SIAM J. SCI. COMPUT.
Vol. 18, No. 6, pp. 1605–1611, November 1997
© 1997 Society for Industrial and Applied Mathematics
005

FAST RADIX 2, 3, 4, AND 5 KERNELS FOR FAST FOURIER TRANSFORMATIONS ON COMPUTERS WITH OVERLAPPING MULTIPLY–ADD INSTRUCTIONS*

S. GOEDECKER†

Abstract. We present a new formulation of fast Fourier transformation (FFT) kernels for radix 2, 3, 4, and 5, which have a perfect balance of multiplies and adds. These kernels give higher performance on machines that have a single multiply–add (mult–add) instruction. We demonstrate the superiority of this new kernel on IBM and SGI workstations.

Key word. FFT kernels

AMS subject classifications. 65-04, 42-04

PII. S1064827595281940

1. Introduction.

Until several years ago, additions were faster on most computers than multiplications. Many attempts were therefore made to formulate FFT kernels with the smallest number of multiplies. Today multiplications are as fast as adds and most machines can do a multiplication and addition simultaneously. Among machines that can do this are high-performance workstations from IBM, HP, DEC, and SGI, as well as vector-supercomputers from Cray, NEC, and Fujitsu. Some machines achieve this parallelism by a mult–add instruction that is assigned during compilation. In this paper, we will derive FFT kernels that can be formulated uniquely in terms of mult–adds. The numerical implementation shows that they give considerable gain in speed.

2. Conventional FFT kernels.

An FFT kernel [1, 2] calculates the innermost part in a transformation, which has the form

$$Z_{out}(i) = \sum_{j=0}^{P-1} Z_{in}(j)\Omega^j \, \omega^{ij} \tag{2.1}$$

for $i = 0, ..., P-1$. The radix of the kernel is given by the prime factor P which is 2, 3, 4, or 5 in this paper. Both Ω and ω are complex numbers of modulus one. Ω is called the twiddling factor [2] and ω is given by $e^{i\frac{2\pi}{P}}$. Efficient evaluation techniques for (2.1) can be found in [3, 4]. For completeness they will be repeated here. The real part of the array Z_{in} is denoted by zinr, the imaginary part by zini, and correspondingly for Z_{out}. The real part and imaginary parts of the twiddling factor Ω^j are denoted by cr_j and ci_j, respectively.

<u>Radix 2</u>
r1 = zinr(0)
s1 = zini(0)
r = zinr(1)
s = zini(1)
r2 = r $*$ cr_1 − s $*$ ci_1
s2 = r $*$ ci_1 + s $*$ cr_1
zoutr(0) = r2 + r1
zouti(0) = s2 + s1
zoutr(1) = r1 − r2
zouti(1) = s1 − s21

*Received by the editors February 21, 1995; accepted for publication (in revised form) February 22, 1996.
http://www.siam.org/journals/sisc/18-6/28194.html
†Max-Planck Institut für Festkörperforschung, D-70569 Stuttgart, Germany (goedeck@prr.mpi-stuttgart.mpg.de).

Radix 4

r1 = zinr(0)
s1 = zini(0)
r = zinr(1)
s = zini(1)
r2 = r $*$ cr_1 − s $*$ ci_1
s2 = r $*$ ci_1 + s $*$ cr_1
r = zinr(2)
s = zini(2)
r3 = r $*$ cr_2 − s $*$ ci_2
s3 = r $*$ ci_2 + s $*$ cr_2
r = zinr(3)
s = zini(3)
r4 = r $*$ cr_3 − s $*$ ci_3
s4 = r $*$ ci_3 + s $*$ cr_3
r = r1 + r3
s = r2 + r4
zoutr(0) = r + s
zoutr(2) = r − s
r = r1 − r3
s = s2 − s4
zoutr(1) = r − s
zoutr(3) = r + s
r = s1 + s3
s = s2 + s4
zouti(0) = r + s
zouti(2) = r − s
r = s1 − s3
s = r2 − r4
zouti(1) = r + s
zouti(3) = r − s

Radix 3

$bb = .5 * \sqrt{3}$
r1 = zinr(0)
s1 = zini(0)
r = zinr(1)
s = zini(1)
r2 = r $*$ cr_1 − s $*$ ci_1
s2 = r $*$ ci_1 + s $*$ cr_1
r = zinr(2)
s = zini(2)
r3 = r $*$ cr_2 − s $*$ ci_2
s3 = r $*$ ci_2 + s $*$ cr_2
r = r2 + r3
s = s2 + s3
zoutr(0) = r + r1
zouti(0) = s + s1
r1 = r1 − .5 $*$ r
s1 = s1 − .5 $*$ s
r2 = bb $*$ (r2−r3)
s2 = bb $*$ (s2−s3)
zoutr(1) = r1 − s2
zouti(1) = s1 + r2
zoutr(2) = r1 + s2
zouti(2) = s1 − r2

Radix 5

$cos_2 = \cos(2 * \pi /5)$
$cos_4 = \cos(4 * \pi /5)$
$sin_2 = \sin(2 * \pi /5)$
$sin_4 = \sin(4 * \pi /5)$
r1 = zinr(0)
s1 = zini(0)
r = zinr(1)
s = zini(1)
r2 = r $*$ cr_1 − s $*$ ci_1
s2 = r $*$ ci_1 + s $*$ cr_1
r = zinr(2)
s = zini(2)
r3 = r $*$ cr_2 − s $*$ ci_2
s3 = r $*$ ci_2 + s $*$ cr_2
r = zinr(3)
s = zini(3)
r4 = r $*$ cr_3 − s $*$ ci_3
s4 = r $*$ ci_3 + s $*$ cr_3
r = zinr(4)
s = zini(4)
r5 = r $*$ cr_4 − s $*$ ci_4
s5 = r $*$ ci_4 + s $*$ cr_4
r25 = r2 + r5
r34 = r3 + r4
s25 = s2 − s5
s34 = s3 − s4
zoutr(0) = r1 + r25 + r34
r = cos_2 $*$ r25 + cos_4 $*$ r34 + r1
s = sin_2 $*$ s25 + sin_4 $*$ s34
zoutr(1) = r − s
zoutr(4) = r + s
r = cos_4 $*$ r25 + cos_2 $*$ r34 + r1
s = sin_4 $*$ s25 − sin_2 $*$ s34
zoutr(2) = r − s
zoutr(3) = r + s
r25 = r2 − r5
r34 = r3 − r4
s25 = s2 + s5
s34 = s3 + s4

zouti(0) = s1 + s25 + s34
r = cos_2 ∗ s25 + cos_4 ∗ s34 + s1
s = sin_2 ∗ r25 + sin_4 ∗ r34
zouti(1) = r + s
zouti(4) = r − s
r = cos_4 ∗ s25 + cos_2 ∗ s34 + s1
s = sin_4 ∗ r25 − sin_2 ∗ r34
zouti(2) = r + s
zouti(3) = r − s

3. The new FFT kernels. The first step in all of the above conventional kernels is the multiplication of the input elements Z_{in} with the twiddling factor. In this part there is an imbalance of multiplies and adds. For both the real part and imaginary part one needs two multiplies but only one add. The add capacities are thus sitting idle, resulting in a reduction of performance. In the following, we will show that by redefining the twiddling factors one can formulate these kernels in such a way that one has a perfect balance of multiplies and adds. In the part coming from the multiplication with the twiddling factor, we get rid of one multiply, giving higher performance. In the remaining part, we have to introduce some additional multiplies but only at places where up to now we had an unbalanced add. Even though we have more operations in this part, the CPU time does not increase since they can be done simultaneously on the machines we are considering.

Radix 2
$ci_1 = ci_1/cr_1$
r1 = zinr(0)
s1 = zini(0)
r = zinr(1)
s = zini(1)
r2 = r − s ∗ ci_1
s2 = r ∗ ci_1 + s
zoutr(0) = r2 ∗ cr_1 + r1
zouti(0) = s2 ∗ cr_1 + s1
zoutr(1) = − r2 ∗ cr_1 + r1
zouti(1) = − s2 ∗ cr_1 + s1

Radix 4
$ci_1 = ci_1/cr_1$
$ci_2 = ci_2/cr_2$
$ci_3 = ci_3/cr_3$
$cr_{31} = cr_3/cr_1$
r1 = zinr(0)
s1 = zini(0)
r = zinr(1)
s = zini(1)
r2 = r − s ∗ ci_1
s2 = r ∗ ci_1 + s
r = zinr(2)
s = zini(2)
r3 = r − s ∗ ci_2
s3 = r ∗ ci_2 + s
r = zinr(3)
s = zini(3)
r4 = r − s ∗ ci_3
s4 = r ∗ ci_3 + s
r = r1 + r3 ∗ cr_2
s = r2 + r4 ∗ cr_{31}
zoutr(0) = r + s ∗ cr_1
zoutr(2) = r − s ∗ cr_1
r = r1 − r3 ∗ cr_2
s = s2 − s4 ∗ cr_{31}
zoutr(1) = r − s ∗ cr_1
zoutr(3) = r + s ∗ cr_1
r = s1 + s3 ∗ cr_2
s = s2 + s4 ∗ cr_{31}
zouti(0) = r + s ∗ cr_1
zouti(2) = r − s ∗ cr_1
r = s1 − s3 ∗ cr_2
s = r2 − r4 ∗ cr_{31}
zouti(1) = r + s ∗ cr_1
zouti(3) = r − s ∗ cr_1

Radix 3
$crh_1 = .5 * cr_1$
$bb = .5 * \sqrt{3}$
$crbb_1 = cr_1 * bb$
$ci_1 = ci_1/cr_1$
$ci_2 = ci_2/cr_2$
$cr_{21} = cr_2/cr_1$
r1 = zinr(0)
s1 = zini(0)
r = zinr(1)

s = zini(1)
r2 = r − s ∗ ci_1
s2 = r ∗ ci_1 + s
r = zinr(2)
s = zini(2)
r3 = r − s ∗ ci_2
s3 = r ∗ ci_2 + s
r = r2 + r3 ∗ cr_{21}
s = s2 + s3 ∗ cr_{21}
zoutr(0) = r ∗ cr_1 + r1
zouti(0) = s ∗ cr_1 + s1
r1 = r1 − r ∗ crh_1
s1 = s1 − s ∗ crh_1
r2 = r2−r3 ∗ cr_{21}
s2 = s2−s3 ∗ cr_{21}
zoutr(1) = r1 − s2 ∗ $crbb_1$
zouti(1) = s1 + r2 ∗ $crbb_1$
zoutr(2) = r1 + s2 ∗ $crbb_1$
zouti(2) = s1 − r2 ∗ $crbb_1$

Radix 5
cos_2 = cos(2 ∗ π /5)
cos_4 = cos(4 ∗ π /5)
sin_2 = sin(2 ∗ π /5)
sin_4 = sin(4 ∗ π /5)
$ci_1 = ci_1/cr_1$
$ci_2 = ci_2/cr_2$
$ci_3 = ci_3/cr_3$
$cr_{23} = cr_2/cr_3$
$cos_{24} = cos_2 * cr_3$
$cos_{44} = cos_4 * cr_3$
$ci_4 = ci_4/cr_4$
$cr_{14} = cr_1/cr_4$
$cos_{25} = cos_2 * cr_4$
$sin_{25} = sin_2 * cr_4$
$cos_{45} = cos_4 * cr_4$
$sin_{45} = sin_4 * cr_4$
$sin_{24} = (sin_2/sin_4) * (cr_3/cr_4)$
$sin_{44} = (sin_4/sin_2) * (cr_3/cr_4)$
r1 = zinr(1)
s1 = zini(1)
r = zinr(2)
s = zini(2)
r2 = r − s ∗ ci_1
s2 = r ∗ ci_1 + s
r = zinr(3)
s = zini(3)
r3 = r − s ∗ ci_2
s3 = r ∗ ci_2 + s
r = zinr(4)
s = zini(4)
r4 = r − s ∗ ci_3
s4 = r ∗ ci_3 + s
r = zinr(5)
s = zini(5)
r5 = r − s ∗ ci_4
s5 = r ∗ ci_4 + s
r25 = r2 ∗ cr_{14} + r5
r34 = r3 ∗ cr_{23} + r4
s25 = s2 ∗ cr_{14} − s5
s34 = s3 ∗ cr_{23} − s4
zoutr(1) = r1 + r25 ∗ cr_4 + r34 ∗ cr_3
r = r1 + cos_{25} ∗ r25 + cos_{44} ∗ r34
s = s25 + sin_{44} ∗ s34
zoutr(2) = r − sin_{25} ∗ s
zoutr(5) = r + sin_{25} ∗ s
r = r1 + cos_{45} ∗ r25 + cos_{24} ∗ r34
s = s25 − sin_{24} ∗ s34
zoutr(3) = r − sin_{45} ∗ s
zoutr(4) = r + sin_{45} ∗ s
r25 = r2 ∗ cr_{14} − r5
r34 = r3 ∗ cr_{23} − r4
s25 = s2 ∗ cr_{14} + s5
s34 = s3 ∗ cr_{23} + s4
zouti(1) = s1 + s25 ∗ cr_4 + s34 ∗ cr_3
r = s1 + cos_{25} ∗ s25 + cos_{44} ∗ s34
s = r25 + sin_{44} ∗ r34
zouti(2) = r + sin_{25} ∗ s
zouti(5) = r − sin_{25} ∗ s
r = s1 + cos_{45} ∗ s25 + cos_{24} ∗ s34
s = r25 − sin_{24} ∗ r34
zouti(3) = r + sin_{45} ∗ s
zouti(4) = r − sin_{45} ∗ s

As we see, instead of the sine and cosine of the twiddling angle we need its sine and tangent (= sine/cosine). For some radices, in addition we need some other quantities that are easily calculated such as the ratio of two twiddling cosines. In case one does multiple or many dimensional FFTs, this additional overhead is completely negligible. The total number of coefficients related to the trigonometric quantities is, however, the same in the two sets of kernels, with the exception of the radix 5 kernel where two additional ones are needed. In all cases, however, the kernels can be implemented

TABLE 1
Number of CPU cycles needed for one pass through a radix P kernel on a machine with multiply–add.

	Conventional kernel	New kernel
Radix 2	8	6
Radix 4	28	22
Radix 3	20	16
Radix 5	48	40

TABLE 2
Comparison of required operations for conventional and new kernels on a machine without mult–add instruction.

	conventional kernel		new kernel	
	mults	adds	mults	adds
Radix 2	4	6	4	6
Radix 4	12	22	14	22
Radix 3	12	16	14	16
Radix 5	32	40	36	40

with 32 floating point registers available on practically all RISC processors. Neglecting cycles that are needed to load or store data, the number of cycles is reduced as shown in Table 1.

Whereas in the framework of conventional kernels a radix 8 kernel does give some additional savings compared with radix 4, this is not true for these new types of kernels. Within this framework it would also not make sense to use number-theoretic prime factor FFTs, since the main motivation of these techniques is to get rid of the twiddling factor. Note also that by precalculating products in the new kernels, one can nearly come back to the operation counts of conventional kernels as shown in Table 2. In the unlikely case that one should use a machine that does not have a multiply–add facility, one does not loose very much.

4. Accuracy of the new kernels. The new kernels consist of repeated transformations of the type

$$ax + by \to a(x + (b/a)y).$$

Instead of adding/subtracting ax and by, one adds/subtracts the scaled quantities x and $(b/a)y$. Since both operands are scaled in the same way the error accumulation properties of the new kernels and old kernels are the same. The slow increase of the error propagation has been verified for all the kernels, and in Figs. 1 and 2 the error propagation properties are shown for the case of radix 4 kernels. The fact that the error propagation of the new set of kernels is slightly better comes from the fact that no intermediate rounding is performed in a mult–add instruction. The accuracy of a mult–add instruction therefore exceeds the IEEE standard.

5. Timing results. The two sets of kernels were compared on an IBM RS6000/590 and an SGI/MIPSR8000 workstation; both have a single mult–add instruction. Both also have two floating point units such that two mult–add instructions can be done within one cycle. In Table 3, the timing results of multiple FFTs are shown for data sets that fit into cache. For the tests of radix 4, 3, and 5 transforms of length 64, 81, and 125 were used, respectively. If the cache is large enough, one can do a large number of multiple FFTs and loop overheads become less important.

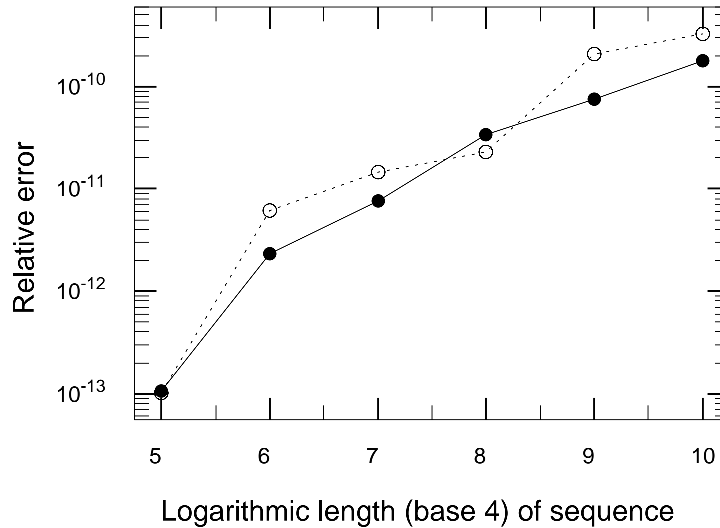

FIG. 1. *The relative error of a sequence of forward-backward transforms of data sets containing up to one million items. The solid line is obtained with the new kernel, the dashed line with the conventional kernel of the ESSL library. As can be seen the error growth is very similar for both. The tests were done with eight bytes floating point numbers.*

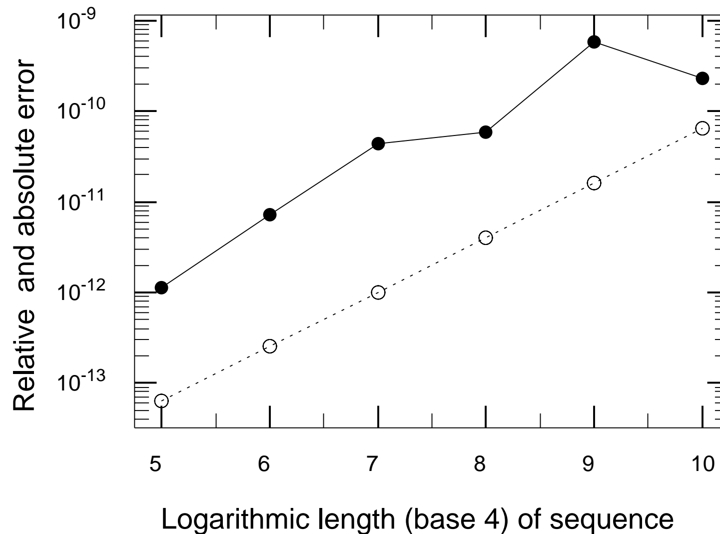

FIG. 2. *The errors in the Fourier spectrum of a sequence that is a superposition of three frequencies and where the exact result is therefore known. The solid line gives the relative error for the nonzero weight frequencies, the dashed line the absolute error for the zero weight frequencies. The results obtained with the new kernel and the ESSL library are indistinguishable on the scale of the figure.*

Since the cache is much larger on the SGI machine, 1024 multiple FFTs could be done, whereas on the IBM machine only 64 could be fit into cache. Even for many simultaneous FFTs it is not possible to reach the theoretical limit. On the SGI machine, for instance, it is not possible to overlap the branching decision adding at least

TABLE 3

Measured cycles for radix 3, 4, and 5 with different combinations of kernels and machines. In the header, the number in brackets gives the number of simultaneous FFTs; in the remaining lines the numbers in brackets give the theoretical limit assuming that the code would run at the peak speed attainable by the machine.

	IBM ESSL (64)	IBM new kernel (64)	SGI new k. (64)	SGI new k. (1024)
Radix 4	14.9 (14)	12.3 (11)	14.1 (11)	13.2 (11)
Radix 3		11.3 (8)	11.1 (8)	10.2 (8)
Radix 5		27.0 (20)	62.0 (20)	62.0 (20)

one cycle per loop iteration. It is also not always possible to overlap all the address calculations. The comparison with the ESSL library (version 2.2.1) could be done only for powers of 4, since it does not allow arbitrary powers of 3 and 5. The radix 2 kernel is load/store bound on the SGI machine and was therefore not included in the test. For radix 5 the SGI compiler was not able to generate an efficient machine code, resulting in very poor performance.

6. Conclusions. To obtain very high performance on modern computer architectures, it is very helpful to have an alternating stream of multiply and add instructions. We have presented FFT kernels that give rise to such an instruction pattern. Timing results show that these kernels clearly outperform other kernels. A very good compiler is a necessity to obtain an optimal instruction pattern with a FORTRAN program.

Acknowledgments. We thank the Cornell Theory Center in Ithaca, NY, and Dr. Rappe at the University of Pennsylvania for giving us access to their machines.

REFERENCES

[1] C. VAN LOAN, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, PA, 1992.

[2] E. O. BRIGHAM, *The Fast Fourier Transform and its Applications*, Prentice–Hall, Englewood Cliffs, NJ, 1988.

[3] C. TEMPERTON, *Self-sorting mixed-radix fast Fourier transforms*, J. Comput. Phys., 52 (1983), pp. 1–23.

[4] R. C. SINGLETON, *An algorithm for computing the mixed radix fast Fourier transform*, IEEE Trans. Audio Electroacoust., 17 (1969).