpose operation.With 4 load operations a temporary vector can be loaded:

$$\begin{pmatrix} tmp_1 \\ tmp_2 \\ tmp_3 \\ tmp_4 \end{pmatrix} = \begin{pmatrix} x_{r,0} & x_{r,1} & x_{r,2} & x_{r,3} \\ x_{r,4} & x_{r,5} & x_{r,6} & x_{r,7} \\ x_{r,8} & x_{r,9} & x_{r,10} & x_{r,11} \\ x_{r,12} & x_{r,13} & x_{r,14} & x_{r,15} \end{pmatrix}$$

The transposed temporary matrix consists of the needed vectors α, β, γ and δ:

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} x_{r,0} & x_{r,4} & x_{r,8} & x_{r,12} \\ x_{r,1} & x_{r,5} & x_{r,9} & x_{r,13} \\ x_{r,2} & x_{r,6} & x_{r,10} & x_{r,14} \\ x_{r,3} & x_{r,7} & x_{r,11} & x_{r,15} \end{pmatrix}$$

In stage $q = 1$, this method has to be used for loading and storing the data vector.

Stages $q = 2$ to $q = t - 1$

Because of the unit stride resulting from the vectorized j-loop and the split complex storage format, α, β, γ and δ can be loaded directly:

$$\begin{pmatrix}\alpha\\ \beta\\ \gamma\\ \delta\end{pmatrix} = \begin{pmatrix} x_{r,0} & x_{r,1} & x_{r,2} & x_{r,3}\\ x_{r,4} & x_{r,5} & x_{r,6} & x_{r,7}\\ x_{r,8} & x_{r,9} & x_{r,10} & x_{r,11}\\ x_{r,12} & x_{r,13} & x_{r,14} & x_{r,15}\end{pmatrix}$$

In one pass, the j-loop for $j = l$, $j = l+1$, $j = l+2$ and $j = l+3$ are computed simultaneously. This results in Algorithm 3.6.

Algorithm 3.6 (Stages $q = 2$ to $q = t-1$)

do $q = 2 : t-1$
  $L := 4^q$
  $r := n/L$
  $L_* := L/4$
  do $l = 0 : L_*/4 - 1$
    $\omega := (e^{-2\pi i \cdot 4l/L}, e^{-2\pi i(4l+1)/L}, e^{-2\pi i(4l+2)/L}, e^{-2\pi i(4l+3)/L})$
    do $k = 0 : r-1$
      $b := kL + 4l$
      $\alpha := (x(b), x(b+1)), x(b+2), x(b+3)))$
      $\beta := \omega \cdot_{vec} (x(b+L_*), x(b+L_*+1)), x(b+L_*+2), x(b+L_*+3)))$
      $\gamma := \omega^2 \cdot_{vec} (x(b+2L_*), x(b+2L_*+1)), x(b+2L_*+2), x(b+2L_*+3)))$
      $\delta := \omega^3 \cdot_{vec} (x(b+3L_*), x(b+3L_*+1)), x(b+3L_*+2), x(b+3L_*+3)))$
      $\tau_0 := \alpha +_{vec} \gamma$
      $\tau_1 := \alpha -_{vec} \gamma$
      $\tau_2 := \beta +_{vec} \delta$
      $\tau_3 := \beta -_{vec} \delta$
      $(x(b), x(b+1)), x(b+2), x(b+3))) := \tau_0 + \tau_2$
      $(x(b+L_*), x(b+L_*+1)), x(b+L_*+2), x(b+L_*+3))) := \tau_1 -_{vec} i\tau_3$
      $(x(b+2L_*), x(b+2L_*+1)), x(b+2L_*+2), x(b+2L_*+3))) := \tau_0 -_{vec} \tau_2$
      $(x(b+3L_*), x(b+3L_*+1)), x(b+3L_*+2), x(b+3L_*+3))) := \tau_1 +_{vec} i\tau_3$
    end do
  end do
end do

Stage $q = t$

In this stage the k-loop degenerates. But this causes no problems in the vectorization as the j-loop is vectorized. This yields unit stride access in the j-loop again. This subproblem is solved by Algorithm 3.7

Algorithm 3.7 (Stage $q = t$)

$L_* := n/4$
do $l = 0 : L_*/4 - 1$

$\omega := (e^{-2\pi i \cdot 4l/n}, e^{-2\pi i(4l+1)/n}, e^{-2\pi i(4l+2)/n}, e^{-2\pi i(4l+3)/n})$
$\alpha := (x(4l), x(4l+1)), x(4l+2), x(4l+3)))$
$\beta := \omega \cdot_{vec} (x(4l+2L_*), x(4l+2L_*+1)), x(4l+2L_*+2), x(4l+2L_*+3)))$
$\gamma := \omega^2 \cdot_{vec} (x(4l+2L_*), x(4l+2L_*+1)), x(4l+2L_*+2), x(4l+2L_*+3)))$
$\delta := \omega^3 \cdot_{vec} (x(4l+3L_*), x(4l+3L_*+1)), x(4l+3L_*+2), x(4l+3L_*+3)))$
$\tau_0 := \alpha +_{vec} \gamma$
$\tau_1 := \alpha -_{vec} \gamma$
$\tau_2 := \beta +_{vec} \delta$
$\tau_3 := \beta -_{vec} \delta$
$(x(4l), x(4l+1)), x(4l+2), x(4l+3))) := \tau_0 +_{vec} \tau_2$
$(x(4l+2L_*), x(4l+2L_*+1)), x(4l+2L_*+2), x(4l+2L_*+3))) := \tau_1 -_{vec} i\tau_3$
$(x(4l+2L_*), x(4l+2L_*+1)), x(4l+2L_*+2), x(4l+2L_*+3))) := \tau_0 -_{vec} \tau_2$
$(x(4l+3L_*), x(4l+3L_*+1)), x(4l+3L_*+2), x(4l+3L_*+3))) := \tau_1 +_{vec} i\tau_3$

end do

3.2 A SIMD Version of FFTW

3.2.1 The FFTW Package

One of the best known FFT packages is FFTW [5], the "Fastest Fourier Transform in the West". Its elegant computer-generated code, its high usability and superior performance made the authors M. Frigo and S. Johnson win the 1999 Wilkinson Prize for Numerical Software.

Homepage: `www.fftw.org`

FFTW is one of the few FFT packages that can be regarded as a modern piece of software. The package is highly portable and its design makes it easy to incorporate new FFT kernels into its framework. This, and the fact that FFTW is able to choose the best performing kernels for a given architecture based on run-time tests lets FFTW exploit entirely different computer architectures to a very high degree. Recently a cooperation between AURORA Group 5 and Matteo Frigo was initiated to develop new hardware adaptive FFT algorithms.

The complete source code of FFTW can be downloaded from the FFTW Web site. FFT benchmark results (obtained with the benchmark suite `benchFFT`) are given at this Web site. Many useful references are given to other FFT resources.

The Structure of FFTW

FFTW is an *architecture adaptive* FFT program, written in C. The computation of the transform is accomplished by an *executor* that consists of highly optimized, composeable blocks of C code called *codelets*. A codelet is a specialized piece of

code, that computes a part of the transform. The combination of codelets applied by the executor is specified by a special data structure called a *plan*. The plan is determined at runtime, before the computation begins, by a *planner* which uses a dynamic programming algorithm to find a fast composition of codelets. The planner tries to minimize the actual *execution time*. So the planner measures the runtime of many plans and selects the fastest. To achieve highest speed, a lot of highly optimized codelets are needed. In FFTW these are generated automatically by an *codelet generator*. This program is a special purpose compiler that can generate codelets for any size.

The following code-fragment shows, how FFTW is used at runtime:

```
fftw_plan plan;
fftw_complex A[n], B[n];

/* plan the computation */
plan=fftw_create_plan(n);

/* execute the plan */
fftw_one(plan, A);

/* the plan can be reused for other inputs of size n */
fftw_one(plan, B);
```

The Executor

The executor implements the Cooley-Tukey FFT algorithm as described in (1.3), but for arbitrary N. If $N = N_1 \cdot N_2$, the executor recursively computes N_1 transforms of size N_2, multiplies the result by the *twiddle factors* and finally computes N_2 transforms of size N_1. To compute the transforms, the codelets are used. For the base case of the recursion the *no twiddle codelets* are used. They implement a special case of the Cooley-Tukey algorithm and calculate the FFT for a given size. Within the recursion the *twiddle codelets* are used. They are like no twiddle codelets, but in addition they multiply their input by the twiddle factors.

The executor takes as input the array to be transformed, and also a plan. The plan specifies the factorization of N as well as which codelets should be used. The executor works by explicit recursion.

The Planner

The planner uses a dynamic programming strategy to find the fastest plan for a given transform size N. Due to combinatorial explosion it is not possible to try all possible plans. In order to use dynamic programming, the *optimal substructure* is assumed: if an optimal plan for size N is known, this plan is still optimal when size N is used as a subproblem of a larger transform. This assumption is in principle

false because of different states of cache in the two cases. But it turns out that this approach yields good results.

FFTW in Pseudocode

The following pseudocode shows how the parts of FFTW interact on execution. The function `main` initializes FFTW and plans the computation of the FFT via the `planner`. The actual computation is done by the `executor` which calls the appropriate `codelets`.

```
main()
{
  complex in[n], out[n]
  plan p

  n:=mysize
  p:=planner(n)
  input(in)

  fftw(in, out, p)

  output(out)
}


fft(input, output, plan)
{
  execute(plan.n, input, output, plan, 1, 1)
  return output
}


execute(n, input, output, plan, instride, outstride)
{
  if plan.type = NO_TWIDDLE
    plan.notwiddlecodelet(input, output, instride, outstride)
  if plan.type = TWIDDLE
    r:=plan.twiddlesize
    m:=n/r
    for i:=0 to r-1 do
      execute(m, input+i*instride, output+i*m*outstride,
        plan.nextstep, instride*r, outstride)
    plan.twiddlecodelet(output, plan.twiddlefactors,
      m * outstride, m, outstride);
}


notwiddlecodelet<N>(input, output, instride, outstride)
```

```
{
  complex in[N], out[N]

  for i:=0 to N-1 do
    in[i]:=input[i*instride]

  FFT(N, in, out)

  for i:=0 to N-1 do
    output[i*outstride]:=out[i]
  return output
}


twiddlecodelet<N>(inoutput, twiddlefactors, stride,  m, dist)
{
  complex in[N], out[N]

  for i:=0 to m-1
    for j:=0 to N-1
      in[i]:=inoutput[i*dist+j*stride]

    FFT(N, in, out)
    apply_twiddlefactors(N, out, i, twiddlefactors)

    for j:=0 to N-1
      inoutput[i*dist+j*stride]:=out[j]
}
```

To vectorize FFTW, two different approaches can be used.

1. The computation in the codelets can be vectorized, if the codelet contains a loop. The vectorization results in computing some passes of the loop simultaneously. This is true for the twiddle-codelets.

2. If more than one codelet has to be executed with the same parameters on strided data, subsequent calls to this codelet can be replaced by one call to a vectorized form of this codelet. This can be done with the no twiddle codelet as well as with the twiddle codelet. The different methods result in a different number of memory accesses due to the way, twiddle factors are needed. The cache locality of FFTW can be perturbed as memory accesses are reordered.

The vectorization requires changes in the codelets, the executor and the planner as well as in FFTWs internal data structures. Every codelet (no twiddle and twiddle) gets an internally vectorized and an externally vectorized version associated. Wherever a codelet (i. e., a pointer to a codelet) is saved, after vectorization three pointers to the associated codelets are saved. So the planner has to initialize

the plans in a different way. These pointers are used by the executor, to use the appropriate vectorized codelet instead a loop of standard codelets.

3.2.2 The SIMD Macro Framework

SIMD vectorization is a highly machine dependent process with no common language extension or API. But there are only a few operations needed within the vectorization process of FFTW. These are standard operations like load a complex number, store a complex number, extract the real parts into a vector, extract the imaginary parts into a vector, build complex numbers from a vector of real parts and a vector of imaginary parts, add two vectors, subtract two vectors, and multiply two vectors.

Additionally on every platform the name of the vector data types and the initialization methods are different.

This differences can all be covered by a single include file, that defines operations, data types and initializations according to the underlying platform and compiler.

Data Types

Two data types are required: `FFTW_SIMD_VECT` is the native SIMD vector data type. In the current implementation this are 4 element vectors. But on different platforms this could be, e.g., 2 element vectors, too.

`FFTW_SIMD_COMPLEX` is the data type used to access data of type `fftw_complex` within a SIMD-codelet.

Data Load Operation

`LOAD_RE_IM(re, im, input, stride)` loads 4 complex numbers from `input`, `input+stride`, `input+2*stride`, `input+3*stride`, saves their real parts into the vector `re` and their imaginary parts into the vector `im`. This high level macro consists of 4 complex load operations and some shuffle operations.

Twiddle Factor Load Operation

`LOAD_TWIDDLE_RE_IM(re, im, tw)` loads a complex number from location `tw` and stores the real part in each element of the vector `re` and the imaginary part in each element of the vector `im`. This high level macro consists of one complex load operation and some shuffling.

Data Store Operation

`STORE_RE_IM(re, im, output, stride)` stores 4 complex numbers to `output`, `output+stride`, `output+2*stride`, `output+3*stride`. These numbers are built from the real part vector `re` and the imaginary part vector `im`. This high level macro consists of 4 complex store operations and some shuffle operations. The complex store operations may result in 8 floating-point stores or 4 floating-point dupel stores, depending on the platform.

Constant Handling Environment

Constants are declared using the macro `FFTW_SIMD_KONST(c, v)` which declares the constant `c` as a vector whose elements are all initialized with the value `v`.

To use a constant, the macro `FFTW_LOAD_KONST_SIMD(c)` is used. This macro returns the addressed constant `c`. It is needed to cover the syntactic differences in accessing SIMD constants on the different platforms.

Arithmetic Operations

`SIMD_ADD(a, b)` returns the vector addition `a + b`. `SIMD_SUB(a, b)` returns the vector subtraction `a - b`. `SIMD_MUL(a, b)` returns the vector of the component-wise products of `a` and `b`. On multiply-add capable architectures, two more arithmetic operations are used: `SIMD_MADD(a, b, c)` and `SIMD_NMSUB(a, b, c)`.

3.2.3 Vectorizing Codelets

To vectorize a codelet, either a loop has to be vectorized or the computation of 4 codelets are put into one vector codelet. All the following transformations and changes to the codelets were included into the FFTW codelet generator. The SIMD codelets are generated automatically as the FPU codelets were.

In the FPU codelet the real parts and the imaginary parts can be accessed independently. That is done via the instructions

```
tmp1 = c_re (input[0]);
tmp7 = c_im (input[0]);
```

In an SIMD codelet, the real part and the imaginary part have to be loaded with one macro. This results in the corresponding code:

```
LOAD_RE_IM (tmp1, tmp7, input + (0), ivectstride);
```

The same operation replacement is needed for storing the data.

```
c_re (output[0]) = (tmp1 + tmp4);
c_im (output[0]) = (tmp10 + tmp11);
```

is transformed into

```
STORE_RE_IM (SIMD_ADD (tmp1, tmp4),
    SIMD_ADD (tmp10, tmp11), output + (0), ovectstride);
```

Any arithmetic operation has to be transformed into the corresponding macros:

```
static const fftw_real K866025403 = FFTW_KONST (+0.86602540);
tmp9 = K866025403 * (tmp3 - tmp2);
```

is transformed into

```
FFTW_SIMD_KONST (K866025403_SIMD, +0.86602540);
tmp9 = SIMD_MUL (FFTW_LOAD_KONST_SIMD(K866025403_SIMD),
    SIMD_SUB (tmp3, tmp2));
```

No Twiddle Codelet Vectorization

The no twiddle codelet does not contain a loop. As the input and output data is not guaranteed to have unit stride, 2 new parameters have to be introduced. The original function prototype

```
void fftw_no_twiddle_XX (const fftw_complex * input,
    fftw_complex * output, int istride, int ostride)
```

is replaced by

```
void fftw_no_twiddle_simd_XX (FFTW_SIMD_COMPLEX * input,
    FFTW_SIMD_COMPLEX * output, int istride, int ostride,
    int ivectstride, int ovectstride)
```

The new parameter `int ivectstride` and `int ovectstride` contain the stride of the data only known in the outer loop of executor, that calls the codelet.

Twiddle Codelet Outer Vectorization

The twiddle codelet contains a loop. But it is possible to vectorize it like an no twiddle codelet. That means again, that a part of the loop of the executor is moved into the codelet. This results in a codelet, that computes the same values as 4 subsequent calls to the FPU codelet with the appropriate data. Again a stride parameter from the executor is needed. Therefore the prototype is transformed from

```
void fftw_twiddle_XX (fftw_complex * A, const fftw_complex * W,
    int iostride, int m, int dist)
```

into

```
void fftw_twiddle_simd_XX (FFTW_SIMD_COMPLEX * A, FFTW_SIMD_COMPLEX * W,
    int iostride, int m, int dist, int iovectstride)
```

Due to the structure of the executor, all calls to a codelet in the executor loop are conducted with the same twiddle factor array. That means, that one complex number has to be loaded and the real part must be written into all components of the `re` vector and the imaginary part must be written into all components of the `im` vector. So the twiddle load operation changes from

```
tmp13 = c_re (W[2]);
tmp15 = c_im (W[2]);
```

into

```
LOAD_TWIDDLE_RE_IM (tmp13, tmp15, W + (2));
```

Twiddle Codelet Inner Vectorization

For the inner vectorization, no major changes to the codelet prototype are needed. Only the correct SIMD data types have to be used. The prototype changes from

```
void fftw_twiddle_XX (fftw_complex * A, const fftw_complex * W,
    int iostride, int m, int dist)
```

into

```
void fftw_twiddle_simd_int_XX (FFTW_SIMD_COMPLEX * A,
    FFTW_SIMD_COMPLEX * W, int iostride, int m, int dist)
```

The internal loop is vectorized. That means, four passes of the loop are carried out simultaneously. The loop instruction transforms from

```
inout = A;
for (i = m; i > 0; i = (i - 1), inout = (inout + dist), W = (W + 3))
```

into

```
inout = A;
m >>= FFTW_LD_SIMD_LEN;
for (i = m; i > 0; i = (i - 1), inout = (inout + FFTW_SIMD_LEN * dist),
    W = (W + FFTW_TWIDDLE_STRIDE_XX * FFTW_SIMD_LEN))
```

Every pass in the loop in a twiddle codelet has its own twiddle factor. That means, in the vectorized pass four different twiddle factors are needed. This can be done by using the load macro. The original code

```
tmp2 = c_re (W[1]);
tmp4 = c_im (W[1]);
```

is transformed into

```
LOAD_RE_IM (tmp2, tmp4, W + (1), FFTW_TWIDDLE_STRIDE_XX);
```

3.2.4 Changes to the Executor

The executor has to use the new pointers to the vectorized codelets. If the number of passes in the loop is not a multiple of four, a mixture of vectorized and standard codelets is needed. E. g., for 10 passes, 2 vectorized passes and 2 standard passes are needed. This must be handled for the executor loop and for the loop in the twiddle codelets within the executor and results in an different index computation within the executor. The recursion entry point is the function `fftw_executor_simple` and all further steps are done by calls to `executor_many_vector`.

In `fftw_executor_simple`, the no twiddle case can only occur with only one call to the codelet as it is the entry point of the recursion. So this case cannot be vectorized. The key point is the twiddle case. It is modified as it follows: The original code

```
case FFTW_TWIDDLE:
     {
      int r = p->nodeu.twiddle.size;
      int m = n / r;
      fftw_twiddle_codelet *codelet;
      fftw_complex *W;

      executor_many_vector(m, in, out,
                   p->nodeu.twiddle.recurse,
                   istride * r, ostride,
                   r, istride, m * ostride);

      codelet = p->nodeu.twiddle.codelet;
      W = p->nodeu.twiddle.tw->twarray;

      codelet(out, W, m * ostride, m, ostride);
     }
```

is rewritten into

```
case FFTW_TWIDDLE:
     {
      int r = p->nodeu.twiddle.size;
      int m = n / r;
      int m1=(m/4)<<2,m2=m%4;

      fftw_twiddle_codelet *codelet;
      fftw_twiddle_codelet_simd_int *vectintcodelet;
      fftw_complex *W;

      executor_many_vector(m, in, out,
                   p->nodeu.twiddle.recurse,
                   istride * r, ostride,
                   r, istride, m * ostride);
      codelet=p->nodeu.twiddle.codelet;
      W = p->nodeu.twiddle.tw->twarray;

      vectintcodelet = p->nodeu.twiddle.vectintcodelet;
      vectintcodelet((FFTW_SIMD_COMPLEX *)out, (FFTW_SIMD_COMPLEX *)W,
          m * ostride, m, ostride);
      if (m2) codelet(out + m1*ostride, W+m1*(p->nodeu.twiddle.size-1),
          m * ostride, m2, ostride);
      }
```

In the `executor_many_vector`, both the twiddle and the no twiddle case have to be changed. The no twiddle case occurs with a lot of repetitions and is the leaf of the recursion. So the vectorized version of the codelets can be used. The original no twiddle code

```
case FFTW_NOTW:
     {
      fftw_notw_codelet *codelet = p->nodeu.notw.codelet;
      for (s = 0; s < howmany; ++s)
       codelet(in + s * idist,
           out + s * odist,
           istride, ostride);
     }
```

is rewritten as

```
case FFTW_NOTW:
     {
      fftw_notw_codelet *codelet = p->nodeu.notw.codelet;
      fftw_notw_codelet_simd *vectcodelet=p->nodeu.notw.vectcodelet;
      for (s = 0; s < (howmany/4); ++s)
       vectcodelet(((FFTW_SIMD_COMPLEX *)in) + 4 * s * idist,
           ((FFTW_SIMD_COMPLEX *)out) + 4 * s * odist,
           istride, ostride,idist,odist);

      for (s = (howmany - (howmany%4)); s < howmany; ++s)
        codelet(in + s * idist,
            out + s * odist,
            istride, ostride);
     }
```

In twiddle case, both the internally and the externally vectorized case is possible. The original code

```
case FFTW_TWIDDLE:
     {
      int r = p->nodeu.twiddle.size;
      int m = n / r;
      fftw_twiddle_codelet *codelet;
      fftw_complex *W;

      for (s = 0; s < r; ++s)
        executor_many_vector(m, in + s * istride,
                    out + s * (m * ostride),
                    p->nodeu.twiddle.recurse,
                    istride * r, ostride,
                    howmany, idist, odist);

      codelet = p->nodeu.twiddle.codelet;
      W = p->nodeu.twiddle.tw->twarray;

      for (s = 0; s < howmany; ++s)
        codelet(out + s * odist, W, m * ostride, m, ostride);
    }
```

can be modified in two different ways: The first method leads to the external vectorized version. Only the standard codelets and the external vectorized codelets are used. This vectorization may lead to cache problems, as elements with big power of two strides are loaded within a few lines of code within the codelet. But this implementation is more obvious and faster to implement. It results in the following code:

```
case FFTW_TWIDDLE:
     {
      int r = p->nodeu.twiddle.size;
      int m = n / r;
      fftw_twiddle_codelet *codelet;
      fftw_twiddle_codelet_simd *vectcodelet;
      fftw_complex *W;

      for (s = 0; s < r; ++s)
        executor_many_vector(m, in + s * istride,
                    out + s * (m * ostride),
                    p->nodeu.twiddle.recurse,
                    istride * r, ostride,
                    howmany, idist, odist);

      codelet = p->nodeu.twiddle.codelet;
      vectcodelet = p->nodeu.twiddle.vectcodelet;
      W = p->nodeu.twiddle.tw->twarray;
      for (s = 0; s < (howmany/4); ++s)
        vectcodelet(((FFTW_SIMD_COMPLEX *)out) + 4 * s * odist,
           (FFTW_SIMD_COMPLEX *) W,
           m * ostride, m, ostride, odist);
      for (s = (howmany - (howmany%4)); s < howmany; ++s)
        codelet(out + s * odist, W, m * ostride, m, ostride);

      break;
     }
```

The internally vectorized version has the same data access pattern as the original FFTW version and has a better data locality than the externally vectorized version. As the no twiddle case cant be vectorized internally, two different vectorization methods have to be used: the external version for the no twiddle case and the internally vectorization for the twiddle case. This leads to a more complex executor implementation. The internally vectorized twiddle codelets lead to the following code:

```
case FFTW_TWIDDLE:
     {
      int r = p->nodeu.twiddle.size;
      int m = n / r;
```

```
int m1=(m/4)<<2,m2=m%4;

fftw_twiddle_codelet *codelet;
fftw_twiddle_codelet_simd_int *vectintcodelet;
fftw_complex *W;

for (s = 0; s < r; ++s)
  executor_many_vector(m, in + s * istride,
              out + s * (m * ostride),
              p->nodeu.twiddle.recurse,
              istride * r, ostride,
              howmany, idist, odist);

codelet = p->nodeu.twiddle.codelet;
vectintcodelet = p->nodeu.twiddle.vectintcodelet;
W = p->nodeu.twiddle.tw->twarray;

for (s = 0; s < howmany; ++s)
{
  vectintcodelet(((FFTW_SIMD_COMPLEX *)out) + s * odist,
     (FFTW_SIMD_COMPLEX *) W, m * ostride, m1, ostride);
  if (m2) codelet(out + s * odist+m1*ostride,
     W+m1*(p->nodeu.twiddle.size-1), m * ostride, m2, ostride);
}
}
```

3.2.5 Changes to the Planner

The FFTW SIMD implementation offers a new flag for the planner, to enable SIMD vectorization. The call to the planner now looks like

```
p = fftw_create_plan(n, FFTW_FORWARD, FFTW_MEASURE|FFTW_USE_SIMD);
```

The flag `FFTW_USE_SIMD` enables FFTWs vector recursion and the SIMD vectorization.

In the planner only the flag `ENABLE_VECTOR_RECURSION` has to be set to the plan, and the computation is done via SIMD, wherever possible. Little modifications in the planner utilities were needed, to store the pointer to the vector codelets in the plan.

Chapter 4

Results on the Pentium III

All experiments described in this chapter were carried out on the following system:

CPU	Intel Pentium III Coppermine 650 MHz, 100 MHz FSB 16 kB L1 Data Cache 16 kB L1 Instruction Cache 256 kB on Chip L2 Cache 2.6 Gflop/s
RAM	256 MB 100MHz SDRAM
Operating System	Windows 2000
Compiler	Intel C/C++ Compiler 4.5

4.1 SIMD Algorithms vs. FPU Algorithms

In this section SIMD FFTW routines are compared to the currently fastest FPU FFT implementation (FFTW). The following routines are considered:

Radix-4 Triple Loop Interleaved Complex Algorithm as derived in Section 3.1.1.

Radix-4 Triple Loop Split Complex Algorithm as derived in Section 3.1.2.

Intel MKL Library: This vendor supplied library uses the Intel SSE accelerate the calculation.

FFTW Internally Vectorized Version as derived in Section 3.2.3. It turns out to be the fastest SIMD implementation of FFTW in Section 4.2.2.

FFTW Single Precision FPU Version as described in Section 3.2.1. It is the fastest machine-independent implementation of the FFT algorithms on various platforms, see [2].

Runtime Results

In Table 4.1 and in Figure 4.1 the runtime of these routines are compared. For in-cache problems, the split complex radix-4 algorithm is the fastest. But the Intel

MKL Library and the FFTW internally vectorized version come very close to its performance. FFTW's memory locality makes the two FFTW versions faster in the out-of-cache case. The combination of SIMD floating-point performance and FFTW memory locality gives the best performance in this area.

size	R4 Interleaved	R4 Split	Intel MKL	FFTW int.	FFTW FPU
2^4	3.22×10^2	2.65×10^2	6.44×10^2	4.33×10^2	4.29×10^2
2^6	2.28×10^3	1.47×10^3	2.00×10^3	1.90×10^3	2.20×10^3
2^8	1.35×10^4	7.89×10^3	8.32×10^3	8.24×10^3	1.27×10^4
2^{10}	7.42×10^4	3.97×10^4	5.21×10^4	4.12×10^4	6.58×10^4
2^{12}	4.04×10^5	2.22×10^5	2.54×10^5	2.18×10^5	3.39×10^5
2^{14}	2.88×10^6	1.11×10^6	1.60×10^6	1.51×10^6	2.27×10^6
2^{16}	3.34×10^7	2.19×10^7	1.56×10^7	1.20×10^7	1.66×10^7
2^{18}	1.64×10^8	1.22×10^8	1.03×10^8	5.79×10^7	7.07×10^7
2^{20}	7.15×10^8	5.41×10^8	4.65×10^8	2.50×10^8	3.71×10^8

Table 4.1: Runtimes in cycles.

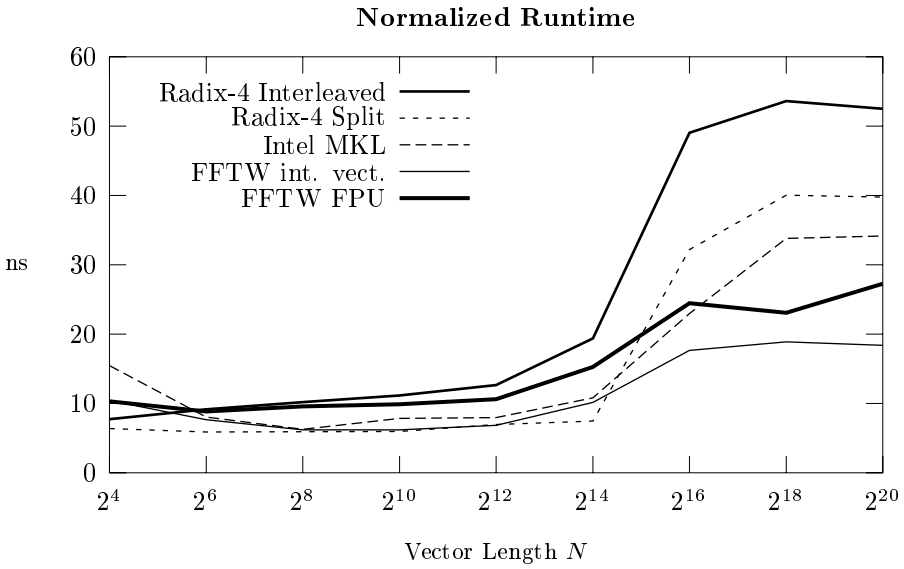

Figure 4.1: Performance of SIMD algorithms compared to the FPU version of FFTW.

Floating-Point Performance Results

In Table 4.2 and in Figure 4.2 the floating-point performance of these routines are compared. For the in-cache case the split complex radix-4 algorithm runs at about 23 % of the peak performance. But due to his higher arithmetic complexity it does not run faster than FFTW, internally vectorized version.

size	R4 Int.	R4 Split	Intel MKL	FFTW int.	FFTW FPU
2^4	18.0 %	18.9 %	7.8 %	8.8 %	7.6 %
2^6	17.6 %	22.9 %	14.4 %	16.1 %	7.8 %
2^8	17.0 %	23.9 %	20.4 %	19.3 %	9.1 %
2^{10}	16.2 %	24.5 %	16.8 %	19.3 %	10.0 %
2^{12}	14.7 %	21.4 %	16.9 %	19.6 %	9.9 %
2^{14}	9.8 %	20.2 %	13.1 %	12.1 %	7.5 %
2^{16}	3.9 %	4.7 %	6.6 %	7.8 %	4.8 %
2^{18}	3.6 %	3.8 %	4.5 %	6.7 %	4.8 %
2^{20}	3.7 %	3.9 %	4.6 %	4.6 %	5.0 %

Table 4.2: Relative floating-point performance (100 % = 2.6 Gflop/s).

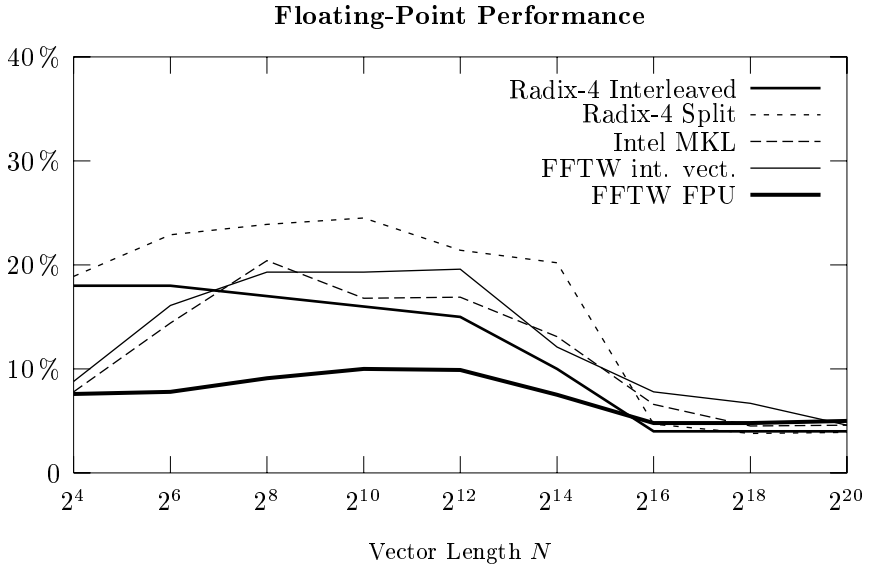

Figure 4.2: Relative floating-point performance of SIMD algorithms compared to the FPU version of FFTW (100 % = 2.6 Gflop/s).

4.2 Results on FFTW

4.2.1 Codelet Performance

No Twiddle Codelets

An SIMD no twiddle codelet does the same as 4 FPU no twiddle codelets. In this section always the runtime of 4 calls to the FPU no twiddle codelets is compared to the runtime of one call to the corresponding SIMD no twiddle codelet. The SIMD no twiddle codelet is between 1.26 and 1.84 times faster than the FPU

version (see Table 4.3 and Figure 4.3).

size	FPU codelet cycles	SIMD codelet cycles	speed-up
1	70	46	1.52
2	112	89	1.26
3	231	132	1.75
4	230	164	1.40
5	420	271	1.55
6	463	309	1.50
7	804	509	1.58
8	560	386	1.45
9	1,034	576	1.80
10	995	617	1.61
11	1,852	1,006	1.84
12	1,077	697	1.55
13	2,027	1,171	1.73
14	1,794	1,051	1.71
15	1,661	1,111	1.50
16	1,419	942	1.51
32	3,528	2,114	1.67
64	8,801	6,944	1.27

Table 4.3: Runtime (in cycles) of no twiddle codelets.

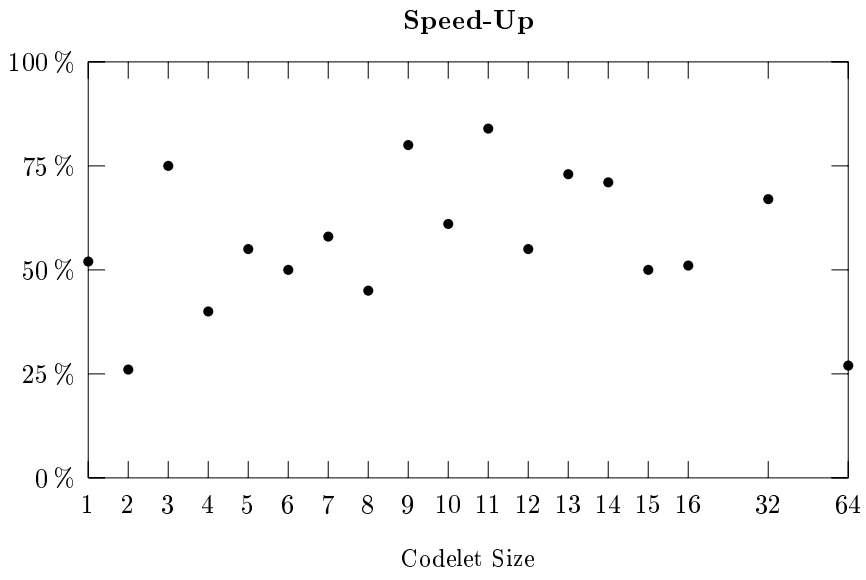

Figure 4.3: No twiddle codelets speed-up.

Externally Vectorized Twiddle Codelets

The externally vectorized SIMD twiddle codelet does the same as 4 FPU twiddle codelets. In this section always the runtime of 4 calls to the FPU twiddle codelet is compared to the runtime of one call to the corresponding externally vectorized SIMD twiddle codelet. The externally vectorized SIMD twiddle codelet is between 1.20 and 1.69 times faster than the FPU version (see Table 4.4 and Figure 4.4).

size	FPU codelet cycles	SIMD codelet cycles	speed-up
2	454	323	1.41
3	856	608	1.41
4	1,088	848	1.28
5	1,909	1,245	1.53
6	2,137	1,508	1.42
7	3,612	2,362	1.53
8	2,964	2,045	1.45
9	4,855	2,870	1.69
10	4,672	3,123	1.50
16	7,408	5,003	1.48
32	17,645	11,026	1.60
64	42,596	35,561	1.20

Table 4.4: Runtime (in cycles) of externally vectorized twiddle codelets.

Internally Vectorized Twiddle Codelets

The internally vectorized SIMD twiddle codelet does the same as one FPU twiddle codelet. In this section always the runtime of one call to the FPU twiddle codelet is compared to the runtime of one call to the corresponding internally vectorized SIMD twiddle codelet. The internally vectorized SIMD twiddle codelet is between 0.98 and 1.69 times faster than the FPU version (see Table 4.5 and Figure 4.4).

4.2.2 FFTW Performance Results

In this section 3 variants of FFTW are compared. All of them are single precision versions.

FPU Version: This is the standard FFTW single-precision version.

Externally Vectorized Version: This is the SIMD-vectorized version, that uses the externally vectorized twiddle codelets.

Internally Vectorized Version: This is the SIMD-vectorized version, that uses the internally vectorized twiddle codelets.

size	FPU codelet cycles	SIMD codelet cycles	speed-up
2	106	108	0.98
3	224	195	1.15
4	295	266	1.11
5	501	397	1.32
6	608	431	1.41
7	900	660	1.36
8	1,680	576	1.33
9	1,251	739	1.69
10	1,180	876	1.35
16	1,969	1,272	1.55
32	4,477	2,955	1.52
64	10,780	9,014	1.20

Table 4.5: Runtime (in cycles) of internally vectorized twiddle codelets.

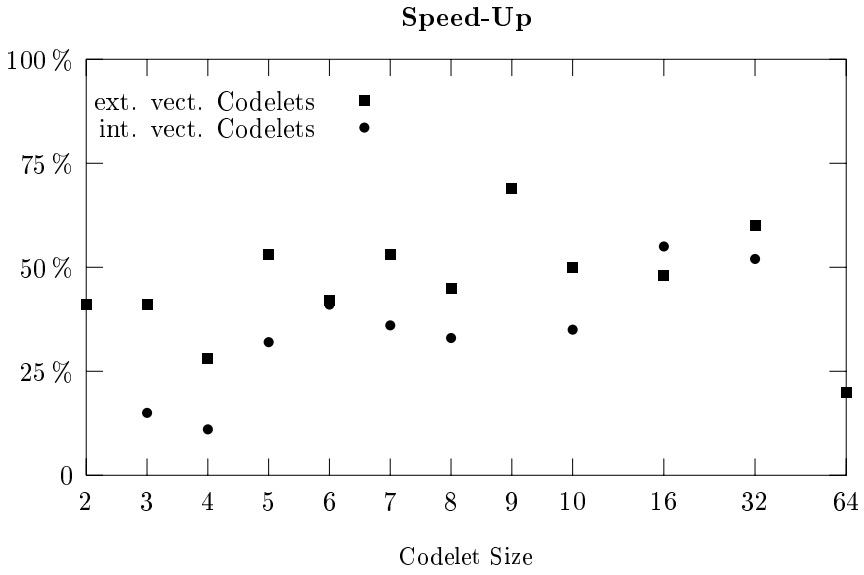

Figure 4.4: Twiddle codelets speed-up.

Runtime Results

In Table 4.6 and Figure 4.5 the runtimes for transform sizes of $n = 2^k$ are compared. In Table 4.7 and Figure 4.7 the runtimes for transform sizes of $n = k \cdot 10^d$ are compared. One can see, that the internally vectorized version is about a factor of 1.5 to 1.65 faster than the FPU version for the in-cache case ($n < 10,000$) and about 1.25 times faster for the out-of-cache case. For very big powers of 2, the speed-up degrades. (see Figure 4.6 and Figure 4.8).

The speed-up for the externally vectorized version is usually less than the speed-

up for the internally vectorized version. This is due to cache associativity problems for large power-of-2 strides and a lower SIMD utilization.

For transform sizes of $n = 2^k$ a better speed-up is achieved than for transform sizes of $n = k \cdot 10^d$. This also reflects cache associativity problems due to large power-of-2 strides. The difference between the externally vectorized version and the internally vectorized version is minimal for the case $n = k \cdot 10^d$.

size	FPU version	externally vectorized version		internally vectorized version	
	cycles	cycles	speed-up	cycles	speed-up
2^4	4.29×10^2	4.29×10^2	1.00	4.33×10^2	0.99
2^5	9.90×10^2	9.91×10^2	1.00	9.88×10^2	1.00
2^6	2.21×10^3	2.04×10^3	1.08	1.91×10^3	1.16
2^7	5.83×10^3	4.18×10^3	1.39	3.89×10^3	1.50
2^8	1.28×10^4	1.01×10^4	1.26	8.25×10^3	1.55
2^9	2.95×10^4	2.29×10^4	1.29	1.87×10^4	1.58
2^{10}	6.59×10^4	4.88×10^4	1.35	4.12×10^4	1.60
2^{11}	1.47×10^5	1.15×10^5	1.28	1.01×10^5	1.46
2^{12}	3.39×10^5	2.70×10^5	1.26	2.18×10^5	1.55
2^{13}	7.52×10^5	6.15×10^5	1.22	4.78×10^5	1.57
2^{14}	2.27×10^6	1.99×10^6	1.14	1.51×10^6	1.50
2^{15}	6.10×10^6	5.31×10^6	1.15	4.68×10^6	1.30
2^{16}	1.67×10^7	1.55×10^7	1.08	1.20×10^7	1.39
2^{17}	3.35×10^7	3.27×10^7	1.03	2.66×10^7	1.26
2^{18}	7.07×10^7	8.64×10^7	0.82	5.79×10^7	1.22
2^{19}	1.55×10^8	1.80×10^8	0.86	1.23×10^8	1.26
2^{20}	3.72×10^8	3.90×10^8	0.95	2.51×10^8	1.48

Table 4.6: Runtime results (in cycles), $n = 2^k$.

SIMD Usage

The FPU version of FFTW uses only the FPU for all floating-point calculations. Due to the SIMD-vectorization of the internally vectorized version and the externally vectorized version and the structure of FFTW not all of the flops can be done in the SIMD-FPU.

Externally Vectorized Version. The first recursion step in the executor is done in the FPU. In all loops that repetitions are not multiples of 4 some FPU codelets have to be used.

Internally Vectorized Codelets. If the number of repetitions of the inner loop of a twiddle codelet is not a multiple of 4 some FPU codelets have to be used.

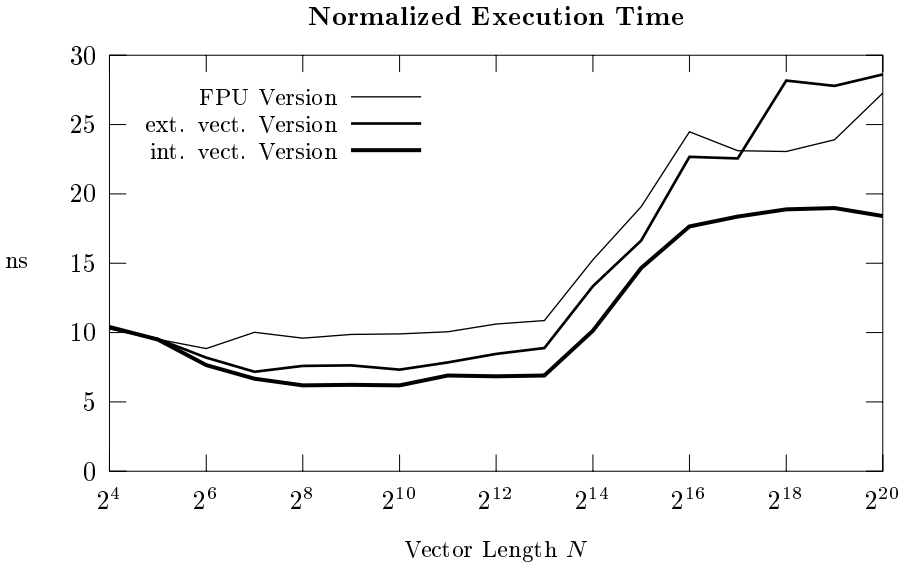

Figure 4.5: Normalized runtime ($T/N \log_2 N$) of FFTW, $n = 2^k$.

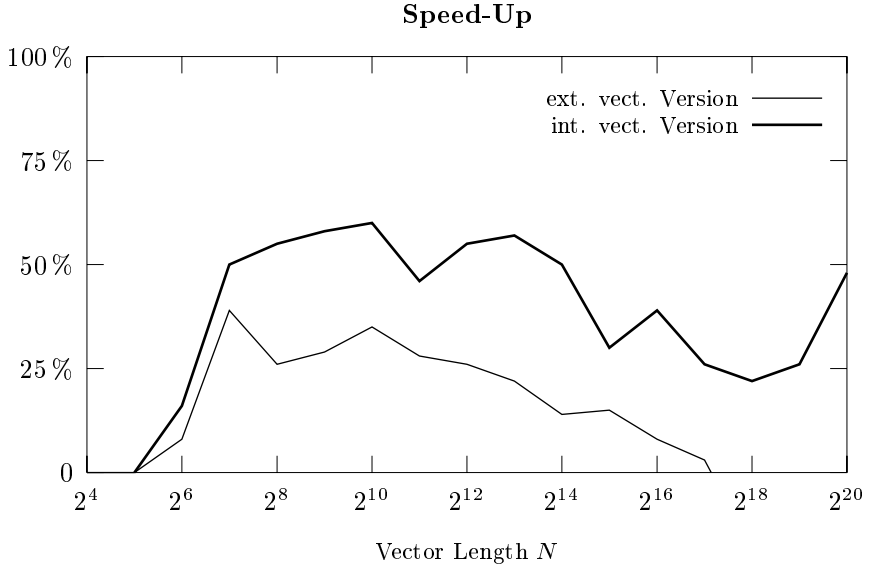

Figure 4.6: Speed-up compared to FFTW FPU, $n = 2^k$.

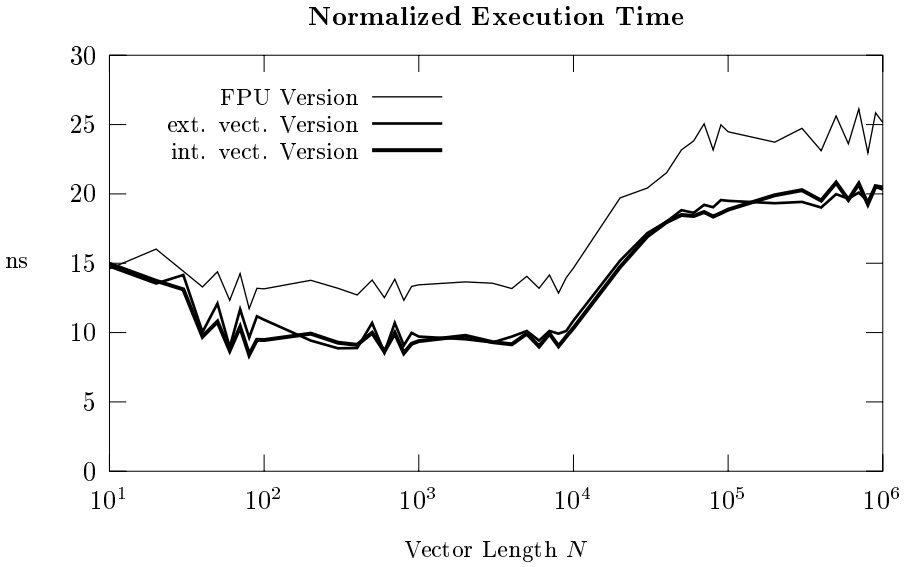

Figure 4.7: Normalized runtime ($T/N \log_2 N$) of FFTW, $n = k \cdot 10^d$.

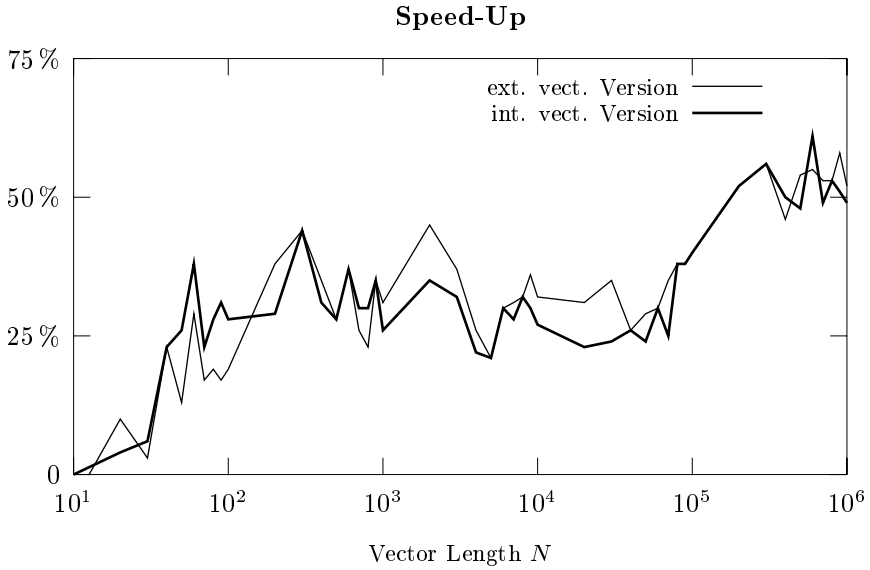

Figure 4.8: Speed-up compared to FFTW FPU, $n = k \cdot 10^d$.

SIMD Usage is defined as the percentage of flops, that are done in the SIMD-FPU.

$$\text{SIMD\_usage} := \frac{4 \times \#\text{SIMD\_flops}}{4 \times \#\text{SIMD\_flops} + \#\text{FPU\_flops}}$$

as one vector-operation in the SIMD-FPU involves 4 flops.

In Table 4.8 the SIMD usage results for transform sizes of $n = 2^k$ are compared and in Table 4.9 the SIMD Usage Results for transform sizes of $n = k \cdot 10^d$ are compared.

It turns out, that the internally vectorized version offers a very good SIMD usage.

Floating-Point Performance Results

In this section, the floating-point performance of the tested routines are compared. FFTW FPU reaches about 15 % of the peak performance. That means, the FPU is runs at 60 % of its peak performance as no SIMD instructions are used. The highest floating-point performance is reached by the internally vectorized version with 22 %. This means the machine runs at 572 Mflop/s. See Table 4.10 and Figure 4.9 for vectorlengths $n = 2^k$ and Table 4.11 and Figure 4.10 for vectorlengths $n = k \cdot 10^d$.

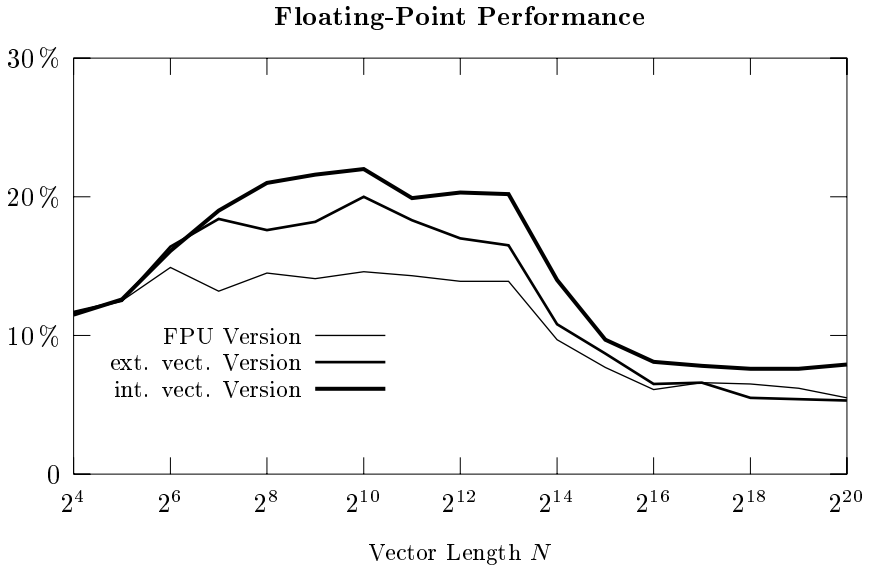

Figure 4.9: Relative floating-point performance, $n = 2^k$ (100 % = 2.6 Gflop/s).

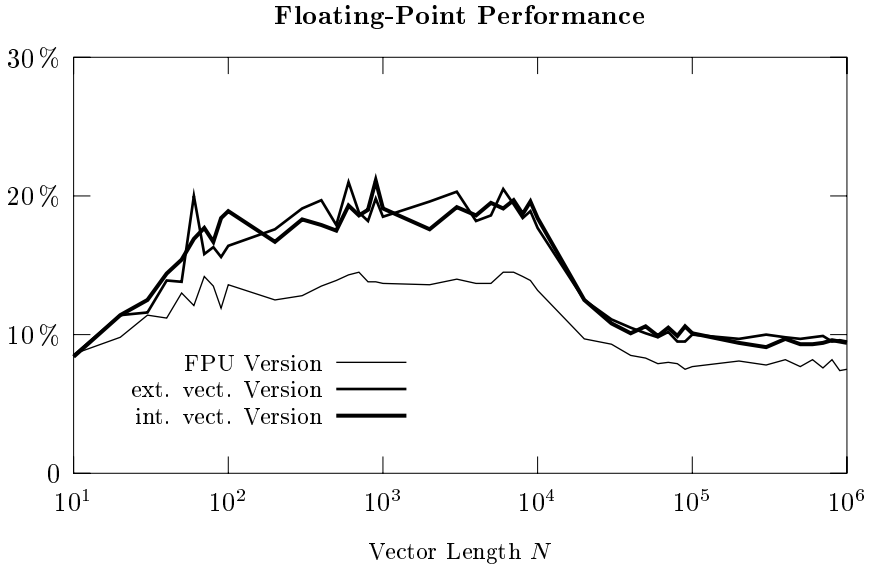

Figure 4.10: Relative floating-point performance, $n = k \cdot 10^d$ (100 % = 2.6 Gflop/s).

size	FPU version	externally vectorized version		internally vectorized version	
	cycles	cycles	speed-up	cycles	speed-up
10	3.15×10^2	3.19×10^2	0.99	3.23×10^2	0.98
20	9.00×10^2	7.61×10^2	1.18	7.72×10^2	1.17
30	1.38×10^3	1.35×10^3	1.02	1.26×10^3	1.10
40	1.84×10^3	1.39×10^3	1.32	1.34×10^3	1.37
50	2.64×10^3	2.22×10^3	1.19	1.98×10^3	1.33
60	2.84×10^3	2.06×10^3	1.38	2.01×10^3	1.41
70	3.97×10^3	3.26×10^3	1.22	2.90×10^3	1.37
80	3.86×10^3	3.17×10^3	1.22	2.76×10^3	1.40
90	5.01×10^3	4.24×10^3	1.18	3.60×10^3	1.39
100	5.69×10^3	4.72×10^3	1.20	4.09×10^3	1.39
200	1.37×10^4	9.37×10^3	1.46	9.87×10^3	1.39
300	2.12×10^4	1.42×10^4	1.49	1.49×10^4	1.42
400	2.86×10^4	2.00×10^4	1.43	2.05×10^4	1.39
500	4.01×10^4	3.12×10^4	1.29	2.91×10^4	1.38
600	4.51×10^4	3.08×10^4	1.46	3.11×10^4	1.45
700	5.96×10^4	4.60×10^4	1.29	4.28×10^4	1.39
800	6.18×10^4	4.55×10^4	1.36	4.27×10^4	1.45
900	7.65×10^4	5.73×10^4	1.33	5.28×10^4	1.45
1,000	8.71×10^4	6.29×10^4	1.39	6.07×10^4	1.43
2,000	1.95×10^5	1.36×10^5	1.44	1.39×10^5	1.40
3,000	3.05×10^5	2.09×10^5	1.46	2.10×10^5	1.46
4,000	4.10×10^5	3.02×10^5	1.36	2.85×10^5	1.44
5,000	5.61×10^5	4.04×10^5	1.39	3.96×10^5	1.42
6,000	6.46×10^5	4.61×10^5	1.40	4.41×10^5	1.47
7,000	8.22×10^5	5.88×10^5	1.40	5.77×10^5	1.43
8,000	8.65×10^5	6.69×10^5	1.29	6.08×10^5	1.42
9,000	1.07×10^6	7.77×10^5	1.38	7.47×10^5	1.44
10,000	1.27×10^6	9.38×10^5	1.35	8.92×10^5	1.42
20,000	3.66×10^6	2.82×10^6	1.30	2.74×10^6	1.34
30,000	5.93×10^6	4.98×10^6	1.19	4.91×10^6	1.21
40,000	8.56×10^6	7.15×10^6	1.20	7.14×10^6	1.20
50,000	1.18×10^7	9.55×10^6	1.23	9.38×10^6	1.25
60,000	1.47×10^7	1.15×10^7	1.28	1.14×10^7	1.29
70,000	1.83×10^7	1.41×10^7	1.30	1.37×10^7	1.34
80,000	1.96×10^7	1.61×10^7	1.22	1.56×10^7	1.26
90,000	2.41×10^7	1.88×10^7	1.28	1.79×10^7	1.34
100,000	2.64×10^7	2.11×10^7	1.26	2.04×10^7	1.30
200,000	5.43×10^7	4.42×10^7	1.23	4.55×10^7	1.19
300,000	8.78×10^7	6.89×10^7	1.27	7.19×10^7	1.22
400,000	1.12×10^8	9.20×10^7	1.21	9.44×10^7	1.18
500,000	1.58×10^8	1.23×10^8	1.28	1.28×10^8	1.23
600,000	1.77×10^8	1.47×10^8	1.20	1.47×10^8	1.21
700,000	2.31×10^8	1.78×10^8	1.30	1.83×10^8	1.26
800,000	2.34×10^8	1.98×10^8	1.18	1.96×10^8	1.19
900,000	2.99×10^8	2.37×10^8	1.26	2.38×10^8	1.26
1,000,000	3.26×10^8	2.63×10^8	1.24	2.65×10^8	1.23

Table 4.7: Runtime results (in cycles), $n = k \cdot 10^d$.

size	FPU Version	externally vectorized version		internally vectorized version	
	flops	flops	SIMD usage	flops	SIMD usage
2^4	2.00×10^2	2.00×10^2	32 %	2.00×10^2	100 %
2^5	4.96×10^3	4.96×10^3	45 %	4.96×10^3	100 %
2^6	1.31×10^3	1.34×10^3	60 %	1.23×10^3	100 %
2^7	3.07×10^3	3.07×10^3	65 %	2.96×10^3	100 %
2^8	7.42×10^4	7.10×10^4	69 %	6.93×10^4	100 %
2^9	1.66×10^4	1.66×10^4	74 %	1.61×10^4	100 %
2^{10}	3.84×10^5	3.89×10^5	78 %	3.62×10^5	100 %
2^{11}	8.40×10^5	8.40×10^5	79 %	8.05×10^5	100 %
2^{12}	1.88×10^5	1.83×10^5	81 %	1.77×10^5	100 %
2^{13}	4.18×10^6	4.06×10^6	83 %	3.86×10^6	100 %
2^{14}	8.79×10^6	8.56×10^6	77 %	8.47×10^6	100 %
2^{15}	1.87×10^6	1.86×10^6	78 %	1.82×10^6	100 %
2^{16}	4.07×10^7	4.05×10^7	80 %	3.89×10^7	100 %
2^{17}	8.80×10^7	8.60×10^7	87 %	8.29×10^7	100 %
2^{18}	1.85×10^7	1.89×10^7	88 %	1.76×10^7	100 %
2^{19}	3.84×10^8	3.87×10^8	83 %	3.73×10^8	100 %
2^{20}	8.17×10^8	8.26×10^8	84 %	7.94×10^8	100 %

Table 4.8: SIMD usage, $n = 2^k$.

size	FPU version	externally vectorized version		internally vectorized version	
	flops	flops	SIMD usage	flops	SIMD usage
10	1.08×10^2	1.08×10^2	0 %	1.08×10^2	0 %
20	3.52×10^3	3.46×10^3	51 %	3.52×10^3	95 %
30	6.28×10^3	6.28×10^3	28 %	6.28×10^3	71 %
40	8.24×10^3	7.72×10^3	56 %	7.72×10^3	91 %
50	1.37×10^3	1.22×10^3	35 %	1.22×10^3	80 %
60	1.38×10^3	1.65×10^3	69 %	1.36×10^3	92 %
70	2.26×10^3	2.05×10^3	43 %	2.05×10^3	83 %
80	2.09×10^3	2.06×10^3	67 %	1.84×10^3	89 %
90	2.38×10^3	2.65×10^3	33 %	2.65×10^3	83 %
100	3.09×10^3	3.09×10^3	72 %	3.09×10^3	99 %
200	6.84×10^4	6.58×10^4	74 %	6.58×10^4	99 %
300	1.09×10^4	1.09×10^4	77 %	1.09×10^4	99 %
400	1.54×10^4	1.58×10^4	78 %	1.47×10^4	99 %
500	2.23×10^4	2.23×10^4	81 %	2.03×10^4	86 %
600	2.58×10^4	2.58×10^4	80 %	2.40×10^4	99 %
700	3.46×10^5	3.46×10^5	83 %	3.19×10^5	86 %
800	3.42×10^5	3.31×10^5	79 %	3.24×10^5	98 %
900	4.22×10^5	4.53×10^5	83 %	4.46×10^5	100 %
1,000	4.78×10^5	4.65×10^5	82 %	4.65×10^5	100 %
2,000	1.06×10^5	1.06×10^5	84 %	9.82×10^5	100 %
3,000	1.71×10^5	1.70×10^5	85 %	1.61×10^5	100 %
4,000	2.25×10^5	2.20×10^5	85 %	2.11×10^5	100 %
5,000	3.07×10^5	3.01×10^5	86 %	3.09×10^5	100 %
6,000	3.74×10^6	3.79×10^6	87 %	3.37×10^6	100 %
7,000	4.76×10^6	4.55×10^6	87 %	4.55×10^6	100 %
8,000	4.92×10^6	4.92×10^6	86 %	4.54×10^6	100 %
9,000	5.98×10^6	5.87×10^6	87 %	5.87×10^6	100 %
10,000	6.66×10^6	6.66×10^6	87 %	6.57×10^6	100 %
20,000	1.41×10^6	1.41×10^6	83 %	1.37×10^6	100 %
30,000	2.20×10^6	2.22×10^6	83 %	2.13×10^6	100 %
40,000	2.90×10^6	3.00×10^6	89 %	2.89×10^6	100 %
50,000	3.88×10^7	3.87×10^7	84 %	3.97×10^7	100 %
60,000	4.68×10^7	4.54×10^7	89 %	4.50×10^7	100 %
70,000	5.90×10^7	5.77×10^7	85 %	5.77×10^7	100 %
80,000	6.20×10^7	6.10×10^7	89 %	6.15×10^7	100 %
90,000	7.24×10^7	7.16×10^7	85 %	7.60×10^7	100 %
100,000	8.13×10^7	8.40×10^7	90 %	8.19×10^7	100 %
200,000	1.76×10^7	1.72×10^7	90 %	1.72×10^7	100 %
300,000	2.75×10^7	2.77×10^7	91 %	2.62×10^7	100 %
400,000	3.69×10^8	3.59×10^8	91 %	3.66×10^8	100 %
500,000	4.84×10^8	4.79×10^8	91 %	4.78×10^8	100 %
600,000	5.79×10^8	5.77×10^8	91 %	5.47×10^8	100 %
700,000	7.03×10^8	7.05×10^8	92 %	6.90×10^8	100 %
800,000	7.66×10^8	7.56×10^8	91 %	7.53×10^8	100 %
900,000	8.91×10^8	9.08×10^8	92 %	9.05×10^8	100 %
1,000,000	9.75×10^8	9.96×10^8	91 %	9.95×10^8	100 %

Table 4.9: SIMD usage, $n = k \cdot 10^d$.

size	FPU version	externally vectorized version	internally vectorized version
2^4	11.7 %	11.7 %	11.5 %
2^5	12.5 %	12.5 %	12.6 %
2^6	14.9 %	16.4 %	16.1 %
2^7	13.2 %	18.4 %	19.0 %
2^8	14.5 %	17.6 %	21.0 %
2^9	14.1 %	18.2 %	21.6 %
2^{10}	14.6 %	20.0 %	22.0 %
2^{11}	14.3 %	18.3 %	19.9 %
2^{12}	13.9 %	17.0 %	20.3 %
2^{13}	13.9 %	16.5 %	20.2 %
2^{14}	9.7 %	10.8 %	14.0 %
2^{15}	7.7 %	8.7 %	9.7 %
2^{16}	6.1 %	6.5 %	8.1 %
2^{17}	6.6 %	6.6 %	7.8 %
2^{18}	6.5 %	5.5 %	7.6 %
2^{19}	6.2 %	5.4 %	7.6 %
2^{20}	5.5 %	5.3 %	7.9 %

Table 4.10: Relative floating-point performance, $n = 2^k$ (100 % = 2.6 Gflop/s).

size	FPU version	externally vectorized version	internally vectorized version
10	8.6 %	8.5 %	8.4 %
20	9.8 %	11.4 %	11.4 %
30	11.4 %	11.6 %	12.5 %
40	11.2 %	13.9 %	14.4 %
50	13.0 %	13.8 %	15.4 %
60	12.1 %	20.0 %	16.9 %
70	14.2 %	15.8 %	17.7 %
80	13.5 %	16.3 %	16.7 %
90	11.9 %	15.6 %	18.4 %
100	13.6 %	16.4 %	18.9 %
200	12.5 %	17.6 %	16.7 %
300	12.8 %	19.1 %	18.3 %
400	13.5 %	19.7 %	17.9 %
500	13.9 %	17.9 %	17.5 %
600	14.3 %	21.0 %	19.3 %
700	14.5 %	18.8 %	18.6 %
800	13.8 %	18.2 %	19.0 %
900	13.8 %	19.8 %	21.1 %
1,000	13.7 %	18.5 %	19.1 %
2,000	13.6 %	19.6 %	17.6 %
3,000	14.0 %	20.3 %	19.2 %
4,000	13.7 %	18.2 %	18.6 %
5,000	13.7 %	18.6 %	19.5 %
6,000	14.5 %	20.5 %	19.1 %
7,000	14.5 %	19.4 %	19.7 %
8,000	14.2 %	18.4 %	18.7 %
9,000	13.9 %	18.9 %	19.6 %
10,000	13.2 %	17.7 %	18.4 %
20,000	9.7 %	12.5 %	12.5 %
30,000	9.3 %	11.1 %	10.8 %
40,000	8.5 %	10.5 %	10.1 %
50,000	8.3 %	10.1 %	10.6 %
60,000	7.9 %	9.8 %	9.9 %
70,000	8.0 %	10.2 %	10.5 %
80,000	7.9 %	9.5 %	9.9 %
90,000	7.5 %	9.5 %	10.6 %
100,000	7.7 %	10.0 %	10.1 %
200,000	8.1 %	9.7 %	9.4 %
300,000	7.8 %	10.0 %	9.1 %
400,000	8.2 %	9.8 %	9.7 %
500,000	7.7 %	9.7 %	9.3 %
600,000	8.2 %	9.8 %	9.3 %
700,000	7.6 %	9.9 %	9.4 %
800,000	8.2 %	9.5 %	9.6 %
900,000	7.4 %	9.6 %	9.5 %
1,000,000	7.5 %	9.5 %	9.4 %

Table 4.11: Relative floating-point performance, $n = k \cdot 10^d$ (100 % = 2.6 Gflop/s).

Chapter 5

Results on the Motorola G4 AltiVec

All experiments described in this chapter were carried out on the following system:

CPU	Motorola 7400 G4 400 MHz 1024 kB unified L2 Cache 3.2 Gflop/s
RAM	128 MB
Operating System	Yellodog Linux 1.2
Compiler	GNU gcc-vec 2.9.5

5.1 Results on FFTW

5.1.1 Codelet Performance

On the Motorola G4 the same measurement methods have been used as on the Intel Pentium III, described in Section 4.2.1

No Twiddle Codelets

In this section always the runtime of 4 calls to the FPU no twiddle codelets is compared to the runtime of one call to the corresponding SIMD no twiddle codelet. The SIMD no twiddle codelet is up to 1.81 times faster than the FPU version (see Table 5.1 and Figure 5.1).

Externally Vectorized Twiddle Codelets

In this section always the runtime of 4 calls to the FPU twiddle codelet is compared to the runtime of one call to the corresponding externally vectorized SIMD twiddle codelet. The externally vectorized SIMD twiddle codelet is between 1.16 and 2.43 times faster than the FPU version (see Table 5.2 and Figure 5.2).

size	cycles FPU codelet	cycles SIMD codelet	speed-up
1	69	64	1.08
2	130	120	1.09
3	250	207	1.21
4	265	250	1.06
5	514	351	1.46
6	608	406	1.50
7	1,083	619	1.75
8	749	541	1.38
9	1,334	737	1.81
10	1,469	819	1.79
11	2,874	1,673	1.72
12	1,513	905	1.67
13	2,856	1,808	1.58
14	2,785	1,765	1.58
15	2,750	1,521	1.81
16	2,246	1,290	1.74
32	5,533	3,162	1.75
64	12,563	7,405	1.70

Table 5.1: Runtime (in cycles) of no twiddle codelets.

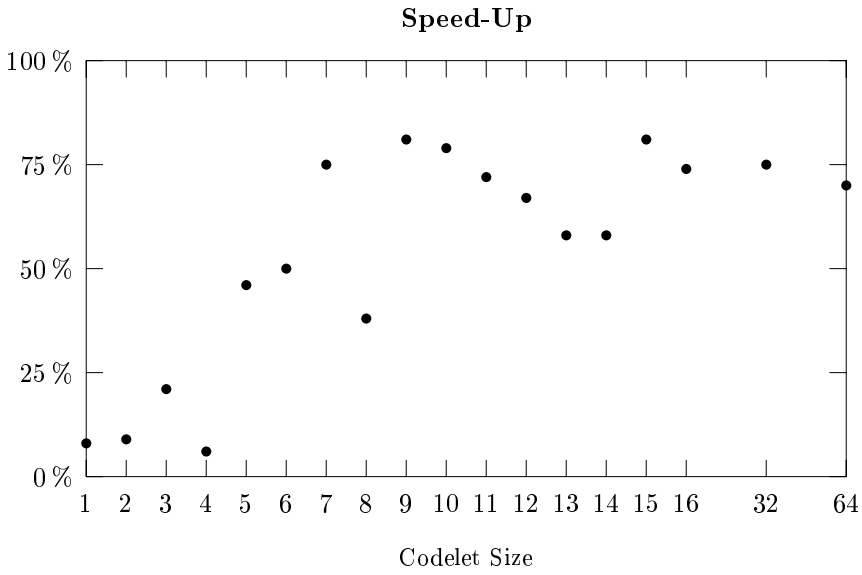

Figure 5.1: No twiddle codelets speed-up

Internally Vectorized Twiddle Codelets

In this section always the runtime of one call to the FPU twiddle codelet is compared to the runtime of one call to the corresponding internally vectorized

size	FPU codelet cycles	SIMD codelet cycles	speed-up
2	608	523	1.16
3	1,599	866	1.85
4	1,830	1,082	1.69
5	3,393	1,606	2.11
6	3,814	1,895	2.01
7	6,352	2,694	2.36
8	4,950	2,662	1.86
9	8,658	3,557	2.43
10	8,151	4,524	1.80
16	13,832	7,613	1.82
32	33,155	15,974	2.08
64	76,710	38,106	2.01

Table 5.2: Runtime (in cycles) of externally vectorized twiddle codelets.

SIMD twiddle codelet. The internally vectorized SIMD twiddle codelet is up to 1.95 times faster than the FPU version (see Table 5.3 and Figure 5.2).

size	cycles FPU codelet	cycles SIMD codelet	speed-up
2	148	179	0.83
3	394	300	1.31
4	452	369	1.23
5	845	546	1.55
6	952	647	1.47
7	1,583	892	1.78
8	1,227	842	1.46
9	2,165	1,112	1.95
10	2,028	1,365	1.49
16	3,453	2,163	1.60
32	8,278	4,410	1.88
64	19,136	11,398	1.68

Table 5.3: Runtime (in cycles) of internally vectorized twiddle codelets.

5.1.2 FFTW Performance Results

In this section 3 variants of FFTW are compared. All of them are single precision versions.

FPU Version: This is the standard FFTW single precision version.

Externally Vectorized Version: This is the SIMD-vectorized version, that uses the externally vectorized twiddle codelets.

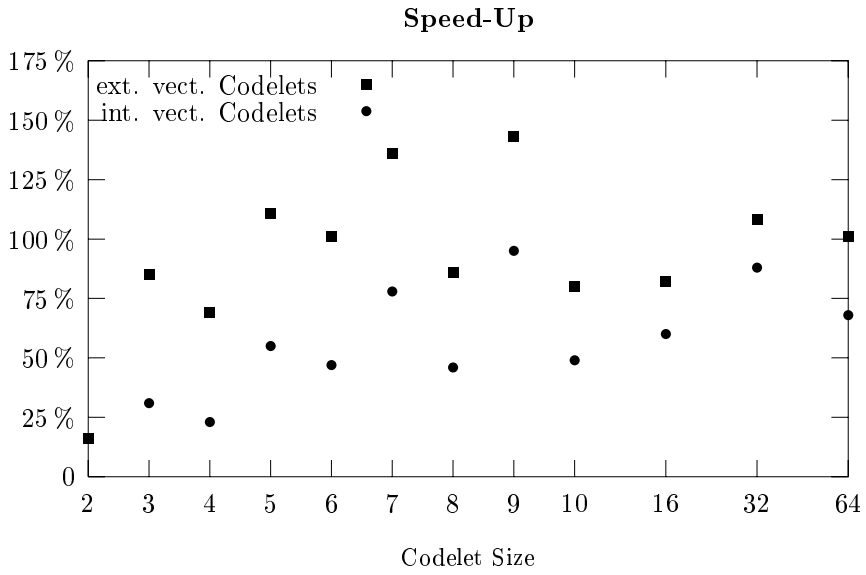

Figure 5.2: Twiddle codelets speed-up.

Internally Vectorized Version: This is the SIMD-vectorized version, that uses the internally vectorized twiddle codelets.

Runtime Results

In Table 5.4 and Figure 5.3 the runtimes for transform sizes of $n = 2^k$ are compared. In Table 5.5 and Figure 5.5 the runtimes for transform sizes of $n = k \cdot 10^d$ are compared. One can see, that the internally vectorized version is about a factor of 1.3 to 1.45 faster than the FPU version for the in-cache case ($n < 10,000$) and about 1.5 to 1.76 times faster for the out-of-cache case. For very big powers of 2, the speed-up increases. See Figure 5.4 and Figure 5.6 for details.

The speed-up for the externally vectorized version is usually less than the speed-up for the internally vectorized version for the power-of-2 case. This is due to cache associativity problems for large power-of-2 strides and a lower SIMD utilization.

For transform sizes of $n = 2^k$ a better speed-up is achieved than for transform sizes of $n = k \cdot 10^d$. The difference between the externally vectorized version and the internally vectorized version is minimal for the case $n = k \cdot 10^d$.

Floating-Point Performance Results

In this section, the floating-point performance of the tested routines are compared. FFTW FPU reaches about 9 % of the peak performance. That means, the FPU is runs at 36 % of its peak performance as no SIMD instructions are used. The

size	FPU version	externally vectorized version		internally vectorized version	
	cycles	cycles	speed-up	cycles	speed-up
2^4	4.27×10^3	4.27×10^3	1.00	4.27×10^3	1.00
2^5	9.46×10^3	9.15×10^3	1.03	9.15×10^3	1.03
2^6	2.01×10^3	1.83×10^3	1.10	1.71×10^3	1.18
2^7	4.88×10^4	3.91×10^4	1.25	3.54×10^4	1.38
2^8	1.07×10^4	8.30×10^4	1.29	7.57×10^4	1.42
2^9	2.44×10^4	1.95×10^4	1.25	1.61×10^4	1.52
2^{10}	5.47×10^5	4.30×10^5	1.27	4.10×10^5	1.33
2^{11}	1.33×10^5	1.02×10^5	1.31	9.77×10^5	1.36
2^{12}	3.05×10^5	2.58×10^5	1.18	2.34×10^5	1.30
2^{13}	7.19×10^6	5.78×10^6	1.24	5.47×10^6	1.31
2^{14}	1.63×10^6	1.44×10^6	1.13	1.25×10^6	1.30
2^{15}	4.00×10^7	3.62×10^7	1.10	3.00×10^6	1.33
2^{16}	1.00×10^7	9.00×10^7	1.11	7.50×10^7	1.33
2^{17}	3.10×10^7	2.80×10^7	1.11	2.00×10^7	1.55
2^{18}	7.80×10^8	6.20×10^8	1.26	4.40×10^8	1.77
2^{19}	1.76×10^8	1.52×10^8	1.16	1.00×10^8	1.76
2^{20}	3.80×10^9	3.40×10^9	1.12	2.16×10^8	1.76

Table 5.4: Runtime results (in cycles), $n = 2^k$.

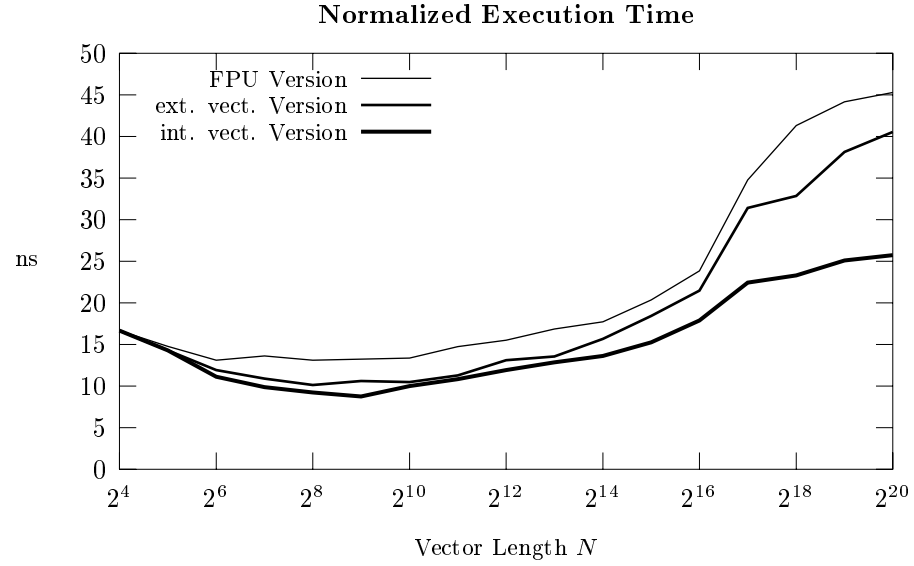

Figure 5.3: Normalized runtime ($T/N \log_2 N$) of FFTW, $n = 2^k$.

size	FPU version	externally vectorized version		internally vectorized version	
	cycles	cycles	speed-up	cycles	speed-up
10	3.05×10^2	3.20×10^3	0.95	3.05×10^2	1.00
20	7.01×10^3	6.40×10^3	1.10	6.71×10^3	1.04
30	1.13×10^3	1.10×10^3	1.03	1.07×10^3	1.06
40	1.46×10^3	1.19×10^3	1.23	1.19×10^3	1.23
50	2.08×10^3	1.83×10^3	1.13	1.65×10^3	1.26
60	2.44×10^3	1.89×10^3	1.29	1.77×10^3	1.38
70	3.30×10^4	2.81×10^3	1.17	2.69×10^3	1.23
80	3.05×10^3	2.56×10^3	1.19	2.38×10^3	1.28
90	4.15×10^4	3.54×10^4	1.17	3.17×10^4	1.31
100	4.52×10^4	3.78×10^4	1.19	3.54×10^4	1.28
200	1.07×10^4	7.81×10^4	1.38	8.30×10^4	1.29
300	1.76×10^4	1.22×10^4	1.44	1.22×10^4	1.44
400	2.25×10^4	1.66×10^4	1.35	1.71×10^4	1.31
500	3.13×10^4	2.44×10^4	1.28	2.44×10^4	1.28
600	3.61×10^5	2.64×10^4	1.37	2.64×10^4	1.37
700	4.69×10^5	3.71×10^5	1.26	3.61×10^5	1.30
800	4.69×10^5	3.81×10^5	1.23	3.61×10^5	1.30
900	6.05×10^5	4.49×10^5	1.35	4.49×10^5	1.35
1,000	6.64×10^5	5.08×10^5	1.31	5.27×10^5	1.26
2,000	1.64×10^5	1.13×10^5	1.45	1.21×10^5	1.35
3,000	2.58×10^5	1.88×10^5	1.37	1.95×10^5	1.32
4,000	3.44×10^6	2.73×10^5	1.26	2.81×10^5	1.22
5,000	4.53×10^6	3.75×10^6	1.21	3.75×10^6	1.21
6,000	5.47×10^6	4.22×10^6	1.30	4.22×10^6	1.30
7,000	7.19×10^6	5.47×10^6	1.31	5.63×10^6	1.28
8,000	7.81×10^6	5.94×10^6	1.32	5.94×10^6	1.32
9,000	9.38×10^6	6.88×10^6	1.36	7.19×10^6	1.30
10,000	1.03×10^6	7.81×10^6	1.32	8.13×10^6	1.27
20,000	2.38×10^6	1.81×10^6	1.31	1.94×10^6	1.23
30,000	3.88×10^7	2.88×10^6	1.35	3.13×10^6	1.24
40,000	5.50×10^7	4.38×10^7	1.26	4.38×10^7	1.26
50,000	7.75×10^7	6.00×10^7	1.29	6.25×10^7	1.24
60,000	9.75×10^7	7.50×10^7	1.30	7.50×10^7	1.30
70,000	1.25×10^7	9.25×10^7	1.35	1.00×10^7	1.25
80,000	1.45×10^7	1.05×10^7	1.38	1.05×10^7	1.38
90,000	1.80×10^7	1.30×10^7	1.38	1.30×10^7	1.38
100,000	2.10×10^7	1.50×10^7	1.40	1.50×10^7	1.40
200,000	5.00×10^8	3.30×10^8	1.52	3.30×10^8	1.52
300,000	8.40×10^8	5.40×10^8	1.56	5.40×10^8	1.56
400,000	1.08×10^8	7.40×10^8	1.46	7.20×10^8	1.50
500,000	1.48×10^8	9.60×10^8	1.54	1.00×10^8	1.48
600,000	1.80×10^8	1.16×10^8	1.55	1.12×10^8	1.61
700,000	2.20×10^8	1.44×10^8	1.53	1.48×10^8	1.49
800,000	2.32×10^8	1.52×10^8	1.53	1.52×10^8	1.53
900,000	2.84×10^8	1.80×10^8	1.58	1.88×10^8	1.51
1,000,000	3.04×10^8	2.00×10^8	1.52	2.04×10^8	1.49

Table 5.5: Runtime results (in cycles), $n = k \cdot 10^d$.

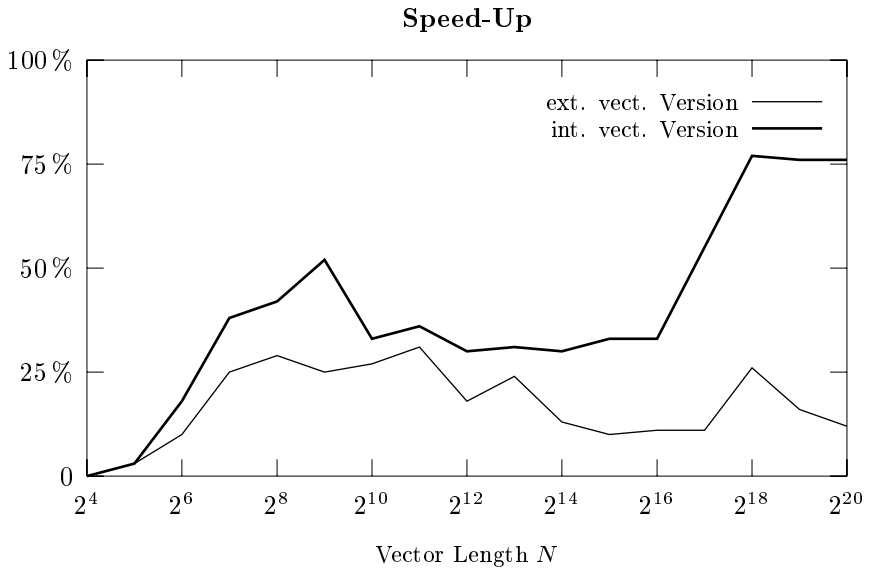

Figure 5.4: Speed-up compared to FFTW FPU, $n = 2^k$

highest floating-point performance is reached by the internally vectorized version with 12.5 %. This means the machine runs at 400 Mflop/s. See Table 5.6 and Figure 5.7 for vectorlengths $n = 2^k$ and Table 5.7 and Figure 5.8 for vectorlengths $n = k \cdot 10^d$.

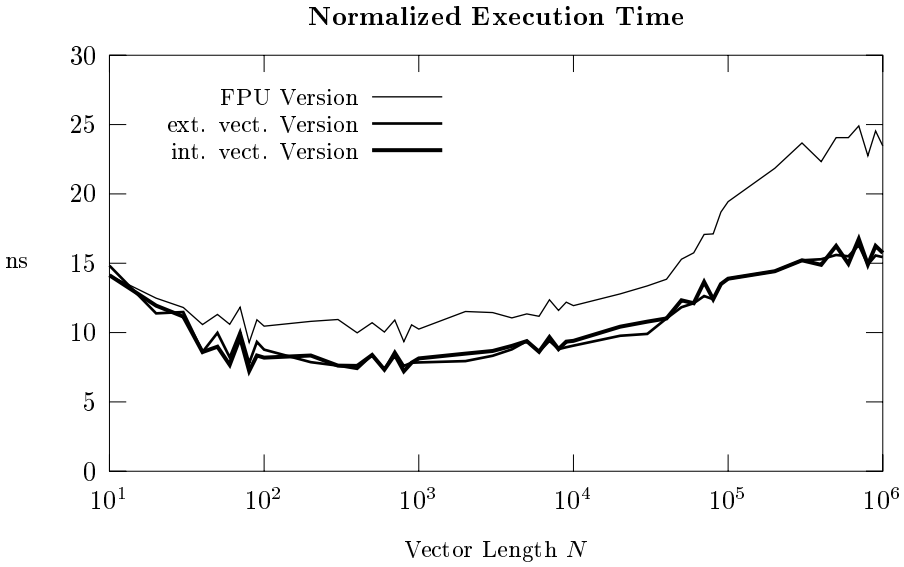

Figure 5.5: Normalized runtime ($T/N \log_2 N$) of FFTW, $n = k \cdot 10^d$.

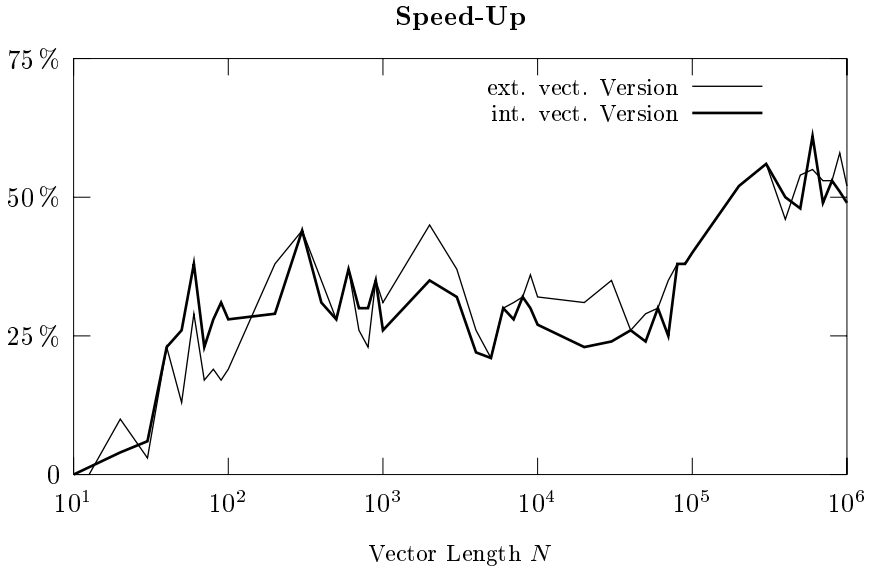

Figure 5.6: Speed-up compared to FFTW FPU, $n = k \cdot 10^d$

size	FPU version	externally vectorized version	internally vectorized version
2^4	5.9 %	5.9 %	5.9 %
2^5	6.6 %	6.8 %	6.8 %
2^6	8.1 %	9.2 %	9.0 %
2^7	7.9 %	9.8 %	10.5 %
2^8	8.6 %	10.7 %	11.4 %
2^9	8.5 %	10.6 %	12.5 %
2^{10}	8.8 %	11.3 %	11.0 %
2^{11}	7.9 %	10.3 %	10.3 %
2^{12}	7.7 %	8.9 %	9.4 %
2^{13}	7.3 %	8.8 %	8.8 %
2^{14}	6.8 %	7.4 %	8.5 %
2^{15}	5.8 %	6.4 %	7.6 %
2^{16}	5.1 %	5.6 %	6.5 %
2^{17}	3.5 %	3.8 %	5.2 %
2^{18}	3.0 %	3.8 %	5.0 %
2^{19}	2.7 %	3.2 %	4.7 %
2^{20}	2.7 %	3.0 %	4.6 %

Table 5.6: Relative floating-point performance, $n = 2^k$ (100 % = 3.2 Gflop/s).

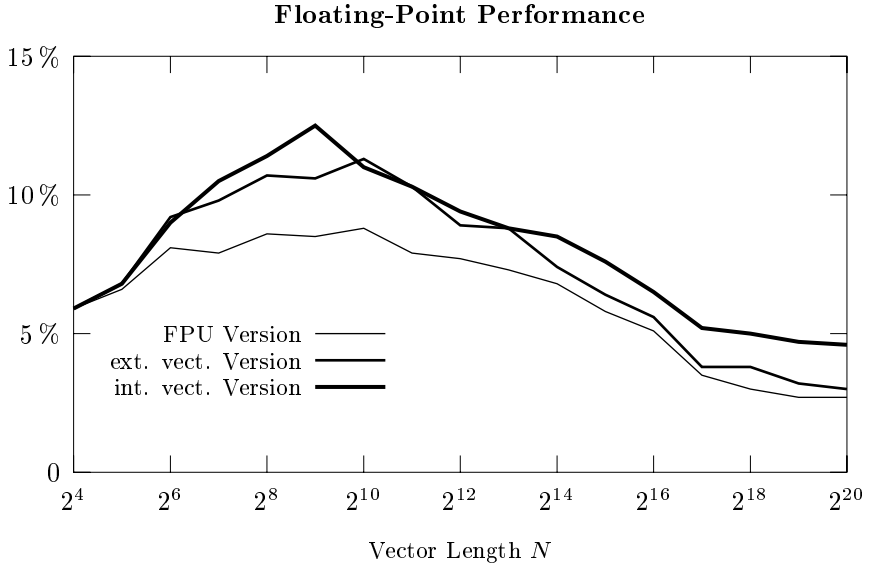

Figure 5.7: Relative floating-point performance, $n = 2^k$ (100 % = 3.2 Gflop/s).

size	FPU version	externally vectorized version	internally vectorized version
10	4.4 %	4.2 %	4.4 %
20	6.3 %	6.8 %	6.6 %
30	7.0 %	7.1 %	7.4 %
40	7.0 %	8.1 %	8.1 %
50	8.3 %	8.3 %	9.3 %
60	7.0 %	10.9 %	9.6 %
70	8.6 %	9.1 %	9.6 %
80	8.6 %	10.1 %	9.7 %
90	7.2 %	9.4 %	10.4 %
100	8.6 %	10.2 %	10.9 %
200	8.0 %	10.5 %	9.9 %
300	7.7 %	11.1 %	11.1 %
400	8.6 %	11.9 %	10.7 %
500	8.9 %	11.4 %	10.4 %
600	8.9 %	12.2 %	11.4 %
700	9.2 %	11.7 %	11.0 %
800	9.1 %	10.9 %	11.2 %
900	8.7 %	12.6 %	12.4 %
1,000	9.0 %	11.4 %	11.0 %
2,000	8.1 %	11.7 %	10.1 %
3,000	8.3 %	11.3 %	10.3 %
4,000	8.2 %	10.1 %	9.4 %
5,000	8.5 %	10.0 %	10.3 %
6,000	8.5 %	11.2 %	10.0 %
7,000	8.3 %	10.4 %	10.1 %
8,000	7.9 %	10.4 %	9.6 %
9,000	8.0 %	10.7 %	10.2 %
10,000	8.1 %	10.7 %	10.1 %
20,000	7.4 %	9.7 %	8.8 %
30,000	7.1 %	9.7 %	8.5 %
40,000	6.6 %	8.6 %	8.3 %
50,000	6.3 %	8.1 %	7.9 %
60,000	6.0 %	7.6 %	7.5 %
70,000	5.9 %	7.8 %	7.2 %
80,000	5.3 %	7.3 %	7.3 %
90,000	5.0 %	6.9 %	7.3 %
100,000	4.8 %	7.0 %	6.8 %
200,000	4.4 %	6.5 %	6.5 %
300,000	4.1 %	6.4 %	6.1 %
400,000	4.3 %	6.1 %	6.4 %
500,000	4.1 %	6.2 %	6.0 %
600,000	4.0 %	6.2 %	6.1 %
700,000	4.0 %	6.1 %	5.8 %
800,000	4.1 %	6.2 %	6.2 %
900,000	3.9 %	6.3 %	6.0 %
1,000,000	4.0 %	6.2 %	6.1 %

Table 5.7: Relative floating-point performance, $n = k \cdot 10^d$ (100 % = 3.2 Gflop/s).

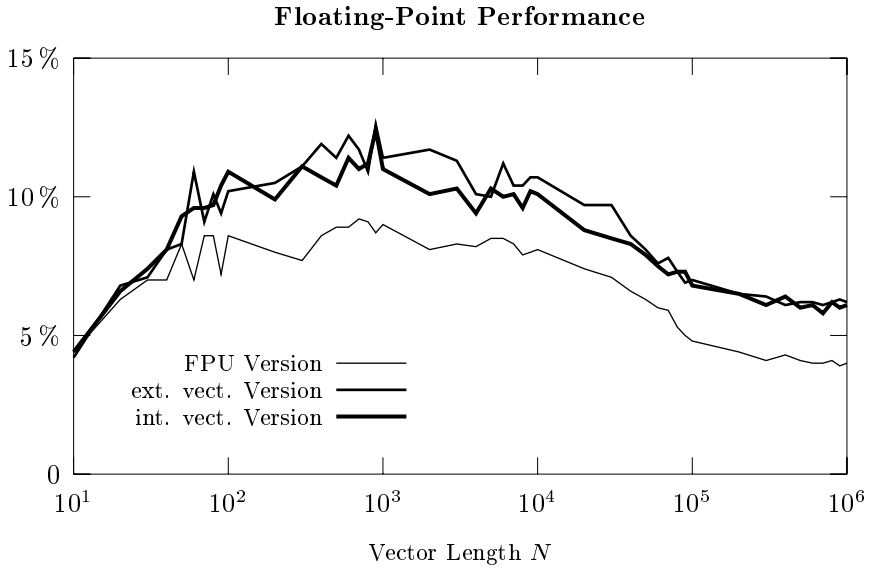

Figure 5.8: Relative floating-point performance, $n = k \cdot 10^d$ (100 % = 3.2 Gflop/s).

Conclusion

Current high-performance desktop computers have been prepared for multi media applications (sound, pictures, videos, and 3D-graphics). Future applications like video on demand, IP telephony and speech recognition will even increase the need of power for these systems.

Many hardware developers have added new extensions to their CPUs (like the Intel SSE and Motorola AltiVec extensions) to boost the performance of their products in this critical area. These extensions introduce the short vector, single instruction, multiple data model (SIMD) into the world of desktop computing. They are meant to speed up multi media applications but they are not portable.

In scientific computing clusters of SMPs are used in a wide range. The upcoming cheaper systems use the processor technology from desktop PCs and small servers. These clusters of SMPs usually contain processors manufactured by Intel or AMD and are run under Linux. Thus, short vector extensions become available for the users of such systems.

To achieve optimal performance, it is important to utilize these features in scientific computing, too. Adaptive scientific software has to overcome the portability problems introduced by the new extensions.

In this thesis FFT algorithms optimized for the Intel Pentium III have been presented and their performance has been demonstrated. Moreover, the highly portable, high performance FFT package FFTW has been accelerated considerably by the use of short vector extensions.

It turned out, that the limitations of the short vector extensions (mainly in the area of memory access and data alignment) prevent the efficient use of a vector unit for carrying out FFTs. Speed-up factors of up to 1.75 (compared to a FPU version) have been achieved. However, in the best case, the CPU runs at only 25 % of its peak performance.

Appendix A

Source Code of Radix-4 Algorithms

```
//      radix4
//
//      SIMD radix4 testkernel
//

//  constants and data

#include <stdlib.h>
#include <math.h>
#include "xmmintrin.h"

#define M_PI    3.141592654d

__m128 *radix4_twiddle[12];
__m128 neglast=_mm_set_ps(-1.0,1.0,1.0,1.0);

//  to compute 4^q
int powers[12]={1,4,16,64,256,1024,4096,16384,65536,262144,1048576,4194304};

static int simd_maxlen=0;
static int vect_maxlen=0;

void simd_radix4_exit(void)
{
    int t;
    for (t=1;t<=simd_maxlen;t++)
        _mm_free(radix4_twiddle[t]);
}

void vect_radix4_exit(void)
{
    int t;
    for (t=1;t<=vect_maxlen;t++)
        _mm_free(radix4_twiddle[t]);
}


//  init the twiddlefactor tables

void simd_radix4_init(int q)
{
    int L,t,j;
    float **mytwiddle=(float **)radix4_twiddle;
```

```
    simd_maxlen=q;

    radix4_twiddle[0]=NULL;
    for (t=1;t<=q;t++)
    {
        radix4_twiddle[t]=(__m128 *)_mm_malloc(sizeof(__m128)*powers[t],16);
    }


    //  twiddlefactors bis n=4^10=2^22
    L=1;
    for (t=1;t<=q;t++)
    {
        L*=4;
        for (j=0;j<L/4;j++)
        {
            //  w^0,...,w^3
            mytwiddle[t][16*j]=1;                   //  w^0=1, __m128[0]
            mytwiddle[t][16*j+1]=0;
            mytwiddle[t][16*j+2]=cos(2*3.141592654*j/L);    //  w
            mytwiddle[t][16*j+3]=-sin(2*3.141592654*j/L);

            mytwiddle[t][16*j+4]=cos(2*3.141592654*j/L*2);//  w^2, __m128[1]
            mytwiddle[t][16*j+5]=-sin(2*3.141592654*j/L*2);
            mytwiddle[t][16*j+6]=cos(2*3.141592654*j/L*3);  //  w^3
            mytwiddle[t][16*j+7]=-sin(2*3.141592654*j/L*3);

            //  shuffled w^0,...,w^3
            mytwiddle[t][16*j+8]=0;         //  shuffled w^0=1, __m128[2]
            mytwiddle[t][16*j+9]=1;
            mytwiddle[t][16*j+10]=-sin(2*3.141592654*j/L);  //  shuffled w
            mytwiddle[t][16*j+11]=cos(2*3.141592654*j/L);
            //  shuffled w^2, __m128[2]
            mytwiddle[t][16*j+12]=-sin(2*3.141592654*j/L*2);
            mytwiddle[t][16*j+13]=cos(2*3.141592654*j/L*2);
            mytwiddle[t][16*j+14]=-sin(2*3.141592654*j/L*3);//  shuffled w^3
            mytwiddle[t][16*j+15]=cos(2*3.141592654*j/L*3);
        }
    }
}


void vect_radix4_init(int q)
{
    int L,t,j,k,j1;
    float **mytwiddle=(float **)radix4_twiddle;

    vect_maxlen=q;

    radix4_twiddle[0]=NULL;
    for (t=1;t<=q;t++)
```

```
    {
        radix4_twiddle[t]=(__m128 *)_mm_malloc(sizeof(__m128)*
          powers[t-1]*6,16);
    }

    //  twiddlefactors bis n=4^10=2^20
    for (t=2;t<=q;t++)
    {
        L=powers[t];
        for (j=0;j<L/16;j++)
            for (k=1;k<=3;k++)
                for (j1=0;j1<4;j1++)
                {
                    mytwiddle[t][j1+j*24+(k-1)*8]=cos(2*3.141592654*
                      (j1+4*j)*k/L);
                    mytwiddle[t][j1+j*24+(k-1)*8+4]=-sin(2*3.141592654*
                      (j1+4*j)*k/L);
                }
    }
}

//  simd radix-4 algorithm

void simd_radix4(__m128 *data, int t)
{
    int q,j,k,n,
        r,rs,L,Ls;
    __m128 x01,x23,x45,x67,
        t0,t1,t2,t3,
        real,imag;
    float *dataptr=(float *)data;

    n=powers[t];
    //  stage q=1   ------------------------------------------------------
    q=1;
    L=2;
    Ls=1;
    r=n>>2;

    //  the j-loop degenerates
    for (k=0;k<r;k++)
    {
        //  load x0=alpha, x1=beta
        x01=data[2*k];
        //  load x2=gamma, x3=delta
        x23=data[2*k+1];

        //  tau0=alpha+gamma, tau1=beta+delta
        t0=_mm_add_ps(x01,x23);
        //  tau2=alpha-gamma, tau3=beta-delta
        t1=_mm_sub_ps(x01,x23);
```

```
    //  reorder to calc tau0+tau1, tau2-itau3,...
    t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));//  extract tau0, tau1
    t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));//  extract tau2, itau3
    t3=_mm_mul_ps(t3,neglast);  // negate highestorder float for (-itau3)

    //  x0=tau0+tau2, x1=tau1-itau3, store x01
    data[2*k]=_mm_add_ps(t2,t3);
    //  x2=tau0-tau2, x3=tau1+itau3, store x23
    data[2*k+1]=_mm_sub_ps(t2,t3);
}

//  stages q=2..t-1 -----------------------------------------------
for (q=2;q<t;q++)
{
    Ls=L;
    L<<=2;      // L = 4^q
    r>>=2;

    //  j=0 extra
    for (k=0;k<r;k++)
    {
        //  load data
        //  load x0, x1
        x01=_mm_shuffle_ps(data[k*L],data[k*L+Ls],_MM_SHUFFLE(1,0,1,0));
        //  load x2, x3
        x23=_mm_shuffle_ps(data[k*L+2*Ls],data[k*L+3*Ls],
          _MM_SHUFFLE(1,0,1,0));
        //  load x4, x5
        x45=_mm_shuffle_ps(data[k*L],data[k*L+Ls],_MM_SHUFFLE(3,2,3,2));
        //  load x6, x7
        x67=_mm_shuffle_ps(data[k*L+2*Ls],data[k*L+3*Ls],
          _MM_SHUFFLE(3,2,3,2));

        //  tau0=alpha+gamma, tau1=beta+delta
        t0=_mm_add_ps(x01,x23);
        //  tau2=alpha-gamma, tau3=beta-delta
        t1=_mm_sub_ps(x01,x23);

        //  reorder to calc tau0+tau1, tau2-itau3,...
        //  extract tau0, tau1
        t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));
        //  extract tau2, itau3
        t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));
        //  negate highestorder float for (-itau3)
        t3=_mm_mul_ps(t3,neglast);
        //  x0=tau0+tau2, x1=tau1-itau3
        x01=_mm_add_ps(t2,t3);
        //  x0=tau0-tau2, x1=tau1+itau3
        x23=_mm_sub_ps(t2,t3);

        //  compute x45, x67
        //  compute (a+bi)(c+di)=(ac-bd)+(ad+bc)i
```

```
        //  real parts
        t0=_mm_mul_ps(x45,radix4_twiddle[q][4]);        // calc ac,bd
        t1=_mm_mul_ps(x67,radix4_twiddle[q][5]);
        t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));  // extract ac
        t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));  //  extract bd
        real=_mm_sub_ps(t2,t3);         // ac-bd ready

        //  imag parts
        t0=_mm_mul_ps(x45,radix4_twiddle[q][6]);        // calc ad,bc
        t1=_mm_mul_ps(x67,radix4_twiddle[q][7]);
        t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));  // extract ad
        t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));  //  extract bc
        imag=_mm_add_ps(t2,t3);         // ad+bc ready

        //  unpack into (ac-bd)+(ad+bc)i
        x45=_mm_unpacklo_ps(real,imag);
        x67=_mm_unpackhi_ps(real,imag);

        //  tau0=alpha+gamma, tau1=beta+delta
        t0=_mm_add_ps(x45,x67);
        //  tau2=alpha-gamma, tau3=beta-delta
        t1=_mm_sub_ps(x45,x67);

        //  reorder to calc tau0+tau1, tau2-itau3,...
        //  extract tau0, tau1
        t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));
        //  extract tau2, itau3
        t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));
        //  negate highestorder float for (-itau3)
        t3=_mm_mul_ps(t3,neglast);

        //  x0=tau0+tau2, x1=tau1-itau3
        x45=_mm_add_ps(t2,t3);
        //  x0=tau0-tau2, x1=tau1+itau3
        x67=_mm_sub_ps(t2,t3);

        //  store x0, x1
        data[k*L]=_mm_shuffle_ps(x01,x45,_MM_SHUFFLE(1,0,1,0));
        //  store x2, x3
        data[k*L+Ls]=_mm_shuffle_ps(x01,x45,_MM_SHUFFLE(3,2,3,2));
        //  store x4, x5
        data[k*L+2*Ls]=_mm_shuffle_ps(x23,x67,_MM_SHUFFLE(1,0,1,0));
        //  store x6, x7
        data[k*L+3*Ls]=_mm_shuffle_ps(x23,x67,_MM_SHUFFLE(3,2,3,2));
    }

    for (j=1;j<Ls;j++)  //  do 2 j-loops simultanously
    {
        for (k=0;k<r;k++)
        {
            //  load data
            //  load x0, x1
```

```
x01=_mm_shuffle_ps(data[k*L+j],data[k*L+Ls+j],
  _MM_SHUFFLE(1,0,1,0));
//  load x2, x3
x23=_mm_shuffle_ps(data[k*L+2*Ls+j],data[k*L+3*Ls+j],
  _MM_SHUFFLE(1,0,1,0));
//  load x4, x5
x45=_mm_shuffle_ps(data[k*L+j],data[k*L+Ls+j],
  _MM_SHUFFLE(3,2,3,2));
//  load x6, x7
x67=_mm_shuffle_ps(data[k*L+2*Ls+j],data[k*L+3*Ls+j],
  _MM_SHUFFLE(3,2,3,2));

//  compute x01, x23
//  compute (a+bi)(c+di)=(ac-bd)+(ad+bc)i
//  real parts
// calc ac,bd
t0=_mm_mul_ps(x01,radix4_twiddle[q][8*j]);
t1=_mm_mul_ps(x23,radix4_twiddle[q][8*j+1]);
// extract ac
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));
//  extract bd
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));
real=_mm_sub_ps(t2,t3);             // ac-bd ready

//  imag parts
// calc ad,bc
t0=_mm_mul_ps(x01,radix4_twiddle[q][8*j+2]);
t1=_mm_mul_ps(x23,radix4_twiddle[q][8*j+3]);
// extract ad
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));
//  extract bc
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));
imag=_mm_add_ps(t2,t3);             // ad+bc ready

//  unpack into (ac-bd)+(ad+bc)i
x01=_mm_unpacklo_ps(real,imag);
x23=_mm_unpackhi_ps(real,imag);

//  tau0=alpha+gamma, tau1=beta+delta
t0=_mm_add_ps(x01,x23);
//  tau2=alpha-gamma, tau3=beta-delta
t1=_mm_sub_ps(x01,x23);

//  reorder to calc tau0+tau1, tau2-itau3,...
//  extract tau0, tau1
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));
//  extract tau2, itau3
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));
//  negate highestorder float for (-itau3)
t3=_mm_mul_ps(t3,neglast);

//  x0=tau0+tau2, x1=tau1-itau3
```

```
x01=_mm_add_ps(t2,t3);
//  x0=tau0-tau2, x1=tau1+itau3
x23=_mm_sub_ps(t2,t3);

//  compute x45, x67
//  compute (a+bi)(c+di)=(ac-bd)+(ad+bc)i
//  real parts
// calc ac,bd
t0=_mm_mul_ps(x45,radix4_twiddle[q][8*j+4]);
t1=_mm_mul_ps(x67,radix4_twiddle[q][8*j+5]);
// extract ac
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));
//  extract bd
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));
real=_mm_sub_ps(t2,t3);         // ac-bd ready

//  imag parts
// calc ad,bc
t0=_mm_mul_ps(x45,radix4_twiddle[q][8*j+6]);
t1=_mm_mul_ps(x67,radix4_twiddle[q][8*j+7]);
// extract ad
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));
//  extract bc
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));
imag=_mm_add_ps(t2,t3);         // ad+bc ready

//  unpack into (ac-bd)+(ad+bc)i
x45=_mm_unpacklo_ps(real,imag);
x67=_mm_unpackhi_ps(real,imag);

//  tau0=alpha+gamma, tau1=beta+delta
t0=_mm_add_ps(x45,x67);
//  tau2=alpha-gamma, tau3=beta-delta
t1=_mm_sub_ps(x45,x67);

//  reorder to calc tau0+tau1, tau2-itau3,...
//  extract tau0, tau1
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));
//  extract tau2, itau3
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));
//  negate highestorder float for (-itau3)
t3=_mm_mul_ps(t3,neglast);
//  x0=tau0+tau2, x1=tau1-itau3
x45=_mm_add_ps(t2,t3);
//  x0=tau0-tau2, x1=tau1+itau3
x67=_mm_sub_ps(t2,t3);

//  store x0, x1
data[k*L+j]=_mm_shuffle_ps(x01,x45,
  _MM_SHUFFLE(1,0,1,0));
//  store x2, x3
data[k*L+j+Ls]=_mm_shuffle_ps(x01,x45,
```

```
                _MM_SHUFFLE(3,2,3,2));
              //  store x4, x5
              data[k*L+j+2*Ls]=_mm_shuffle_ps(x23,x67,
                _MM_SHUFFLE(1,0,1,0));
              //  store x6, x7
              data[k*L+j+3*Ls]=_mm_shuffle_ps(x23,x67,
                _MM_SHUFFLE(3,2,3,2));
          }
      }
  }

  //  stage q=4 (=t)  ------------------------------------------------
  if (t>1)
  {
      q=t;
      r=1;
      Ls=L;   //  to correct SIMD=4

      //  j=0:    trivial twiddlefactors
      //  load data
      //  load x0, x1
      x01=_mm_shuffle_ps(data[0],data[Ls],_MM_SHUFFLE(1,0,1,0));
      //  load x2, x3
      x23=_mm_shuffle_ps(data[2*Ls],data[3*Ls],_MM_SHUFFLE(1,0,1,0));
      //  load x4, x5
      x45=_mm_shuffle_ps(data[0],data[Ls],_MM_SHUFFLE(3,2,3,2));
      //  load x6, x7
      x67=_mm_shuffle_ps(data[2*Ls],data[3*Ls],_MM_SHUFFLE(3,2,3,2));

      //  calc x0,...,x3
      //  tau0=alpha+gamma, tau1=beta+delta
      t0=_mm_add_ps(x01,x23);
      //  tau2=alpha-gamma, tau3=beta-delta
      t1=_mm_sub_ps(x01,x23);

      //  reorder to calc tau0+tau1, tau2-itau3,...
      //  extract tau0, tau1
      t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));
      //  extract tau2, itau3
      t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));
      //  negate highestorder float for (-itau3)
      t3=_mm_mul_ps(t3,neglast);

      //  x0=tau0+tau2, x1=tau1-itau3
      x01=_mm_add_ps(t2,t3);
      //  x0=tau0-tau2, x1=tau1+itau3
      x23=_mm_sub_ps(t2,t3);

      //  calculate x4,...,x7
      //  compute (a+bi)(c+di)=(ac-bd)+(ad+bc)i
      //  real parts
      t0=_mm_mul_ps(x45,radix4_twiddle[q][4]);        // calc ac,bd
```

```
t1=_mm_mul_ps(x67,radix4_twiddle[q][5]);
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));  // extract ac
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));  //  extract bd
real=_mm_sub_ps(t2,t3);             // ac-bd ready

//  imag parts
t0=_mm_mul_ps(x45,radix4_twiddle[q][6]);        // calc ad,bc
t1=_mm_mul_ps(x67,radix4_twiddle[q][7]);
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));  // extract ad
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));  //  extract bc
imag=_mm_add_ps(t2,t3);             // ad+bc ready

//  unpack into (ac-bd)+(ad+bc)i
x45=_mm_unpacklo_ps(real,imag);
x67=_mm_unpackhi_ps(real,imag);

//  tau0=alpha+gamma, tau1=beta+delta
t0=_mm_add_ps(x45,x67);
//  tau2=alpha-gamma, tau3=beta-delta
t1=_mm_sub_ps(x45,x67);

//  reorder to calc tau0+tau1, tau2-itau3,...
//  extract tau0, tau1
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));
//  extract tau2, itau3
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));
//  negate highestorder float for (-itau3)
t3=_mm_mul_ps(t3,neglast);

//  x0=tau0+tau2, x1=tau1-itau3
x45=_mm_add_ps(t2,t3);
//  x0=tau0-tau2, x1=tau1+itau3
x67=_mm_sub_ps(t2,t3);

//  store x0, x1
data[0]=_mm_shuffle_ps(x01,x45,_MM_SHUFFLE(1,0,1,0));
//  store x2, x3
data[Ls]=_mm_shuffle_ps(x01,x45,_MM_SHUFFLE(3,2,3,2));
//  store x4, x5
data[2*Ls]=_mm_shuffle_ps(x23,x67,_MM_SHUFFLE(1,0,1,0));
//  store x6, x7
data[3*Ls]=_mm_shuffle_ps(x23,x67,_MM_SHUFFLE(3,2,3,2));

//  j=1..Ls-1 - standard computation
for (j=1;j<Ls;j++)  //  2 loops simultanously
{
    //  load x0, x1
    x01=_mm_shuffle_ps(data[j],data[Ls+j],
      _MM_SHUFFLE(1,0,1,0));
    //  load x2, x3
    x23=_mm_shuffle_ps(data[2*Ls+j],data[3*Ls+j],
      _MM_SHUFFLE(1,0,1,0));
```

```
//  load x4, x5
x45=_mm_shuffle_ps(data[j],data[Ls+j],
  _MM_SHUFFLE(3,2,3,2));
//  load x6, x7
x67=_mm_shuffle_ps(data[2*Ls+j],data[3*Ls+j],
  _MM_SHUFFLE(3,2,3,2));

//  calculate x0,...,x3
//  compute (a+bi)(c+di)=(ac-bd)+(ad+bc)i
//  real parts
t0=_mm_mul_ps(x01,radix4_twiddle[q][8*j]);      // calc ac,bd
t1=_mm_mul_ps(x23,radix4_twiddle[q][8*j+1]);
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));  // extract ac
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));  //  extract bd
real=_mm_sub_ps(t2,t3);         // ac-bd ready

//  imag parts
t0=_mm_mul_ps(x01,radix4_twiddle[q][8*j+2]);    // calc ad,bc
t1=_mm_mul_ps(x23,radix4_twiddle[q][8*j+3]);
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));  // extract ad
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));  //  extract bc
imag=_mm_add_ps(t2,t3);         // ad+bc ready

//  unpack into (ac-bd)+(ad+bc)i
x01=_mm_unpacklo_ps(real,imag);
x23=_mm_unpackhi_ps(real,imag);

//  tau0=alpha+gamma, tau1=beta+delta
t0=_mm_add_ps(x01,x23);
//  tau2=alpha-gamma, tau3=beta-delta
t1=_mm_sub_ps(x01,x23);

//  reorder to calc tau0+tau1, tau2-itau3,...
//  extract tau0, tau1
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));
//  extract tau2, itau3
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));
//  negate highestorder float for (-itau3)
t3=_mm_mul_ps(t3,neglast);

//  x0=tau0+tau2, x1=tau1-itau3
x01=_mm_add_ps(t2,t3);
//  x0=tau0-tau2, x1=tau1+itau3
x23=_mm_sub_ps(t2,t3);

//  calculate x4,...,x7
//  compute (a+bi)(c+di)=(ac-bd)+(ad+bc)i
//  real parts
t0=_mm_mul_ps(x45,radix4_twiddle[q][8*j+4]);    // calc ac,bd
t1=_mm_mul_ps(x67,radix4_twiddle[q][8*j+5]);
t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));  // extract ac
t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));  //  extract bd
```

```
            real=_mm_sub_ps(t2,t3);         // ac-bd ready

            //  imag parts
            t0=_mm_mul_ps(x45,radix4_twiddle[q][8*j+6]);    // calc ad,bc
            t1=_mm_mul_ps(x67,radix4_twiddle[q][8*j+7]);
            t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,0,2,0));  // extract ad
            t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(3,1,3,1));  //  extract bc
            imag=_mm_add_ps(t2,t3);         //  ad+bc ready

            //  unpack into (ac-bd)+(ad+bc)i
            x45=_mm_unpacklo_ps(real,imag);
            x67=_mm_unpackhi_ps(real,imag);

            //  tau0=alpha+gamma, tau1=beta+delta
            t0=_mm_add_ps(x45,x67);
            //  tau2=alpha-gamma, tau3=beta-delta
            t1=_mm_sub_ps(x45,x67);

            //  reorder to calc tau0+tau1, tau2-itau3,...
            //  extract tau0, tau1
            t2=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(1,0,1,0));
            //  extract tau2, itau3
            t3=_mm_shuffle_ps(t0,t1,_MM_SHUFFLE(2,3,3,2));
            //  negate highestorder float for (-itau3)
            t3=_mm_mul_ps(t3,neglast);

            //  x0=tau0+tau2, x1=tau1-itau3
            x45=_mm_add_ps(t2,t3);
            //  x0=tau0-tau2, x1=tau1+itau3
            x67=_mm_sub_ps(t2,t3);

            //  store x0, x1
            data[j]=_mm_shuffle_ps(x01,x45,_MM_SHUFFLE(1,0,1,0));
            //  store x2, x3
            data[j+Ls]=_mm_shuffle_ps(x01,x45,_MM_SHUFFLE(3,2,3,2));
            //  store x4, x5
            data[j+2*Ls]=_mm_shuffle_ps(x23,x67,_MM_SHUFFLE(1,0,1,0));
            //  store x6, x7
            data[j+3*Ls]=_mm_shuffle_ps(x23,x67,_MM_SHUFFLE(3,2,3,2));
        }
    }
}


//  Radix 4 for n>=16

void vect_radix4(__m128 *real,__m128 *imag, int t)
{
    __m128 alpha_r,beta_r,gamma_r,delta_r,alpha_i,beta_i,gamma_i,delta_i,
        tau0_r,tau1_r,tau2_r,tau3_r,tau0_i,tau1_i,tau2_i,tau3_i,
        x0_r,x1_r,x2_r,x3_r,x0_i,x1_i,x2_i,x3_i,
        omega_r,omega_i,omega2_r,omega2_i,omega3_r,omega3_i;
```

```
__m128 **localtwiddle,*stagetwiddle,
    *localreal,*localimag;

int i,j,k,q,
    L,r,Ls;


localtwiddle=radix4_twiddle;
localreal=real;
localimag=imag;

//  Stage q=1   ----------------------------------------------
L=4;
r=powers[t-2];
Ls=1;

for (k=0;k<r;k++)
{
    //  load real parts
    alpha_r=localreal[4*k];
    beta_r=localreal[4*k+1];
    gamma_r=localreal[4*k+2];
    delta_r=localreal[4*k+3];

    //  extract vectors xmm0=alpha_r, xmm1=beta_r,
    //  xmm2=gamma_r,xmm3=delta_r
    _MM_TRANSPOSE4_PS(alpha_r,beta_r,gamma_r,delta_r);

    //  tau0=alpha+gamma
    tau0_r=_mm_add_ps(alpha_r,gamma_r);
    //  tau1=alpha-gamma
    tau1_r=_mm_sub_ps(alpha_r,gamma_r);
    //  tau2=beta+delta
    tau2_r=_mm_add_ps(beta_r,delta_r);
    //  tau3=beta-delta
    tau3_r=_mm_sub_ps(beta_r,delta_r);

    //  load imag parts
    alpha_i=localimag[k];
    beta_i=localimag[k+1];
    gamma_i=localimag[k+2];
    delta_i=localimag[k+3];

    //  extract vectors xmm0=alpha_i, xmm1=beta_i,
    //  xmm2=gamma_i, xmm3=delta_i
    _MM_TRANSPOSE4_PS(alpha_i,beta_i,gamma_i,delta_i);

    //  tau0=alpha+gamma
    tau0_i=_mm_add_ps(alpha_i,gamma_i);
    //  tau1=alpha-gamma
    tau1_i=_mm_sub_ps(alpha_i,gamma_i);
    //  tau2=beta+delta
```

```
        tau2_i=_mm_add_ps(beta_i,delta_i);
        //  tau3=beta-delta
        tau3_i=_mm_sub_ps(beta_i,delta_i);

        //  results real parts
        //  x0=tau0+tau2
        x0_r=_mm_add_ps(tau0_r,tau2_r);
        //  x1=tau1-itau3
        x1_r=_mm_add_ps(tau1_r,tau3_i);
        //  x2=tau0-tau2
        x2_r=_mm_sub_ps(tau0_r,tau2_r);
        //  x3=tau1+itau3
        x3_r=_mm_sub_ps(tau1_r,tau3_i);

        //  build data vectors
        _MM_TRANSPOSE4_PS(x0_r,x1_r,x2_r,x3_r);
        //  store real parts
        localreal[4*k]=x0_r;
        localreal[4*k+1]=x1_r;
        localreal[4*k+2]=x2_r;
        localreal[4*k+3]=x3_r;

        //  results imag parts
        //  x0=tau0+tau2
        x0_i=_mm_add_ps(tau0_i,tau2_i);
        //  x1=tau1-itau3
        x1_i=_mm_sub_ps(tau1_i,tau3_r);
        //  x2=tau0-tau2
        x2_i=_mm_sub_ps(tau0_i,tau2_i);
        //  x3=tau1+itau3
        x3_i=_mm_add_ps(tau1_i,tau3_r);

        //  build data vectors
        _MM_TRANSPOSE4_PS(x0_i,x1_i,x2_i,x3_i);
        //  store imag parts
        localimag[k]=x0_i;
        localimag[k+1]=x1_i;
        localimag[k+2]=x2_i;
        localimag[k+3]=x3_i;
    }

    //  Stage q=2..t-1  ---------------------------------------------------
    L=1;
    r=powers[t-1];
    for (q=2;q<t;q++)
    {
        //  L=4^q, r=n/L, Ls=L/4
        Ls=L;
        L<<=2;
        r>>=2;
        //  local pointer to twiddlefactors
        stagetwiddle=localtwiddle[q];
```

```
for (j=0;j<Ls;j++)
{
    //  load twiddlefactors
    omega_r=stagetwiddle[6*j];
    omega_i=stagetwiddle[6*j+1];
    omega2_r=stagetwiddle[6*j+2];
    omega2_i=stagetwiddle[6*j+3];
    omega3_r=stagetwiddle[6*j+4];
    omega3_i=stagetwiddle[6*j+5];

    for (k=0;k<r;k++)
    {
        //  alpha=x0
        alpha_r=localreal[k*L+j];
        alpha_i=localimag[k*L+j];
        //  beta=w*x1
        x1_r=localreal[k*L+Ls+j];
        x1_i=localimag[k*L+Ls+j];
        beta_r=_mm_sub_ps(_mm_mul_ps(x1_r,omega_r),
            _mm_mul_ps(x1_i,omega_i));
        beta_i=_mm_add_ps(_mm_mul_ps(x1_r,omega_i),
            _mm_mul_ps(x1_i,omega_r));
        //  gamma=w^2*x2
        x2_r=localreal[k*L+2*Ls+j];
        x2_i=localimag[k*L+2*Ls+j];
        gamma_r=_mm_sub_ps(_mm_mul_ps(x1_r,omega2_r),
            _mm_mul_ps(x1_i,omega2_i));
        gamma_i=_mm_add_ps(_mm_mul_ps(x1_r,omega2_i),
            _mm_mul_ps(x1_i,omega2_r));
        //  delta=w^3*x3
        x3_r=localreal[k*L+3*Ls+j];
        x3_i=localimag[k*L+3*Ls+j];
        delta_r=_mm_sub_ps(_mm_mul_ps(x1_r,omega3_r),
            _mm_mul_ps(x1_i,omega3_i));
        delta_i=_mm_add_ps(_mm_mul_ps(x1_r,omega3_i),
            _mm_mul_ps(x1_i,omega3_r));

        //  tau0=alpha+gamma
        tau0_r=_mm_add_ps(alpha_r,gamma_r);
        tau0_i=_mm_add_ps(alpha_i,gamma_i);
        //  tau1=alpha-gamma;
        tau1_r=_mm_sub_ps(alpha_r,gamma_r);
        tau1_i=_mm_sub_ps(alpha_i,gamma_i);
        //  tau2=beta+delta
        tau2_r=_mm_add_ps(beta_r,delta_r);
        tau2_i=_mm_add_ps(beta_i,delta_i);
        //  tau3=beta-delta;
        tau3_r=_mm_sub_ps(beta_r,gamma_r);
        tau3_i=_mm_sub_ps(beta_i,delta_i);

        //  x0=tau0+tau2
        x0_r=_mm_add_ps(tau0_r,tau2_r);
```

```
            x0_i=_mm_add_ps(tau0_i,tau2_i);
            //  x1=tau1-itau3
            x1_r=_mm_add_ps(tau1_r,tau3_i);
            x1_i=_mm_sub_ps(tau1_i,tau3_r);
            //  x2=tau0-tau2
            x2_r=_mm_sub_ps(tau0_r,tau2_r);
            x2_i=_mm_sub_ps(tau0_i,tau2_i);
            //  x3=tau1-itau3
            x3_r=_mm_sub_ps(tau1_r,tau3_i);
            x3_i=_mm_add_ps(tau1_i,tau3_r);

            //  store data
            localreal[k*L+j]=x0_r;
            localimag[k*L+j]=x0_i;
            localreal[k*L+Ls+j]=x1_r;
            localimag[k*L+Ls+j]=x1_i;
            localreal[k*L+2*Ls+j]=x2_r;
            localimag[k*L+2*Ls+j]=x2_i;
            localreal[k*L+3*Ls+j]=x3_r;
            localimag[k*L+3*Ls+j]=x3_i;
        }

    }

}

//  Stage q=t   ----------------------------------------------------
//  L=n, r=1, Ls=L/4
q=t;
Ls=L;
L<<=2;
r>>=2;
//  local pointer to twiddlefactors
stagetwiddle=localtwiddle[q];

for (j=0;j<Ls;j++)
{
    //  load twiddlefactors
    omega_r=stagetwiddle[6*j];
    omega_i=stagetwiddle[6*j+1];
    omega2_r=stagetwiddle[6*j+2];
    omega2_i=stagetwiddle[6*j+3];
    omega3_r=stagetwiddle[6*j+4];
    omega3_i=stagetwiddle[6*j+5];

    //  alpha=x0
    alpha_r=localreal[j];
    alpha_i=localimag[j];
    //  beta=w*x1
    x1_r=localreal[Ls+j];
    x1_i=localimag[Ls+j];
    beta_r=_mm_sub_ps(_mm_mul_ps(x1_r,omega_r),
```

```
    _mm_mul_ps(x1_i,omega_i));
beta_i=_mm_add_ps(_mm_mul_ps(x1_r,omega_i),
    _mm_mul_ps(x1_i,omega_r));
//  gamma=w^2*x2
x2_r=localreal[2*Ls+j];
x2_i=localimag[2*Ls+j];
gamma_r=_mm_sub_ps(_mm_mul_ps(x1_r,omega2_r),
    _mm_mul_ps(x1_i,omega2_i));
gamma_i=_mm_add_ps(_mm_mul_ps(x1_r,omega2_i),
    _mm_mul_ps(x1_i,omega2_r));
//  delta=w^3*x3
x3_r=localreal[3*Ls+j];
x3_i=localimag[3*Ls+j];
delta_r=_mm_sub_ps(_mm_mul_ps(x1_r,omega3_r),
    _mm_mul_ps(x1_i,omega3_i));
delta_i=_mm_add_ps(_mm_mul_ps(x1_r,omega3_i),
    _mm_mul_ps(x1_i,omega3_r));

//  tau0=alpha+gamma
tau0_r=_mm_add_ps(alpha_r,gamma_r);
tau0_i=_mm_add_ps(alpha_i,gamma_i);
//  tau1=alpha-gamma;
tau1_r=_mm_sub_ps(alpha_r,gamma_r);
tau1_i=_mm_sub_ps(alpha_i,gamma_i);
//  tau2=beta+delta
tau2_r=_mm_add_ps(beta_r,delta_r);
tau2_i=_mm_add_ps(beta_i,delta_i);
//  tau3=beta-delta;
tau3_r=_mm_sub_ps(beta_r,gamma_r);
tau3_i=_mm_sub_ps(beta_i,delta_i);

//  x0=tau0+tau2
x0_r=_mm_add_ps(tau0_r,tau2_r);
x0_i=_mm_add_ps(tau0_i,tau2_i);
//  x1=tau1-itau3
x1_r=_mm_add_ps(tau1_r,tau3_i);
x1_i=_mm_sub_ps(tau1_i,tau3_r);
//  x2=tau0-tau2
x2_r=_mm_sub_ps(tau0_r,tau2_r);
x2_i=_mm_sub_ps(tau0_i,tau2_i);
//  x3=tau1-itau3
x3_r=_mm_sub_ps(tau1_r,tau3_i);
x3_i=_mm_add_ps(tau1_i,tau3_r);

//  store data
localreal[j]=x0_r;
localimag[j]=x0_i;
localreal[Ls+j]=x1_r;
localimag[Ls+j]=x1_i;
localreal[2*Ls+j]=x2_r;
localimag[2*Ls+j]=x2_i;
localreal[3*Ls+j]=x3_r;
```

```
        localimag[3*Ls+j]=x3_i;
    }
}
```

Appendix B

Source Code of FFTW Modifications

B.1 fftw-simd.h

```
/*      fftw-simd.h

        definitions for the simd-enabled fftw codelets

*/

#ifndef __FFTW_SIMD_H
#define __FFTW_SIMD_H

#ifdef __ICL
#include "xmmintrin.h"
#endif

#ifdef __cplusplus
extern "C" {
#endif

/*  define Flag for SIMD-Usage      */
#define FFTW_USE_SIMD       0x2000

/*  define SIMD vectorlength        */
#define FFTW_LD_SIMD_LEN    2
#define FFTW_SIMD_LEN       4

/*  Intel C/C++ Compiler specific
---------------------------------- */
#ifdef __ICL

/*  define data types               */
typedef __m128
FFTW_SIMD_VECT;
typedef __m64 FFTW_SIMD_COMPLEX;

/*  define const operations         */
#define FFTW_SIMD_KONST(c,v)                                    \
    static const __declspec(align(16)) float (c)[4]={v,v,v,v}
#define FFTW_LOAD_KONST_SIMD(c) *(FFTW_SIMD_VECT *)(c)

/*  define arithmetic operations    */
#define SIMD_ADD(a,b)   _mm_add_ps((a),(b))
```

```
#define SIMD_SUB(a,b)   _mm_sub_ps((a),(b))
#define SIMD_MUL(a,b)   _mm_mul_ps((a),(b))

/*  define reorder operations       */
#define GET_REAL_VECT(cmplx1,cmplx2)                                \
    _mm_shuffle_ps((cmplx1),(cmplx2),_MM_SHUFFLE(2,0,2,0))

#define GET_IMAG_VECT(cmplx1,cmplx2)                                \
    _mm_shuffle_ps((cmplx1),(cmplx2),_MM_SHUFFLE(3,1,3,1))

#define FILL_RE(cmplx)                                              \
    _mm_shuffle_ps((cmplx),(cmplx),_MM_SHUFFLE(0,0,0,0))

#define FILL_IM(cmplx)                                              \
    _mm_shuffle_ps((cmplx),(cmplx),_MM_SHUFFLE(1,1,1,1))

#define GET_COMPLEX_LO(re,im)                                       \
    _mm_unpacklo_ps((re),(im))

#define GET_COMPLEX_HI(re,im)                                       \
    _mm_unpackhi_ps((re),(im))

/*  define load operations          */ #define
LOAD_COMPLEX_LO(result,complex)                                     \
    (result) = _mm_loadl_pi((result),(complex))

#define LOAD_COMPLEX_HI(result,complex)                             \
    (result) = _mm_loadh_pi((result),(complex))

#define LOAD_RE_IM(re,im,input,stride)                              \
{                                                                   \
    FFTW_SIMD_VECT ldtmp1,ldtmp2;                                   \
    LOAD_COMPLEX_LO(ldtmp1,(input));                                \
    LOAD_COMPLEX_HI(ldtmp1,(input) + (stride));                     \
    LOAD_COMPLEX_LO(ldtmp2,(input) + 2 * (stride));                 \
    LOAD_COMPLEX_HI(ldtmp2,(input) + 3 * (stride));                 \
    (re)=GET_REAL_VECT(ldtmp1,ldtmp2);                              \
    (im)=GET_IMAG_VECT(ldtmp1,ldtmp2);                              \
}

#define LOAD_TWIDDLE_RE_IM(re,im,tw)                                \
{                                                                   \
    FFTW_SIMD_VECT tmp;                                             \
    LOAD_COMPLEX_LO(tmp,(tw));                                      \
    (re)=FILL_RE(tmp);                                              \
    (im)=FILL_IM(tmp);                                              \
}


/*  define store operations         */
#define STORE_COMPLEX_LO(mem,vcmplx) _mm_storel_pi((mem),(vcmplx))
```

```
#define STORE_COMPLEX_HI(mem,vcmplx) _mm_storeh_pi((mem),(vcmplx))

#define STORE_RE_IM(re,im,output,stride)                          \
{                                                                 \
    FFTW_SIMD_VECT sttmp1,sttmp2;                                 \
    sttmp1=GET_COMPLEX_LO((re),(im));                             \
    sttmp2=GET_COMPLEX_HI((re),(im));                             \
    STORE_COMPLEX_LO((output),sttmp1);                            \
    STORE_COMPLEX_HI((output) + (stride),sttmp1);                 \
    STORE_COMPLEX_LO((output) + 2 * (stride),sttmp2);             \
    STORE_COMPLEX_HI((output) + 3 * (stride),sttmp2);             \
}

#endif

/*  GNU gcc-vec for Motorola G4 specific
------------------------------ */

#ifdef __GNUC__
/*  define data types     */
typedef vector float
FFTW_SIMD_VECT;
typedef struct c{float re,im;} FFTW_SIMD_COMPLEX;

#define VEC_NULL (FFTW_SIMD_VECT)(0.0,0.0,0.0,0.0)

/*  define const operations         */
#define FFTW_SIMD_KONST(c,v)                                      \
    static const vector float (c)=(vector float)(v)
#define FFTW_LOAD_KONST_SIMD(c) (c)

/*  define arithmetic operations    */
#define SIMD_ADD(a,b) vec_add((a),(b))
#define SIMD_SUB(a,b) vec_sub((a),(b))
#define SIMD_MUL(a,b) vec_madd((a),(b),VEC_NULL)
#define SIMD_MADD(a,b,c) vec_madd((a),(b),(c))
#define SIMD_NMSUB(a,b,c) vec_nmsub((a),(b),(c))
#define SIMD_NMUL(a,b) vec_nmsub((a),(b),VEC_NULL)

#define LOAD_COMPLEX(c,fp)                                        \
{                                                                 \
    FFTW_SIMD_VECT lc_tmp;                                        \
    lc_tmp=vec_ld(0,(float *)(fp));                               \
    (c)=vec_perm(lc_tmp,lc_tmp,vec_lvsl(0,(float *)(fp)));        \
}

#define LOAD_RE_IM(re,im,input,stride)                            \
{                                                                 \
    FFTW_SIMD_VECT lri_tmp0,lri_tmp1,                             \
        lri_tmp2,lri_tmp3,lri_tmp4,lri_tmp5;                      \
    LOAD_COMPLEX(lri_tmp0,(FFTW_COMPLEX*)(input));                \
    LOAD_COMPLEX(lri_tmp1,(FFTW_COMPLEX*)(input)+(stride));       \
```

```
    LOAD_COMPLEX(lri_tmp2,(FFTW_COMPLEX*)(input)+2*(stride));   \
    LOAD_COMPLEX(lri_tmp3,(FFTW_COMPLEX*)(input)+3*(stride));   \
    lri_tmp4=vec_mergeh(lri_tmp0,lri_tmp2);                     \
    lri_tmp5=vec_mergeh(lri_tmp1,lri_tmp3);                     \
    (re)=vec_mergeh(lri_tmp4,lri_tmp5);                         \
    (im)=vec_mergel(lri_tmp4,lri_tmp5);                         \
}

#define LOAD_TWIDDLE_RE_IM(re,im,tw)                            \
{                                                               \
    FFTW_SIMD_VECT ltwri_tmp;                                   \
    LOAD_COMPLEX(ltwri_tmp,tw);                                 \
    (re)=vec_splat(ltwri_tmp,0);                                \
    (im)=vec_splat(ltwri_tmp,1);                                \
}


/*  define store operations         */

#define STORE_COMPLEX_1(c,fp)                                   \
{                                                               \
    FFTW_SIMD_VECT sc_tmp;                                      \
    sc_tmp=vec_perm((c),(c),vec_lvsr(0,(float *)(fp)));         \
    vec_ste(sc_tmp,0,(float *)(fp));                            \
    vec_ste(sc_tmp,4,(float *)(fp));                            \
}

#define STORE_COMPLEX_2(c,fp)                                   \
{                                                               \
    FFTW_SIMD_VECT sc_tmp;                                      \
    sc_tmp=vec_perm((c),(c),vec_lvsl(8,(float *)(fp)));         \
    vec_ste(sc_tmp,0,(float *)(fp));                            \
    vec_ste(sc_tmp,4,(float *)(fp));                            \
}

#define STORE_RE_IM(re,im,output,stride)                        \
{                                                               \
    FFTW_SIMD_VECT sri_tmp0,sri_tmp1;                           \
    sri_tmp0=vec_mergeh((re),(im));                             \
    sri_tmp1=vec_mergel((re),(im));                             \
    STORE_COMPLEX_1((sri_tmp0),(FFTW_COMPLEX*)(output));        \
    STORE_COMPLEX_2((sri_tmp0),                                 \
        (FFTW_COMPLEX*)(output)+(stride));                      \
    STORE_COMPLEX_1((sri_tmp1),                                 \
        (FFTW_COMPLEX*)(output)+2*(stride));                    \
    STORE_COMPLEX_2((sri_tmp1),                                 \
        (FFTW_COMPLEX*)(output)+3*(stride));                    \
}

#endif

#ifdef __cplusplus
```

```
}
#endif

#endif
```

B.2 Radix-4 No Twiddle Codelets

```
#include "fftw-simd.h" #include <fftw-int.h> #include <fftw.h>

/* Generated by: ./genfft -notwiddlesimd 4 */

/*
 * This function contains 16 FP additions, 0 FP multiplications,
 * (or, 16 additions, 0 multiplications, 0 fused multiply/add),
 * 12 stack variables, and 16 memory accesses
 */


/*
 * Generator Id's :
 * $Id: exprdag.ml,v 1.40 1999/05/17 14:08:53 fftw Exp $
 * $Id: fft.ml,v 1.43 1999/05/17 19:44:18 fftw Exp $
 * $Id: to_c.ml,v 1.24 1999/02/19 17:22:17 athena Exp $
 */

void fftw_no_twiddle_4 (const fftw_complex * input, fftw_complex
* output,
    int istride, int ostride)
{
  fftw_real tmp3;
  fftw_real tmp11;
  fftw_real tmp9;
  fftw_real tmp15;
  fftw_real tmp6;
  fftw_real tmp10;
  fftw_real tmp14;
  fftw_real tmp16;
  {
    fftw_real tmp1;
    fftw_real tmp2;
    fftw_real tmp7;
    fftw_real tmp8;
    tmp1 = c_re (input[0]);
    tmp2 = c_re (input[2 * istride]);
    tmp3 = (tmp1 + tmp2);
    tmp11 = (tmp1 - tmp2);
    tmp7 = c_im (input[0]);
    tmp8 = c_im (input[2 * istride]);
    tmp9 = (tmp7 - tmp8);
    tmp15 = (tmp7 + tmp8);
```

```
  }
  {
    fftw_real tmp4;
    fftw_real tmp5;
    fftw_real tmp12;
    fftw_real tmp13;
    tmp4 = c_re (input[istride]);
    tmp5 = c_re (input[3 * istride]);
    tmp6 = (tmp4 + tmp5);
    tmp10 = (tmp4 - tmp5);
    tmp12 = c_im (input[istride]);
    tmp13 = c_im (input[3 * istride]);
    tmp14 = (tmp12 - tmp13);
    tmp16 = (tmp12 + tmp13);
  }
  c_re (output[2 * ostride]) = (tmp3 - tmp6);
  c_re (output[0]) = (tmp3 + tmp6);
  c_im (output[ostride]) = (tmp9 - tmp10);
  c_im (output[3 * ostride]) = (tmp10 + tmp9);
  c_re (output[3 * ostride]) = (tmp11 - tmp14);
  c_re (output[ostride]) = (tmp11 + tmp14);
  c_im (output[2 * ostride]) = (tmp15 - tmp16);
  c_im (output[0]) = (tmp15 + tmp16);
}




void fftw_no_twiddle_simd_4 (FFTW_SIMD_COMPLEX * input,
FFTW_SIMD_COMPLEX * output,
    int istride, int ostride, int ivectstride, int ovectstride)
{
  FFTW_SIMD_VECT tmp3;
  FFTW_SIMD_VECT tmp11;
  FFTW_SIMD_VECT tmp9;
  FFTW_SIMD_VECT tmp15;
  FFTW_SIMD_VECT tmp6;
  FFTW_SIMD_VECT tmp10;
  FFTW_SIMD_VECT tmp14;
  FFTW_SIMD_VECT tmp16;
  {
    FFTW_SIMD_VECT tmp1;
    FFTW_SIMD_VECT tmp2;
    FFTW_SIMD_VECT tmp7;
    FFTW_SIMD_VECT tmp8;
    LOAD_RE_IM (tmp1, tmp7, input + (0), ivectstride);
    LOAD_RE_IM (tmp2, tmp8, input + (2 * istride), ivectstride);
    tmp3 = SIMD_ADD (tmp1, tmp2);
    tmp11 = SIMD_SUB (tmp1, tmp2);
    tmp9 = SIMD_SUB (tmp7, tmp8);
    tmp15 = SIMD_ADD (tmp7, tmp8);
  }
```

```
  {
    FFTW_SIMD_VECT tmp4;
    FFTW_SIMD_VECT tmp5;
    FFTW_SIMD_VECT tmp12;
    FFTW_SIMD_VECT tmp13;
    LOAD_RE_IM (tmp4, tmp12, input + (istride), ivectstride);
    LOAD_RE_IM (tmp5, tmp13, input + (3 * istride), ivectstride);
    tmp6 = SIMD_ADD (tmp4, tmp5);
    tmp10 = SIMD_SUB (tmp4, tmp5);
    tmp14 = SIMD_SUB (tmp12, tmp13);
    tmp16 = SIMD_ADD (tmp12, tmp13);
  }
  STORE_RE_IM (SIMD_SUB (tmp11, tmp14), SIMD_ADD (tmp10, tmp9), output +
    (3 * ostride), ovectstride);
  STORE_RE_IM (SIMD_ADD (tmp11, tmp14), SIMD_SUB (tmp9, tmp10), output +
    (ostride), ovectstride);
  STORE_RE_IM (SIMD_SUB (tmp3, tmp6), SIMD_SUB (tmp15, tmp16), output +
    (2 * ostride), ovectstride);
  STORE_RE_IM (SIMD_ADD (tmp3, tmp6), SIMD_ADD (tmp15, tmp16), output +
    (0), ovectstride);
}


fftw_codelet_desc fftw_no_twiddle_4_desc = {
  "fftw_no_twiddle_4",
  (void (*)()) fftw_no_twiddle_4,
  (void (*)()) fftw_no_twiddle_simd_4,
  (void (*)()) NULL,
  4,
  FFTW_FORWARD,
  FFTW_NOTW,
  89,
  0,
  (const int *) 0,
};
```

B.3 Radix-4 Twiddle Codelets

```
#include "fftw-simd.h" #include <fftw-int.h> #include <fftw.h>

/* Generated by: ./genfft -twiddlesimd 4 */

/*
 * This function contains 22 FP additions, 12 FP multiplications,
 * (or, 16 additions, 6 multiplications, 6 fused multiply/add),
 * 14 stack variables, and 16 memory accesses
 */


/*
```

```
 * Generator Id's :
 * $Id: exprdag.ml,v 1.40 1999/05/17 14:08:53 fftw Exp $
 * $Id: fft.ml,v 1.43 1999/05/17 19:44:18 fftw Exp $
 * $Id: to_c.ml,v 1.24 1999/02/19 17:22:17 athena Exp $
 */

void fftw_twiddle_4 (fftw_complex * A, const fftw_complex * W,
    int iostride, int m, int dist)
{
  int i;
  fftw_complex *inout;
  inout = A;
  for (i = m; i > 0; i = (i - 1), inout = (inout + dist), W = (W + 3))
    {
      fftw_real tmp1;
      fftw_real tmp25;
      fftw_real tmp6;
      fftw_real tmp24;
      fftw_real tmp12;
      fftw_real tmp20;
      fftw_real tmp17;
      fftw_real tmp21;
      tmp1 = c_re (inout[0]);
      tmp25 = c_im (inout[0]);
      {
    fftw_real tmp3;
    fftw_real tmp5;
    fftw_real tmp2;
    fftw_real tmp4;
    tmp3 = c_re (inout[2 * iostride]);
    tmp5 = c_im (inout[2 * iostride]);
    tmp2 = c_re (W[1]);
    tmp4 = c_im (W[1]);
    tmp6 = (tmp2 * tmp3 - tmp4 * tmp5);
    tmp24 = (tmp4 * tmp3 + tmp2 * tmp5);
      }
      {
    fftw_real tmp9;
    fftw_real tmp11;
    fftw_real tmp8;
    fftw_real tmp10;
    tmp9 = c_re (inout[iostride]);
    tmp11 = c_im (inout[iostride]);
    tmp8 = c_re (W[0]);
    tmp10 = c_im (W[0]);
    tmp12 = (tmp8 * tmp9 - tmp10 * tmp11);
    tmp20 = (tmp10 * tmp9 + tmp8 * tmp11);
      }
      {
    fftw_real tmp14;
    fftw_real tmp16;
    fftw_real tmp13;
```

```
    fftw_real tmp15;
    tmp14 = c_re (inout[3 * iostride]);
    tmp16 = c_im (inout[3 * iostride]);
    tmp13 = c_re (W[2]);
    tmp15 = c_im (W[2]);
    tmp17 = (tmp13 * tmp14 - tmp15 * tmp16);
    tmp21 = (tmp15 * tmp14 + tmp13 * tmp16);
      }
      {
    fftw_real tmp7;
    fftw_real tmp18;
    fftw_real tmp27;
    fftw_real tmp28;
    tmp7 = (tmp1 + tmp6);
    tmp18 = (tmp12 + tmp17);
    c_re (inout[2 * iostride]) = (tmp7 - tmp18);
    c_re (inout[0]) = (tmp7 + tmp18);
    tmp27 = (tmp25 - tmp24);
    tmp28 = (tmp12 - tmp17);
    c_im (inout[iostride]) = (tmp27 - tmp28);
    c_im (inout[3 * iostride]) = (tmp28 + tmp27);
      }
      {
    fftw_real tmp23;
    fftw_real tmp26;
    fftw_real tmp19;
    fftw_real tmp22;
    tmp23 = (tmp20 + tmp21);
    tmp26 = (tmp24 + tmp25);
    c_im (inout[0]) = (tmp23 + tmp26);
    c_im (inout[2 * iostride]) = (tmp26 - tmp23);
    tmp19 = (tmp1 - tmp6);
    tmp22 = (tmp20 - tmp21);
    c_re (inout[3 * iostride]) = (tmp19 - tmp22);
    c_re (inout[iostride]) = (tmp19 + tmp22);
      }
    }
}




void fftw_twiddle_simd_4 (FFTW_SIMD_COMPLEX * A,
FFTW_SIMD_COMPLEX * W,
    int iostride, int m, int dist, int iovectstride)
{
  int i;
  FFTW_SIMD_COMPLEX *inout;
  inout = A;
  for (i = m; i > 0; i = (i - 1), inout = (inout + dist), W = (W + 3))
    {
      FFTW_SIMD_VECT tmp1;
```

```
  FFTW_SIMD_VECT tmp25;
  FFTW_SIMD_VECT tmp6;
  FFTW_SIMD_VECT tmp24;
  FFTW_SIMD_VECT tmp12;
  FFTW_SIMD_VECT tmp20;
  FFTW_SIMD_VECT tmp17;
  FFTW_SIMD_VECT tmp21;
  LOAD_RE_IM (tmp1, tmp25, inout + (0), iovectstride);
  {
FFTW_SIMD_VECT tmp3;
FFTW_SIMD_VECT tmp5;
FFTW_SIMD_VECT tmp2;
FFTW_SIMD_VECT tmp4;
LOAD_RE_IM (tmp3, tmp5, inout + (2 * iostride), iovectstride);
LOAD_TWIDDLE_RE_IM (tmp2, tmp4, W + (1));
tmp6 = SIMD_SUB (SIMD_MUL (tmp2, tmp3), SIMD_MUL (tmp4, tmp5));
tmp24 = SIMD_ADD (SIMD_MUL (tmp4, tmp3), SIMD_MUL (tmp2, tmp5));
  }
  {
FFTW_SIMD_VECT tmp9;
FFTW_SIMD_VECT tmp11;
FFTW_SIMD_VECT tmp8;
FFTW_SIMD_VECT tmp10;
LOAD_RE_IM (tmp9, tmp11, inout + (iostride), iovectstride);
LOAD_TWIDDLE_RE_IM (tmp8, tmp10, W + (0));
tmp12 = SIMD_SUB (SIMD_MUL (tmp8, tmp9), SIMD_MUL (tmp10, tmp11));
tmp20 = SIMD_ADD (SIMD_MUL (tmp10, tmp9), SIMD_MUL (tmp8, tmp11));
  }
  {
FFTW_SIMD_VECT tmp14;
FFTW_SIMD_VECT tmp16;
FFTW_SIMD_VECT tmp13;
FFTW_SIMD_VECT tmp15;
LOAD_RE_IM (tmp14, tmp16, inout + (3 * iostride), iovectstride);
LOAD_TWIDDLE_RE_IM (tmp13, tmp15, W + (2));
tmp17 = SIMD_SUB (SIMD_MUL (tmp13, tmp14), SIMD_MUL (tmp15, tmp16));
tmp21 = SIMD_ADD (SIMD_MUL (tmp15, tmp14), SIMD_MUL (tmp13, tmp16));
  }
  {
FFTW_SIMD_VECT tmp7;
FFTW_SIMD_VECT tmp18;
FFTW_SIMD_VECT tmp27;
FFTW_SIMD_VECT tmp28;
FFTW_SIMD_VECT tmp23;
FFTW_SIMD_VECT tmp26;
FFTW_SIMD_VECT tmp19;
FFTW_SIMD_VECT tmp22;
tmp7 = SIMD_ADD (tmp1, tmp6);
tmp18 = SIMD_ADD (tmp12, tmp17);
tmp27 = SIMD_SUB (tmp25, tmp24);
tmp28 = SIMD_SUB (tmp12, tmp17);
tmp23 = SIMD_ADD (tmp20, tmp21);
```

```
    tmp26 = SIMD_ADD (tmp24, tmp25);
    STORE_RE_IM (SIMD_ADD (tmp7, tmp18), SIMD_ADD (tmp23, tmp26),
        inout + (0), iovectstride);
    STORE_RE_IM (SIMD_SUB (tmp7, tmp18), SIMD_SUB (tmp26, tmp23),
        inout + (2 * iostride), iovectstride);
    tmp19 = SIMD_SUB (tmp1, tmp6);
    tmp22 = SIMD_SUB (tmp20, tmp21);
    STORE_RE_IM (SIMD_SUB (tmp19, tmp22), SIMD_ADD (tmp28, tmp27),
        inout + (3 * iostride), iovectstride);
    STORE_RE_IM (SIMD_ADD (tmp19, tmp22), SIMD_SUB (tmp27, tmp28),
        inout + (iostride), iovectstride);
      }
    }
}


#define FFTW_TWIDDLE_STRIDE_4 3


void fftw_twiddle_simd_int_4 (FFTW_SIMD_COMPLEX * A,
FFTW_SIMD_COMPLEX * W,
    int iostride, int m, int dist)
{
  int i;
  FFTW_SIMD_COMPLEX *inout;
  inout = A;
  m >>= FFTW_LD_SIMD_LEN;
  for (i = m; i > 0; i = (i - 1), inout = (inout + FFTW_SIMD_LEN * dist),
    W = (W + FFTW_TWIDDLE_STRIDE_4 * FFTW_SIMD_LEN))
    {
      FFTW_SIMD_VECT tmp1;
      FFTW_SIMD_VECT tmp25;
      FFTW_SIMD_VECT tmp6;
      FFTW_SIMD_VECT tmp24;
      FFTW_SIMD_VECT tmp12;
      FFTW_SIMD_VECT tmp20;
      FFTW_SIMD_VECT tmp17;
      FFTW_SIMD_VECT tmp21;
      LOAD_RE_IM (tmp1, tmp25, inout + (0), dist);
      {
    FFTW_SIMD_VECT tmp3;
    FFTW_SIMD_VECT tmp5;
    FFTW_SIMD_VECT tmp2;
    FFTW_SIMD_VECT tmp4;
    LOAD_RE_IM (tmp3, tmp5, inout + (2 * iostride), dist);
    LOAD_RE_IM (tmp2, tmp4, W + (1), FFTW_TWIDDLE_STRIDE_4);
    tmp6 = SIMD_SUB (SIMD_MUL (tmp2, tmp3), SIMD_MUL (tmp4, tmp5));
    tmp24 = SIMD_ADD (SIMD_MUL (tmp4, tmp3), SIMD_MUL (tmp2, tmp5));
      }
      {
    FFTW_SIMD_VECT tmp9;
    FFTW_SIMD_VECT tmp11;
```

```
    FFTW_SIMD_VECT tmp8;
    FFTW_SIMD_VECT tmp10;
    LOAD_RE_IM (tmp9, tmp11, inout + (iostride), dist);
    LOAD_RE_IM (tmp8, tmp10, W + (0), FFTW_TWIDDLE_STRIDE_4);
    tmp12 = SIMD_SUB (SIMD_MUL (tmp8, tmp9), SIMD_MUL (tmp10, tmp11));
    tmp20 = SIMD_ADD (SIMD_MUL (tmp10, tmp9), SIMD_MUL (tmp8, tmp11));
      }
      {
    FFTW_SIMD_VECT tmp14;
    FFTW_SIMD_VECT tmp16;
    FFTW_SIMD_VECT tmp13;
    FFTW_SIMD_VECT tmp15;
    LOAD_RE_IM (tmp14, tmp16, inout + (3 * iostride), dist);
    LOAD_RE_IM (tmp13, tmp15, W + (2), FFTW_TWIDDLE_STRIDE_4);
    tmp17 = SIMD_SUB (SIMD_MUL (tmp13, tmp14), SIMD_MUL (tmp15, tmp16));
    tmp21 = SIMD_ADD (SIMD_MUL (tmp15, tmp14), SIMD_MUL (tmp13, tmp16));
      }
      {
    FFTW_SIMD_VECT tmp7;
    FFTW_SIMD_VECT tmp18;
    FFTW_SIMD_VECT tmp27;
    FFTW_SIMD_VECT tmp28;
    FFTW_SIMD_VECT tmp23;
    FFTW_SIMD_VECT tmp26;
    FFTW_SIMD_VECT tmp19;
    FFTW_SIMD_VECT tmp22;
    tmp7 = SIMD_ADD (tmp1, tmp6);
    tmp18 = SIMD_ADD (tmp12, tmp17);
    tmp27 = SIMD_SUB (tmp25, tmp24);
    tmp28 = SIMD_SUB (tmp12, tmp17);
    tmp23 = SIMD_ADD (tmp20, tmp21);
    tmp26 = SIMD_ADD (tmp24, tmp25);
    STORE_RE_IM (SIMD_ADD (tmp7, tmp18), SIMD_ADD (tmp23, tmp26),
        inout + (0), dist);
    STORE_RE_IM (SIMD_SUB (tmp7, tmp18), SIMD_SUB (tmp26, tmp23),
        inout + (2 * iostride), dist);
    tmp19 = SIMD_SUB (tmp1, tmp6);
    tmp22 = SIMD_SUB (tmp20, tmp21);
    STORE_RE_IM (SIMD_SUB (tmp19, tmp22), SIMD_ADD (tmp28, tmp27),
        inout + (3 * iostride), dist);
    STORE_RE_IM (SIMD_ADD (tmp19, tmp22), SIMD_SUB (tmp27, tmp28),
        inout + (iostride), dist);
      }
    }
}


static const int twiddle_order[] = {1, 2, 3}; fftw_codelet_desc
fftw_twiddle_4_desc = {
  "fftw_twiddle_4",
  (void (*)()) fftw_twiddle_4,
  (void (*)()) fftw_twiddle_simd_4,
```

```
  (void (*)()) fftw_twiddle_simd_int_4,
  4,
  FFTW_FORWARD,
  FFTW_TWIDDLE,
  88,
  3,
  twiddle_order,
};
```

B.4 fftw.h

```
/* FF SIMD */
#define FFTW_ENABLE_FLOAT
#define FFTW_ENABLE_VECTOR_RECURSE
#define USE_VECT_LOOP
#define USE_VECT_LOOP_INT
/* end FF SIMD */

/*  SIMD Codelets FF        */
typedef void (fftw_notw_codelet_simd)
     (FFTW_SIMD_COMPLEX *, FFTW_SIMD_COMPLEX *, int, int, int, int);
typedef void (fftw_twiddle_codelet_simd)
     (FFTW_SIMD_COMPLEX *, const FFTW_SIMD_COMPLEX *, int,
      int, int, int);
typedef void (fftw_twiddle_codelet_simd_int)
     (FFTW_SIMD_COMPLEX *, const FFTW_SIMD_COMPLEX *, int,
      int,int);
/*  end SIMD Codelets FF    */

/* description of a codelet */
typedef struct {
     const char *name;      /* name of the codelet */
     void (*codelet) ();    /* pointer to the codelet itself */
/*  SIMD Codelet FF         */
     void (*vectcodelet) ();
     void (*vectintcodelet) ();
/*  end SIMD Codelet FFt    */
     int size;          /* size of the codelet */
     fftw_direction dir;    /* direction */
     enum fftw_node_type type;  /* TWIDDLE or NO_TWIDDLE */
     int signature;     /* unique id */
     int ntwiddle;      /* number of twiddle factors */
     const int *twiddle_order;  /*
                 * array that determines the order
                 * in which the codelet expects
                 * the twiddle factors
                 */
} fftw_codelet_desc;

/* structure that holds all the data needed for a given step */
```

```
typedef struct fftw_plan_node_struct {
     enum fftw_node_type type;
     /* SIMD Vectorization FF       */
     union {
      /* nodes of type FFTW_NOTW */
      struct {
           int size;
           fftw_notw_codelet *codelet;
           fftw_notw_codelet_simd *vectcodelet;
           const fftw_codelet_desc *codelet_desc;
      } notw;

      /* nodes of type FFTW_TWIDDLE */
      struct {
           int size;
           fftw_twiddle_codelet *codelet;
           fftw_twiddle_codelet_simd *vectcodelet;
           fftw_twiddle_codelet_simd_int *vectintcodelet;
           fftw_twiddle *tw;
           struct fftw_plan_node_struct *recurse;
           const fftw_codelet_desc *codelet_desc;
      } twiddle;
      /* SIMD Vectorization FF end  */

     int refcnt;
} fftw_plan_node;
```

B.5 executor.c

```
#ifdef FFTW_ENABLE_VECTOR_RECURSE

/* executor_many_vector is like executor_many, but it pushes the
   howmany loop down to the leaves of the transform: */
static void executor_many_vector(int n, const fftw_complex *in,
                 fftw_complex *out,
                 fftw_plan_node *p,
                 int istride,
                 int ostride,
                 int howmany, int idist, int odist)
{
     int s;

     switch (p->type) {
     case FFTW_NOTW:
          {
           fftw_notw_codelet *codelet = p->nodeu.notw.codelet;
           /*   SIMD vectorization FF       */
           fftw_notw_codelet_simd *vectcodelet=p->nodeu.notw.vectcodelet;

           HACK_ALIGN_STACK_ODD;
```

```
           for (s = 0; s < (howmany/4); ++s)
            vectcodelet(((FFTW_SIMD_COMPLEX *)in) + 4 * s * idist,
                ((FFTW_SIMD_COMPLEX *)out) + 4 * s * odist,
                istride, ostride,idist,odist);

           for (s = (howmany - (howmany%4)); s < howmany; ++s)
            codelet(in + s * idist,
                out + s * odist,
                istride, ostride);
           break;
          }

     case FFTW_TWIDDLE:
          {
#ifdef USE_VECT_LOOP
           int r = p->nodeu.twiddle.size;
           int m = n / r;
           int m1=(m/4)<<2,m2=m%4;

           fftw_twiddle_codelet *codelet;
           fftw_twiddle_codelet_simd_int *vectintcodelet;
           fftw_complex *W;

           for (s = 0; s < r; ++s)
            executor_many_vector(m, in + s * istride,
                         out + s * (m * ostride),
                         p->nodeu.twiddle.recurse,
                         istride * r, ostride,
                         howmany, idist, odist);

           codelet = p->nodeu.twiddle.codelet;
           vectintcodelet = p->nodeu.twiddle.vectintcodelet;
           W = p->nodeu.twiddle.tw->twarray;

           /* This may not be the right thing.  We maybe should have
              the howmany loop for the twiddle codelets at the
              topmost level of the recursion, since odist is big;
              i.e. separate recursions for twiddle and notwiddle. */
           HACK_ALIGN_STACK_EVEN;
           for (s = 0; s < howmany; ++s)
           {
            vectintcodelet(((FFTW_SIMD_COMPLEX *)out) + s * odist,
                (FFTW_SIMD_COMPLEX *) W, m * ostride, m1, ostride);
            if (m2) codelet(out + s * odist+m1*ostride,
                W+m1*(p->nodeu.twiddle.size-1), m * ostride, m2, ostride);
           }


           break;
#else
           int r = p->nodeu.twiddle.size;
```

```
           int m = n / r;
           fftw_twiddle_codelet *codelet;
           fftw_twiddle_codelet_simd *vectcodelet;
           fftw_complex *W;

           for (s = 0; s < r; ++s)
            executor_many_vector(m, in + s * istride,
                        out + s * (m * ostride),
                        p->nodeu.twiddle.recurse,
                        istride * r, ostride,
                        howmany, idist, odist);

           codelet = p->nodeu.twiddle.codelet;
           vectcodelet = p->nodeu.twiddle.vectcodelet;
           W = p->nodeu.twiddle.tw->twarray;

           /* This may not be the right thing.  We maybe should have
              the howmany loop for the twiddle codelets at the
              topmost level of the recursion, since odist is big;
              i.e. separate recursions for twiddle and notwiddle. */
           HACK_ALIGN_STACK_EVEN;

           for (s = 0; s < (howmany/4); ++s)
            vectcodelet(((FFTW_SIMD_COMPLEX *)out) + 4 * s * odist,
                (FFTW_SIMD_COMPLEX *) W, m * ostride, m, ostride, odist);
           for (s = (howmany - (howmany%4)); s < howmany; ++s)
            codelet(out + s * odist, W, m * ostride, m, ostride);

           break;
#endif
           /*   end SIMD vectorization FF   */
          }

    default:
          fftw_die("BUG in executor: invalid plan\n");
          break;
     }
}

#endif /* FFTW_ENABLE_VECTOR_RECURSE */

/*
 * Do *not* declare simple executor static--we need to call it
 * from other files...also, preface its name with "fftw_"
 * to avoid any possible name collisions.
 */
void fftw_executor_simple(int n, const fftw_complex *in,
              fftw_complex *out,
              fftw_plan_node *p,
              int istride,
              int ostride,
              fftw_recurse_kind recurse_kind)
```

```
{
     switch (p->type) {
     case FFTW_NOTW:
          HACK_ALIGN_STACK_ODD;
          (p->nodeu.notw.codelet)(in, out, istride, ostride);
          break;

     case FFTW_TWIDDLE:
          {
           int r = p->nodeu.twiddle.size;
           int m = n / r;
/*  FF SIMD vectorization begin     */
           int m1=(m/4)<<2,m2=m%4;

           fftw_twiddle_codelet *codelet;
           fftw_twiddle_codelet_simd_int *vectintcodelet;
           fftw_complex *W;

#ifdef FFTW_ENABLE_VECTOR_RECURSE
           if (recurse_kind == FFTW_NORMAL_RECURSE) {
#endif
            executor_many(m, in, out,
                      p->nodeu.twiddle.recurse,
                      istride * r, ostride,
                      r, istride, m * ostride,
                      FFTW_NORMAL_RECURSE);
           codelet = p->nodeu.twiddle.codelet;
           W = p->nodeu.twiddle.tw->twarray;

           HACK_ALIGN_STACK_EVEN;
           codelet(out, W, m * ostride, m, ostride);
#ifdef FFTW_ENABLE_VECTOR_RECURSE
           }
           else {
            executor_many_vector(m, in, out,
                         p->nodeu.twiddle.recurse,
                         istride * r, ostride,
                         r, istride, m * ostride);
           codelet=p->nodeu.twiddle.codelet;
           W = p->nodeu.twiddle.tw->twarray;

#ifdef USE_VECT_LOOP_INT
           vectintcodelet = p->nodeu.twiddle.vectintcodelet;
           HACK_ALIGN_STACK_EVEN;
                vectintcodelet((FFTW_SIMD_COMPLEX *)out,
           (FFTW_SIMD_COMPLEX *)W, m * ostride, m, ostride);
           if (m2) codelet(out + m1*ostride, W+m1*(p->nodeu.twiddle.size-1),
                m * ostride, m2, ostride);
#else
           HACK_ALIGN_STACK_EVEN;
           codelet(out, W, m * ostride, m, ostride);
#endif
```

```
           }
#endif /*  FF SIMD vectorization end       */
           break;
          }

     default:
          fftw_die("BUG in executor: invalid plan\n");
          break;
     }
}

void fftw_one(fftw_plan plan, fftw_complex *in, fftw_complex *out)
{
     int n = plan->n;

     if (plan->flags & FFTW_IN_PLACE)
      executor_simple_inplace(n, in, out, plan->root, 1,
                  plan->recurse_kind);
     else
      fftw_executor_simple(n, in, out, plan->root, 1, 1,
                   plan->recurse_kind);
}
```

B.6 planner.c

```
static fftw_plan planner_normal(fftw_plan *table, int n,
fftw_direction dir,
                int flags, int vector_size,
                fftw_complex *in, int istride,
                fftw_complex *out, int ostride)
{
     fftw_plan best = (fftw_plan) 0;
     fftw_plan newplan;
     fftw_plan_node *node;

     /* see if we have any codelet that solves the problem */
     {
      FOR_ALL_CODELETS(p) {
           if (p->dir == dir && p->type == FFTW_NOTW) {
            if (p->size == n) {
             node = fftw_make_node_notw(n, p);
             newplan = fftw_make_plan(n, dir, node, flags,
                          p->type, p->signature,
                          FFTW_NORMAL_RECURSE,
                          vector_size);
             fftw_use_plan(newplan);
             compute_cost(newplan, in, istride, out, ostride);
```

```
             run_plan_hooks(newplan);
             best = fftw_pick_better(newplan, best);
            }
           }
      }
     }

     /* Then, try all available twiddle codelets */
     {
      FOR_ALL_CODELETS(p) {
           if (p->dir == dir && p->type == FFTW_TWIDDLE) {
            if ((n % p->size) == 0 &&
            p->size > 1 &&
            (!best || n != p->size)) {
             fftw_plan r = planner(table, n / p->size, dir,
                           flags | FFTW_NO_VECTOR_RECURSE,
                           vector_size,
                           in, istride, out, ostride);
             node = fftw_make_node_twiddle(n, p,
                                r->root, flags);
             newplan = fftw_make_plan(n, dir, node, flags,
                         p->type, p->signature,
                                     FFTW_NORMAL_RECURSE,
                         vector_size);
             fftw_use_plan(newplan);
             /* FF SIMD Vectorization       */
             if (flags&FFTW_USE_SIMD)
                 newplan->recurse_kind=FFTW_VECTOR_RECURSE;
             /* FF end SIMD Vectorization   */
             fftw_destroy_plan_internal(r);
             compute_cost(newplan, in, istride, out, ostride);
             run_plan_hooks(newplan);
             best = fftw_pick_better(newplan, best);
            }
           }
      }
     }

     return best;
}
```

B.7 putils.c

```
fftw_plan_node *fftw_make_node_notw(int size, const
fftw_codelet_desc *config) {
     fftw_plan_node *p = fftw_make_node();

     p->type = config->type;
     p->nodeu.notw.size = size;
     p->nodeu.notw.codelet = (fftw_notw_codelet *) config->codelet;
```

```
     /* enable SIMD vectorization FF        */
     p->nodeu.notw.vectcodelet =
        (fftw_notw_codelet_simd *) config->vectcodelet;
     /* end enable SIMD vectorization FF    */
     p->nodeu.notw.codelet_desc = config;
     return p;
}

fftw_plan_node *fftw_make_node_twiddle(int n,
                      const fftw_codelet_desc *config,
                      fftw_plan_node *recurse,
                      int flags)
{
     fftw_plan_node *p = fftw_make_node();

     p->type = config->type;
     p->nodeu.twiddle.size = config->size;
     p->nodeu.twiddle.codelet = (fftw_twiddle_codelet *) config->codelet;
     /* enable SIMD vectorization FF        */
     p->nodeu.twiddle.vectcodelet =
        (fftw_twiddle_codelet_simd *) config->vectcodelet;
     p->nodeu.twiddle.vectintcodelet =
        (fftw_twiddle_codelet_simd_int *) config->vectintcodelet;
     /* end enable SIMD vectorization FF    */
     p->nodeu.twiddle.recurse = recurse;
     p->nodeu.twiddle.codelet_desc = config;
     fftw_use_node(recurse);
     if (flags & FFTW_MEASURE)
      p->nodeu.twiddle.tw = fftw_create_twiddle(n, config);
     else
      p->nodeu.twiddle.tw = 0;
     return p;
}
```

B.8 wisdom.c

```
void fftw_wisdom_add(int n, int flags, fftw_direction dir,
             enum fftw_wisdom_category category,
             int istride, int ostride,
             enum fftw_node_type type,
             int signature,
             fftw_recurse_kind recurse_kind)
{
     struct wisdom *p;

/*   SIMD Vectorization     */
/*     if ((flags &FFTW_NO_VECTOR_RECURSE) &&
          recurse_kind == FFTW_VECTOR_RECURSE)
          fftw_die("bug in planner (conflicting plan options)\n");
*/
```

```
/*  End SIMD Vectorization  */
     if (!(flags & FFTW_USE_WISDOM))
      return;       /* simply ignore if wisdom is disabled */

     if (!(flags & FFTW_MEASURE))
      return;       /* only measurements produce wisdom */

     if (fftw_wisdom_lookup(n, flags, dir, category, istride, ostride,
                &type, &signature, &recurse_kind, 1))
      return;       /* wisdom overwrote old wisdom */

     p = (struct wisdom *) fftw_malloc(sizeof(struct wisdom));

     p->n = n;
     p->flags = flags;
     p->dir = dir;
     p->category = category;
     p->istride = istride;
     p->ostride = ostride;
     p->type = type;
     p->signature = signature;
     p->recurse_kind = recurse_kind;

     /* remember this wisdom */
     p->next = wisdom_list;
     wisdom_list = p;
}
```

References

[1] M. Auer, *FMA Optimized FFT Algorithms*, Institute for Applied and Numerical Mathematics, Technical University of Vienna, 1998.

[2] M. Auer, R. Benedik, F. Franchetti, H. Karner, P. Kristöfel, R. Schachinger, A. Slateff, C. W. Ueberhuber, *Performance Evaluation of FFT Routines—Machine Independent Serial Programs*, AURORA Tech. Report TR1999-05, Institute for Applied and Numerical Mathematics, Technical University of Vienna, 1999.

[3] M. Auer, F. Franchetti, H. Karner, C. W. Ueberhuber, *Performance Evaluation of FFT Algorithms Using Performance Counters*, AURORA Tech. Report TR1998-20, Institute for Applied and Numerical Mathematics, Technical University of Vienna, 1998.

[4] H. Karner, M. Auer, C. W. Ueberhuber, *Accelerating FFTW by Multiply-Add Optimization*, AURORA Tech. Report TR1999-13, Institute for Applied and Numerical Mathematics, Technical University of Vienna, 1999.

[5] M. Frigo, S. Johnson, *FFTW: An Adaptive Software Architecture for the FFT*, Proceedings of the ICASSP Conference, 1998, Vol. 3, p. 1381.

[6] J. W. Cooley, J. W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Math. Comp. 19 (1965), pp. 297–301.

[7] M. C. Pease, *An Adaption of the Fast Fourier Transform for Parallel Processing*, Journal of the ACM 15 (1968), pp. 252–264.

[8] J. J. Dongarra, E. Grosse, *Distribution of Mathematical Software Via Electronic Mail*, Comm. ACM 30 (1987), pp. 403–407.

[9] C. W. Ueberhuber, *Numerical Computation*, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1997.

[10] U. Baum, M. Clausen, *Some Lower and Upper Complexity Bounds for Generalized Fourier Transforms and Their Inverses*, SIAM J. Comput. 2 (1991), 451–459.

[11] R. C. Agarwal, *A Vector and Parallel Implementation of the FFT Algorithm on the IBM 3090*, in "Aspects of Computation on Asynchronous Parallel Processors; Proceedings of the IFIP WG 2.5 Working Conference" (M. Wright, Ed.), North-Holland, Amsterdam, 1989, pp. 45–54.

[12] C. Temperton, *Implementation of a Prime Factor FFT Algorithm on CRAY-1*, Parallel Comput. 6 (1988), pp. 99–108.

[13] J. Johnson, R. W. Johnson, D. Rodriguez, R. Tolimieri, *A Methodology for Designing, Modifying, and Implementing FFT Algorithms on Various Architectures*, Circuits Systems Signal Process. 9 (1990), pp. 449–500.

[14] A. Norton, A. J. Silberger, *Parallelization and Performance Analysis of the Cooley-Tukey FFT Algorithm for Shared-Memory Architectures*, IEEE Trans. Comput. 36 (1987), pp. 581–591.

[15] N. P. Pitsianis, *The Kronecker Product in Optimization and Fast Transform Generation*, PhD Thesis, Department of Computer Science, Cornell University, 1997.

[16] P. N. Swarztrauber, *FFT Algorithms for Vector Computers*, Parallel Comput. 1 (1984), pp. 45–63.

[17] C. F. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM Press, Philadelphia, 1992.

[18] Intel Corporation, *Intel® C/C++ Compiler User's Guide — With Spport for the Streaming SIMD Extensions*, 1999.

[19] Intel Corporation, *Intel Architecture Software Developer's Manual — Volume 1: Basic Architecture*, 1999.

[20] Intel Corporation, *Intel Architecture Software Developer's Manual — Volume 2: Instruction Set Reference*, 1999.

[21] Intel Corporation, *Intel Architecture Software Developer's Manual — Volume 3: System Programming*, 1999.

[22] Intel Corporation, *Intel® Architecture Optimization — Reference Manual*, 1999.

[23] Intel Corporation, *AP-833 Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel® C/C++ Compiler*, 1999.

[24] Intel Corporation, *AP-931 Streaming SIMD Extensions — LU Decomposition*, 1999.

[25] Intel Corporation, *AP-808 Split Radix Fast Fourier Transform Using Streaming SIMD Extensions*, 1999.

[26] Motorola Corporation, *AltiVec™ Technology Programming Environments Manual*, 1998.

[27] Motorola Corporation, *AltiVec^{TM} Technology Programming Interface Manual*, 1998.