Mathematical Models and Methods in Applied Sciences

MULTIPLY-ADD OPTIMIZED FFT KERNELS

HERBERT KARNER, MARTIN AUER and CHRISTOPH W. UEBERHUBER
Institute for Applied and Numerical Mathematics
Technical University Vienna
Wiedner Hauptstrasse 8–10/115, A–1040 Vienna, Austria
Tel.: +43/1/58801 11512, Fax: +43/1/58801 11599.
karner@titania.tuwien.ac.at martin@auer.net christof@uranus.tuwien.ac.at

Modern computer architecture provides a special instruction—the *fused multiply-add* (FMA) instruction—to perform both a multiplication and an addition operation at the same time. In this paper newly developed radix-2, radix-3, and radix-5 FFT kernels that efficiently take advantage of this powerful instruction are presented. If a processor is provided with FMA instructions, the radix-2 FFT algorithm introduced has the lowest complexity of all Cooley-Tukey radix-2 algorithms. All floating-point operations are executed as FMA instructions. Compared to conventional radix-3 and radix-5 kernels the new radix-3 and radix-5 kernels greatly improve the utilization of FMA instructions resulting in a significant complexity reduction. In general, the advantages of the FFT algorithms presented in this paper are their low complexity, their high efficiency, and their striking simplicity. Numerical experiments show that FFT programs using the new kernels clearly outperform conventional FFT routines, even the best available FFT programs.

Keywords: DFT, FFT, Kronecker products, arithmetic complexity, fused multiply-add, multiply-add optimized algorithms

1. Introduction

The *fast Fourier transform* (FFT) is one of the principal algorithmic tools in the field of scientific computing. The FFT has such a large number of applications that it is not an overstatement to call it ubiquitous.

Though the invention of the FFT algorithm can be traced back to Gauss, the rediscovery by Cooley and Tukey in their 1965 paper[1] is responsible for the algorithm's widespread use.

Numerous studies have been published on how the FFT can be implemented efficiently on advanced computer systems. The first step was made by Pease[14] in 1968. In his pioneering paper, which was not based on the Cooley-Tukey approach, Pease used a Kronecker product formulation to describe an FFT algorithm well suited for implementations on parallel computers.

By algebraically manipulating Kronecker product formulas, different FFT pro-

grams 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 (Karner, Ueberhuber[7], Krommer, Ueberhuber[10]). Therefore Kronecker products are used in the present paper as a fundamental tool for the development and description of FFT algorithms.

Modern computer 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.

In this paper radix-2, radix-3, and radix-5 FFT kernels that efficiently take advantage of this powerful instruction are presented. If a processor is provided with FMA instructions, the radix-2 FFT algorithm introduced has the lowest complexity of all Cooley-Tukey radix-2 algorithms. All floating-point operations are executed as FMA instructions. Compared to conventional radix-3 and radix-5 kernels the new radix-3 and radix-5 kernels greatly improve the usage of FMA instructions resulting in a significant complexity reduction.

Numerical experiments show that FFT programs using the new kernels clearly outperform conventional FFT routines, even the most widely used and most efficient FFT packages.

Another advantage of the FFT kernels presented in this paper is the fact that these new kernels are fully "compatible" with conventional kernels, making it easy to incorporate them into existing FFT frameworks.

2. Arithmetic Complexity

Counting the number of necessary computational steps is an abstract method of assessing the resource consumption of an algorithm. In spite of the abstraction (using computational steps instead of time measurement), a complexity analysis is not independent of particular hardware properties. There may be significant differences in the complexity indices obtained for the very same algorithm due to the fact that for each computer the elementary unit of work may be different. In the field of numerical data processing, it is common to regard a floating-point operation as the elementary unit of work. The evaluation of a discrete Fourier transform, i. e., the evaluation of a particular matrix-vector product involves only addition and multiplication operations. The following definitions are therefore restricted to addition and multiplication operations.

Definition 1 (Complex Arithmetic Complexity) *Let $\mu_C \in \mathbb{N}$ ($\alpha_C \in \mathbb{N}$) denote the number of complex multiplication (addition) operations needed to perform a specific numerical computation. Then, the complex arithmetic complexity is defined by $\pi_C := \mu_C + \alpha_C$.*

Definition 2 (Real Arithmetic Complexity) *Let $\mu_R \in \mathbb{N}$ ($\alpha_R \in \mathbb{N}$) denote the number of real multiplication (addition) operations needed to perform a specific numerical computation. Then, the real arithmetic complexity is defined by $\pi_R := \mu_R + \alpha_R$.*

A nontrivial complex multiplication needs 4 real multiplications and 2 real additions ("4 + 2 *method*") and a complex addition needs 2 real additions, thus a connection between complex and real complexity numbers can be established: $\mu_R = 4\mu_C$, $\alpha_R = 2\mu_C + 2\alpha_C$, and $\pi_R = 6\mu_C + 2\alpha_C$.

Multiply-Add Complexity. In addition to instructions for conventional unary and binary floating-point operations, such as addition, subtraction, multiplication, division, floating-point to integer conversion, negation, absolute value, and comparison, modern RISC processors (and high-end PC processors) provide *fused multiply-add* (FMA) instructions that perform the ternary operation $\pm a \pm b \times c$ in the same amount of time needed for a single floating-point addition or multiplication. FMA instructions have significant performance consequences. With a decoding rate of one instruction per clock cycle, the peak throughput is two floating-point operations per cycle for FMA instructions. For individual add and multiply instructions, it is only one floating-point operation per clock cycle. For processors in FMA architecture it is reasonable to regard the more complex multiply-add operation as another elementary unit of work.

On some modern computer processors, e. g., PowerPC processors, there is no intermediate rounding operation between the multiply and the add. Because the result of the multiplication is not rounded prior to the add, the full precision of the product is kept.

On a processor in FMA architecture a complex multiplication requires 4 instructions (2 multiplications and 2 multiply-adds) and a complex addition requires 2 multiply-add instructions. π_{fma} denotes the *FMA arithmetic complexity*, i. e., the number of multiply-add operations needed to perform a specific numerical task. It holds that

$$\pi_{\text{fma}} \geq \max(\alpha_R, \mu_R). \tag{2.1}$$

To characterize the degree to which an algorithm takes advantage of multiply-add instructions, it is useful to introduce the term FMA utilization.

Definition 3 (FMA Utilization) *The FMA utilization is given by*

$$F := \frac{\pi_R - \pi_{\text{fma}}}{\pi_{\text{fma}}}\ 100\ [\%]. \tag{2.2}$$

The FMA utilization can be used to express what percentage of the floating-point operations is performed by multiply-add instructions.

Cooley-Tukey FFT algorithms require more real additions than real multiplications. Therefore, in conventional FFT programs it is not possible to schedule all of the multiplications so that they appear as genuine multiply-add operations. The number of multiply-add operations always exceeds the number of real additions, i. e., $\pi_{\text{fma}} > \alpha_R$. An inferior FMA utilization is unavoidable.

Related Work. In 1993 Linzer and Feig[11 12] introduced *scaled* radix-2, radix-4, and split-radix FFT algorithms with $\pi_{\text{fma}} = \alpha_R$. In these algorithms scaled

twiddle-factors

$$\widetilde{\omega_N^k} := \omega_N^k / \alpha_{N,k} \tag{2.3}$$

are used. The scaling by means of

$$\alpha_{N,k} := \max(|\cos(2\pi k/N)|, |\sin(2\pi k/N)|) \tag{2.4}$$

brings about that either the real or the imaginary part of $\widetilde{\omega_N^k}$ is normalized such that it has absolute value 1. Thus every complex multiplication by $\widetilde{\omega_N^k}$ is accomplished by two genuine multiply-add operations.

Disadvantages of scaled FFT algorithms are their high program complexity[12] and their high computational effort. Increased run times are caused by the division operations which are 10–30 times slower than addition or multiplication operations.[18] Division operations cause a stall of the floating-point pipeline of RISC processors and—if possible—should be avoided.

In 1997 Goedecker[3] published scaled radix-2, radix-3, radix-4, and radix-5 FFT kernels with a lower program complexity than the algorithms proposed by Linzer and Feig.[11 12]

In contrast to the algorithms introduced in this paper, Goedecker's kernels require extra load operations and need additional workspace. The necessity for time-consuming division operations, needed to scale the twiddle-factors, makes Goedecker's FFT kernels unsuitable for FFT programs using on-line computed twiddle-factors. Additionally, to avoid division by zero exceptions, cautious handling of trivial twiddle-factors is required.

3. Kronecker Products

The use of Kronecker products offers a unifying basis for the description of FFT algorithms. Van Loan[19] used Kronecker product formulations in his 1992 state of the art presentation of FFT algorithms. In the twenty-five years between the publications of Pease[14] and Van Loan[19], only a few authors used this powerful technique: Temperton[17] and Johnson et al.[6] for FFT implementations on classic vector computers and Norton and Silberger[13] on parallel computers with MIMD architecture. Recently, Gupta[4] and Pitsianis[15] used Kronecker product formulations to synthesize FFT programs.

The Kronecker product approach makes it easy to modify FFT algorithms 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.

Definition 4 (Kronecker Product) *The Kronecker product (direct product or tensor 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*

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

Kronecker products have the following algebraic properties (Horn, Johnson[5]).

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.

Mixed-Product Property. If A, B, C, D are arbitrary matrices for which the products AC and BD are defined, then

$$(A \otimes B)\,(C \otimes D) = A\,C \otimes B\,D. \tag{3.5}$$

4. Stride Permutations

Stride permutations are frequently used tools in Kronecker product representations of FFT algorithms.

Definition 5 (Stride Permutation) *For a vector $x \in \mathbb{C}^{mn}$ the stride permutation L_n^{mn} is defined by*

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

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 components with indices equal to $1 \bmod n$ and so on. The notation L_n^{mn} indicates that the elements of a vector of length mn are loaded into m segments each with stride n.

5. DFT Matrix

The discrete Fourier transform (DFT) is defined by the special matrix-vector product $y = F_N\,x$.

Definition 6 (Discrete Fourier Transform) *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

$$y_k := \sum_{j=0}^{N-1} \omega_N^{kj} x_j, \quad k = 0 : N-1, \tag{5.6}$$

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 *phase-factors* or *twiddle-factors* (Gentleman and Sande[2]). They constitute the elements of the DFT matrix, i. e.,

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

6. Fundamental Factorization

The key idea behind FFT algorithms is to use the divide-and-conquer paradigm. N point DFTs are split into successively smaller DFTs.

Theorem 1 (Fundamental Splitting) (Johnson et al.[6]) *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 leads to the following factorization of the DFT matrix F_{p^n}.

Theorem 2 (Fundamental Single-Radix Factorization) (Johnson et al.[6])

$$F_{p^n} = \Big[\prod_{i=1}^{n} (I_{p^{i-1}} \otimes F_p \otimes I_{p^{n-i}})(I_{p^{i-1}} \otimes T_{p^{n-i}}^{p^{n-i+1}})\Big] R_{p^n}. \tag{6.7}$$

This factorization enables the splitting of 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 which is responsible for the permutation of the input data sequence (see Van Loan[19]).

By using (3.5), 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, (6.7) becomes

$$F_{p^n} = \Big[\prod_{i=1}^{n} (I_{p^{i-1}} \otimes B_{p,p^{n-i+1}})\Big] R_{p^n}. \tag{6.8}$$

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

Algorithm 1 (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 decisive 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 primarily depends on design and implementation of the butterfly kernel.

7. Radix-2 Butterfly Computation

Binary algorithms, in which the number of points is a power of two, are by far the most widely used FFT algorithms. The radix-2 butterfly update can be written as the two-dimensional matrix-vector product

$$\begin{pmatrix} x(j) \\ x(j+L/2) \end{pmatrix} := \begin{pmatrix} 1 & \omega_L^j \\ 1 & -\omega_L^j \end{pmatrix} \begin{pmatrix} x(j) \\ x(j+L/2) \end{pmatrix} \tag{7.9}$$

with $j = 0 : L/2 - 1$. Consequently, $x := B_{2,L} x$ can be implemented as

Algorithm 2 (Radix-2 Butterfly)

do $j = 0 : L/2 - 1$
  $\tau := \omega_L^j x(j + L/2)$
  $x(j + L/2) := x(j) - \tau$
  $x(j) := x(j) + \tau$
end do

Complexity. The elementary radix-2 butterfly computation of Algorithm 2 involves one complex multiplication ($\mu_C = 1$) and two complex additions ($\alpha_C = 2$), i. e., 10 real floating-point operations ($\pi_R = 10$). On a computer with multiply-add architecture the computation involves 2 multiply-add operations, 2 multiplications, and 4 additions. Accordingly $\pi_{\text{fma}} = 8$ and the FMA utilization F is a mere 25 %.

The real arithmetic complexity of the entire radix-2 FFT computation is $\pi_R = 5N \log_2 N$ or, on a computer with FMA architecture, $\pi_{\text{fma}} = 4N \log_2 N$.

FMA Optimized Radix-2 Butterfly Computation. Since

$$\begin{pmatrix} 1 & \omega_L^j \\ 1 & -\omega_L^j \end{pmatrix} = \begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & -\omega_L^j \end{pmatrix} \tag{7.10}$$

the radix-2 butterfly update $x := B_{2,L} x$ can alternatively be implemented as

Algorithm 3 (FMA Optimized Radix-2 Butterfly)

do $j = 0 : L/2 - 1$
$\quad x(j + L/2) := x(j) - \omega_L^j x(j + L/2)$
$\quad x(j) := 2x(j) - x(j + L/2)$
end do

Complexity. Using factorization (7.10) an elementary radix-2 butterfly computation requires 12 real floating-point operations ($\pi_R = 12$). On an FMA processor, Algorithm 3 involves 6 multiply-add operations ($\pi_{\text{fma}} = 6$), leading to the highest possible FMA utilization of 100 %.

The arithmetic complexity of the entire FMA optimized radix-2 FFT algorithm is $\pi_R = 6N \log_2 N$ or, on a computer with FMA processor, $\pi_{\text{fma}} = 3N \log_2 N$ which yields a complexity reduction by 25 %.

Radix-4 and Split-Radix Kernels. The method presented above can be applied to radix-4 and split-radix kernels as well (see Karner et al.[8]). Their arithmetic complexity is shown in Table 1.

On a computer which has FMA instructions, the multiply-add optimized radix-4 butterfly computation requires fewer memory accesses as well as fewer floating-point operations than a conventional Cooley-Tukey radix-4 butterfly update, assuming that the twiddle-factors are pre-computed and stored in an array. If the twiddle-factors are calculated on-line, the advantages of the new algorithms are even greater, since only two twiddle-factors have to be calculated instead of the three twiddle-factors needed by conventional Cooley-Tukey radix-4 butterfly updates.

8. Radix-3 Butterfly Computation

Conventional Cooley-Tukey radix-3 butterfly kernels require 28 real floating-point operations ($\pi_R = 28$) (Temperton[17]). On FMA processors the computation involves 22 floating-point operations ($\pi_{\text{fma}} = 22$), leading to an FMA utilization of only 27 %.

The arithmetic complexity of the entire Cooley-Tukey radix-3 FFT algorithm is $\pi_R = 9.33N \log_3 N$ or, on a computer with FMA processor, $\pi_{\text{fma}} = 7.33N \log_3 N$.

FMA Optimized Radix-3 Butterfly Computation. The radix-3 butterfly update $x := B_{3,L}x$ can be written as

$$\begin{pmatrix} x(j) \\ x(j + L/3) \\ x(j + 2L/3) \end{pmatrix} := F_3 \operatorname{diag}(1, \omega_L^j, \omega_L^{2j}) \begin{pmatrix} x(j) \\ x(j + L/3) \\ x(j + 2L/3) \end{pmatrix} \tag{8.11}$$

with $j = 0 : L/3 - 1$. Utilizing Winograd's convolution[20] (8.11) can be evaluated as follows.

$$\begin{aligned} x(j) &:= \omega_3^0[x(j) + \omega_L^j x(j + L/3) + \omega_L^{2j} x(j + 2L/3)] \\ x(j + L/3) &:= \omega_3^0 x(j) + w_1 \\ x(j + 2L/3) &:= \omega_3^0 x(j) + w_2, \end{aligned} \tag{8.12}$$

where

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} := \begin{pmatrix} \omega_3^1 & \omega_3^2 \\ \omega_3^2 & \omega_3^1 \end{pmatrix} \mathrm{diag}(\omega_L^j, \omega_L^{2j}) \begin{pmatrix} x(j+L/3) \\ x(j+2L/3) \end{pmatrix}. \tag{8.13}$$

The convolution (8.13) can be factorized into

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} := \begin{pmatrix} \frac{\omega_3^1+\omega_3^2}{2} & \\ & \frac{\omega_3^1-\omega_3^2}{2} \end{pmatrix} \begin{pmatrix} 1 & \omega_L^{2j} \\ 1 & -\omega_L^{2j} \end{pmatrix} \mathrm{diag}(\omega_L^j, 1) \begin{pmatrix} x(j+L/3) \\ x(j+2L/3) \end{pmatrix}$$

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} := \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}. \tag{8.14}$$

Let $c_1 := (\omega_3^1+\omega_3^2)/2 = -1/2$, and $c_2 := (\omega_3^1-\omega_3^2)/2 = -i\sqrt{3}/2$. Using factorization

$$\begin{pmatrix} 1 & \omega_L^{2j} \\ 1 & -\omega_L^{2j} \end{pmatrix} = \begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & -\omega_L^{2j} \end{pmatrix}, \tag{8.15}$$

the FMA optimized radix-3 butterfly update can now be calculated using the following algorithm.

Algorithm 4 (FMA Optimized Radix-3 Butterfly)

do $j = 0 : L/3-1$
  $z_1 := \omega_L^j x(j+L/3)$
  $s_1 := z_1 - \omega_L^{2j} x(j+2L/3)$; $s_2 := 2z_1 - s_1$
  $s_3 := s_2 + x(j)$
  $s_4 := x(j) + c_1 s_2$
  $s_5 := s_4 - c_2 s_1$; $s_6 := 2s_4 - s_5$
  $x(j) := s_3$
  $x(j+L/3) := s_6$
  $x(j+2L/3) := s_5$
end do

Complexity. This elementary radix-3 butterfly update involves 32 real floating-point operations ($\pi_R = 32$). On an FMA processor the computation involves 18 floating-point operations ($\pi_{\mathrm{fma}} = 18$), leading to an FMA utilization of 78 %.

The arithmetic complexity of the entire multiply-add optimized radix-3 FFT algorithm is $\pi_R = 10.67N\log_3 N$ or, on a computer with FMA processor, $\pi_{\mathrm{fma}} = 6N\log_3 N$.

9. Radix-5 Butterfly Computation

Conventional Cooley-Tukey radix-5 kernels require 68 real floating-point operations ($\pi_R = 68$) (Temperton[17]). On an FMA processor the computation involves 52 floating-point operations ($\pi_{\mathrm{fma}} = 52$), leading to an FMA utilization of 31 %.

The arithmetic complexity of the entire Cooley-Tukey radix-5 FFT algorithm is $\pi_R = 13.6N\log_5 N$ or, on a computer with FMA processor, $\pi_{\mathrm{fma}} = 10.4N\log_5 N$.

FMA Optimized Radix-5 Butterfly Computation. The radix-5 butterfly update $x := B_{5,L}x$, $x \in \mathbb{C}^L$ can be written as

$$\begin{pmatrix} x(j) \\ x(j+L/5) \\ x(j+2L/5) \\ x(j+3L/5) \\ x(j+4L/5) \end{pmatrix} := F_5 \operatorname{diag}(1, \omega_L^j, \omega_L^{2j}, \omega_L^{3j}, \omega_L^{4j}) \begin{pmatrix} x(j) \\ x(j+L/5) \\ x(j+2L/5) \\ x(j+3L/5) \\ x(j+4L/5) \end{pmatrix}$$

with $j = 0 : L/5 - 1$. Using Winograd's cyclic convolution[20] and a factorization similar to (7.10) the FMA optimized radix-5 butterfly update can be calculated using the following algorithm. (A detailed derivation can be found in Karner et al.[9]).

Algorithm 5 (FMA Optimized Radix-5 Butterfly)

do $j = 0 : L/5 - 1$
$z_0 := x(j)$
$z_1 := \omega_L x(j + L/5)$
$z_2 := \omega_L^{2j} x(j + 2L/5)$
$s_1 := z_1 - \omega_L^{4j} x(j + 4L/5);$ $\quad s_2 := 2z_1 - s_1$
$s_3 := z_2 - \omega_L^{3j} x(j + 3L/5);$ $\quad s_4 := 2z_2 - s_3$
$s_5 := s_2 + s_4;$ $\quad s_6 := s_2 - s_4$
$s_7 := z_0 - c_1 s_5$
$s_8 := s_7 - c_2 s_6;$ $\quad s_9 := 2s_7 - s_8$
$s_{10} := s_1 + c_3 s_3;$ $\quad s_{11} := c_3 s_1 - s_3$
$x(j) := z_0 + s_5;$ $\quad t_1 := s_9 - ic_4 s_{10}$
$x(j + L/5) := 2s_9 - t_1;$ $\quad t_2 := s_8 - ic_4 s_{11}$
$x(j + 2L/5) := 2s_8 - t_2$
$x(j + 3L/5) := t_2$
$x(j + 4L/5) := t_1$
end do

with

$$c_1 = 1/4, \;\; c_2 = \sqrt{5}/4, \;\; c_3 = \sqrt{(5 - \sqrt{5})/(5 + \sqrt{5})}, \;\; c_4 = 1/2\sqrt{(5/2 + \sqrt{5}/2)}.$$

Complexity. This radix-5 butterfly algorithm involves 78 real floating-point operations ($\pi_R = 78$). On an FMA processor the computation involves 44 floating-point operations ($\pi_{\text{fma}} = 44$), leading to an FMA utilization of 77 %.

The arithmetic complexity of the entire multiply-add optimized radix-5 FFT algorithm is $\pi_R = 15.6N \log_5 N$ or, on a computer with FMA processor, $\pi_{\text{fma}} = 8.8N \log_5 N$.

10. Minimum FMA Complexity FFT Algorithms

Since addition operations predominate multiplication operations in the elementary Cooley-Tukey butterfly computation, the number of addition operations required, α_R, yields a lower bound for the achievable FMA complexity.

Radix-2 Butterfly Computation. The elementary Cooley-Tukey radix-2 butterfly update requires $\alpha_R = 6$ additions. Since the multiply-add optimized radix-2 butterfly update also requires $\pi_{\text{fma}} = 6$ FMA operations, Algorithm 3 is *FMA optimal*, i. e., there is no possibility of computing elementary radix-2 butterfly updates with less FMA operations.

Radix-3 and Radix-5 Butterfly Computation. The elementary Cooley-Tukey radix-3 butterfly update requires $\alpha_R = 16$ additions; Algorithm 4 needs $\pi_{\text{fma}} = 18$ FMA operations, i. e., its complexity is 12.5 % higher than the optimum complexity.

The elementary Cooley-Tukey radix-5 butterfly update needs $\alpha_R = 40$ additions; Algorithm 5 needs $\pi_{\text{fma}} = 44$ FMA operations, i. e., 10 % more operations than the minimum.

11. Experimental Results

Numerical experiments were performed on one processor of an SGI Power Challenge XL R10000. In these experiments newly developed FMA optimized radix-3 and radix-5 FFT algorithms were compared with the double precision routine `fftw` from the package FFTW.[a]

Since the radix-2 kernel is memory bound on the SGI Power Challenge it was not included in the numerical experiments.[3]

The multiply-add optimized FFT routines were implemented in C (double precision) using pre-computed twiddle-factors (just like FFTW).

Normalized FMA Arithmetic Complexity. A useful normalization of the FMA arithmetic complexity is given by

$$\overline{\pi}_{\text{fma}} := \frac{\pi_{\text{fma}}}{N \log N},$$

where N denotes the length of the transform.

Normalized values of the FMA arithmetic complexity of FFTPACK[b], FFTW, and FMA optimized programs for radix-3 FFTs are shown in Figure 1, and for radix-5 FFTs in Figure 2. These complexity numbers were measured using the *performance monitor counter* (PMC) of the MIPS R10000 processor, special registers able to count various types of events such as issued and graduated instructions.

Due to the fact that FFTW and FFTPACK avoid multiplications by trivial twiddle-factors, FFTW and FFTPACK achieve lower complexity values for shorter transform lengths than for longer transform lengths. The newly developed FFT routines have not been optimized yet with respect to their handling of trivial twiddle-factors and therefore their normalized arithmetic complexity remains constant.

[a] FFTW is a public domain FFT package available via NETLIB. At present FFTW is the fastest publicly available FFT package.

[b] FFTPACK is a public domain FFT package available via NETLIB. FFTPACK is the most widely used FFT package.

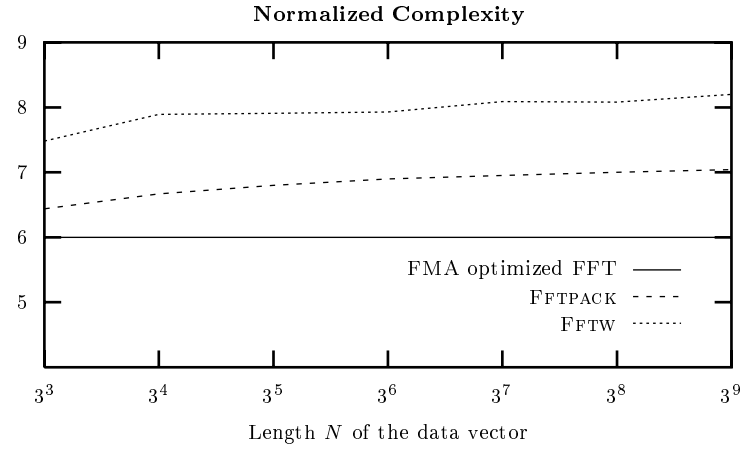

Fig. 1. Normalized FMA arithmetic complexity of radix-3 FFT algorithms

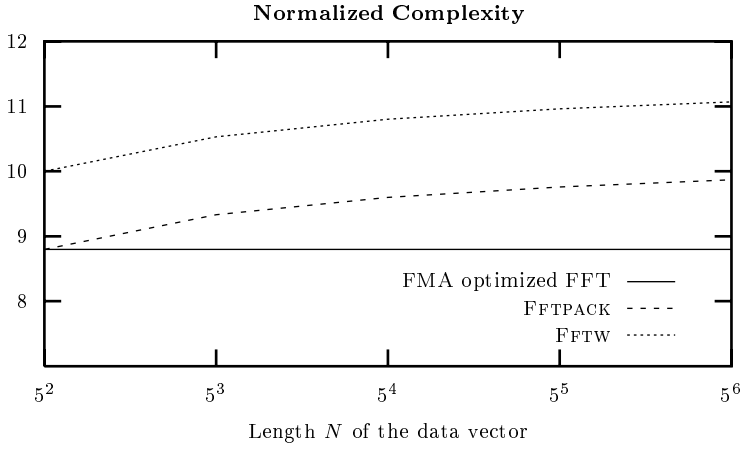

Fig. 2. Normalized FMA arithmetic complexity of radix-5 FFT algorithms

Normalized Execution Time. The normalized execution time is given by

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

where T denotes the run time, and N the length of the transform. Measured values of the normalized execution time are shown in Figures 3 and 4.

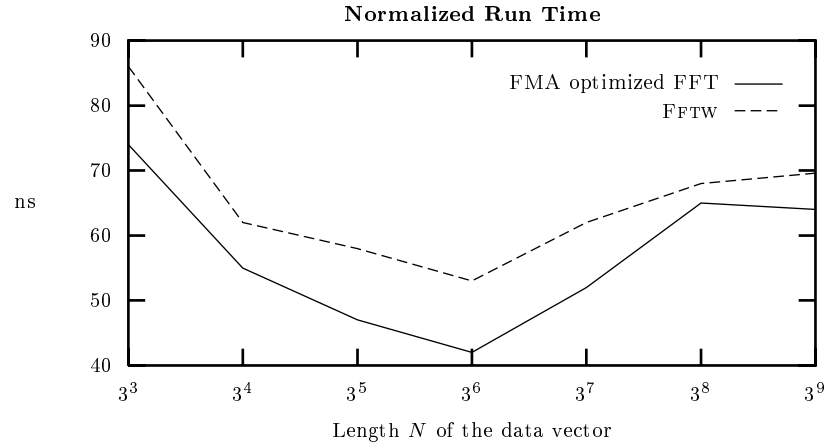

Fig. 3. Normalized execution time of radix-3 FFT algorithms in nanoseconds

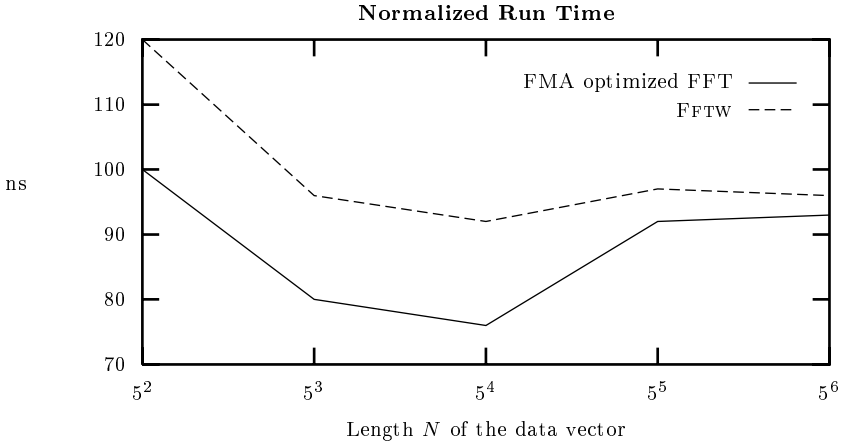

Fig. 4. Normalized execution time of radix-5 FFT algorithms in nanoseconds

Both, radix-3 and radix-5 run time experiments show that as long as the required data can be held in primary cache a considerable speedup can be achieved by using the FMA optimized kernels. If cache misses make memory accesses more expensive, which happens for longer transform lengths, the performance gain of the new kernels becomes smaller.

12. Conclusion

On computer processors with multiply-add capability, i. e., on most high-performance processors, the new multiply-add optimized FFT algorithms have lower operation counts and lower execution times than well-established conventional FFT algorithms.

Table 1. Arithmetic complexity of elementary butterfly updates.

Algorithm		π_R	π_{fma}	FMA Utilization
CT (FFTPACK)	radix-2	10	8	25 %
FFTW	radix-2	10	8	25 %
FMA optimized	radix-2	12	6	100 %
CT (FFTPACK)	radix-4	34	28	21 %
FFTW	radix-4	34	28	21 %
FMA optimized	radix-4	48	24	100 %
CT	split-radix	24	20	20 %
FMA optimized	split-radix	36	18	100 %
CT (FFTPACK)	radix-3	28	22	27 %
FFTW	radix-3	30	24	25 %
FMA optimized	radix-3	32	18	78 %
CT (FFTPACK)	radix-5	68	52	31 %
FFTW	radix-5	78	58	34 %
FMA optimized	radix-5	78	44	77 %

Table 1 shows a comparison between conventional Cooley-Tukey (CT) kernels (used, for instance, by FFTPACK), the kernels used by FFTW, and the new FMA optimized kernels introduced in this paper. Precise information about FMA optimized radix-4 and split-radix FFT algorithms can be found in Karner et al.[8]

To sum up, it can be said that the advantages of the new multiply-add optimized FFT algorithms presented in this paper are their low complexity, their high efficiency on modern computer systems, their striking simplicity, and their effectiveness for utilizing on-line computed twiddle-factors.

Acknowledgment

Particular thanks go to Ronald Benedik and Andreas Slateff who tirelessly and carefully helped us to carry out the computer experiments. In addition, we would like to acknowledge the financial support of the Austrian Science Fund FWF.

References

1. J. W. Cooley and J. W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, *Math. Comp.* **19** (1965) 297–301.
2. W. M. Gentleman and G. Sande, *Fast Fourier Transforms—For Fun and Profit*, in *Proceedings of the 1966 Fall Joint Computer Conference* (AFIPS, 1966), pp. 563–578.

3. S. Goedecker, *Fast Radix 2, 3, 4, and 5 Kernels for Fast Fourier Transformations on Computers with Overlapping Multiply-Add Instructions*, *SIAM J. Sci. Comput* **18** (1997) 1605–1611.
4. S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson, *A Framework for Generating Distributed-Memory Parallel Programs for Block Recursive Algorithms*, *J. Parallel and Distributed Computing* **34** (1996) 137–153.
5. R. A. Horn and C. R. Johnson, **Topics in Matrix Analysis** (Cambridge University Press, 1991).
6. J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, *A Methodology for Designing, Modifying, and Implementing FFT Algorithms on Various Architectures*, *Circuits Systems Signal Process.* **9** (1990) 449–500.
7. H. Karner and C. W. Ueberhuber, *Architecture Adaptive FFT Algorithms*, in *Proceedings of the Second IASTED International Conference on European Parallel and Distributed Systems (Euro-PDS'98)*, eds. O. Bukhres and H. El-Rewini (IASTED/ACTA Press, 1998), pp. 331–334.
8. H. Karner, M. Auer and C. W. Ueberhuber, *Optimum Complexity FFT Algorithms for RISC Processors*, AURORA Tech. Report TR1998-03, Institute for Applied and Numerical Mathematics, Technical University of Vienna (1998).
9. H. Karner, M. Auer and C. W. Ueberhuber, *FMA Optimized FFT Algorithms—An Empirical Performance Study*, Technical Report, Scientific Parallel Computation Group, Technical University of Vienna (1998).
10. A. R. Krommer and C. W. Ueberhuber, *Architecture Adaptive Algorithms*, *Parallel Comput.* **19** (1993) 409–435.
11. E. N. Linzer and E. Feig, *Modified FFTs for Fused Multiply-Add Architectures*, *Math. Comp.* **60** (1993) 347–361.
12. E. N. Linzer and E. Feig, *Implementation of Efficient FFT Algorithms on Fused Multiply-Add Architectures*, *IEEE Trans. Signal Processing* **41** (1993) 93–107.
13. A. Norton and A. J. Silberger, *Parallelization and Performance Analysis of the Cooley-Tukey FFT Algorithm for Shared-Memory Architectures IEEE Trans. Comput.* **36** (1987) 581–591.
14. M. C. Pease, *An Adaptation of the Fast Fourier Transform for Parallel Processing*, *J. ACM* **15** (1968) 252–264.
15. N. P. Pitsianis, *The Kronecker Product in Optimization and Fast Transform Generation*, Department of Computer Science, Cornell University, PhD Thesis (1997).
16. C. Temperton, *Self-Sorting Mixed-Radix Fast Fourier Transforms*, *J. Comput. Phys.* **52** (1983) 1–23.
17. C. Temperton, *Fast Mixed-Radix Real Fourier Transforms*, *J. Comput. Phys.* **52** (1983) 340–350.
18. C. W. Ueberhuber, **Numerical Computation** (Springer-Verlag, 1997).
19. C. F. VanLoan, **Computational Frameworks for the Fast Fourier Transform** (SIAM Press, 1992).
20. S. Winograd, *On Computing the Discrete Fourier Transform*, *Math. Comp.* **32** (1978) 175–199.