FFTSS: A HIGH PERFORMANCE FAST FOURIER TRANSFORM LIBRARY

Akira NUKADA

CREST, JST
Department of Computer Science, University of Tokyo

ABSTRACT

In this paper, we introduce a new Fast Fourier Transform (FFT) library. In developing this software, we focus on the efficient execution of the floating-point operation instructions. To achieve high performance on various processors, we provide the source code which compilers can optimize easily. Since the compilers provided by processor vendors have powerful optimizers for loop sentenses, the code generated by them will run very fast as long as the iteration count of the inner most loop is large enough. In such a case, the library outperforms other libraries even provided by processor vendors.

1. INTRODUCTION

Today, there are many FFT libraries in the world. Each of them has its own characteristics. The processor vendors provide FFT libraries such as IBM ESSL library for PowerPC processors and Intel Math Kernel Library (MKL) for Intel Processors. On the other hand, various open-source libraries are provided by many developers. Especially FFTW [1] library is the most popular as cross-platform FFT library.

We advanced the research in the Scalable Software Infrastructure(SSI) [2] Project, and released the FFTSS library as a part of the results of the project. The high performances on various processors are requested for an open-source library. For this reason, the FFTSS library package contains many kinds of FFT kernel sets, and the best kernel set is selected at runtime. The kernels are written as the codes which the compilers can easily optimize. In addition, the library includes the kernels which use the special features dependent on the types of the processors, and includes the kernels which are opmtized for specific processor. The radix-8 kernels [3] in the library are based on a new kernel we have developed.

2. CODING POLICY

Open-source libraries are usually described in the high level programming languages such as C, Fortran. Users compile them, and then execute their application programs. The optimization by the compilers greatly influence the execution performance of the application.

The approach of the FFTSS library is to prepare the code which the compilers can optimize easily. The library is written mainly in C language. To give more information to the compiler, const keyword and C99 keywords such as inline, restrict, complex are used. Especially, the const and restrict keywords are important for using the pointers to the arrays in C programs. Using these keyword allows the compilers' aggressive optimizations.

In addition, we can add hints based on the fact that this is an FFT program. A typical FFT program is described by the following, double loop sentence.

```
for (i = 0; i < N1; i++) {
  { load twiddle factors here. }
  for (j = 0; j < N2; j++)
    { compute kernel. }
}
```

The iteration count of the inner loop may becomes one or small number. In such a case, the efficiency of the instruction execution is heavily degraded. We certainly know the fact that the iteration count of the inner loop becomes smaller and smaller, and finally it becomes one.

Traditionally, the technique of loop exchange is used against this problem. If the iteration count of the inner loop is smaller than that of the outer loop, then exchange the inner and out loop.

But in case of the typical FFT program, the exchange increase the load operations of twiddle factors. Instead of the loop exchange, we decide to use loop unrolling of the inner loop. The inner loop is unrolled only when the iteration count is small enough.

In case of N2=1, the double loop is converted as follows.

```
for (i = 0; i < N1; i++) {
  { load twiddle factors here. }
  { compute kernel. }
}
```

In case of N2=4, the double loop is converted as follows in the same manner.

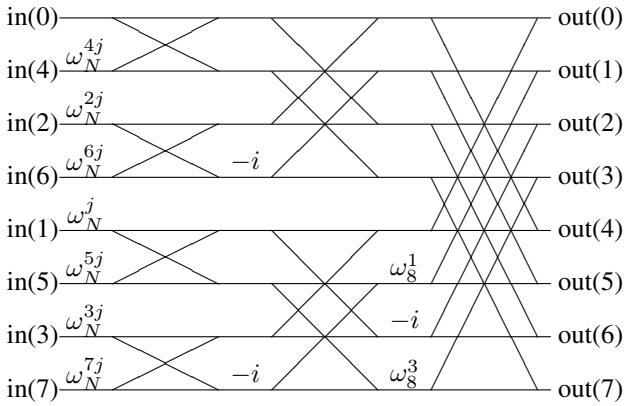

Fig. 1. Conventional radix-8 FFT kernel

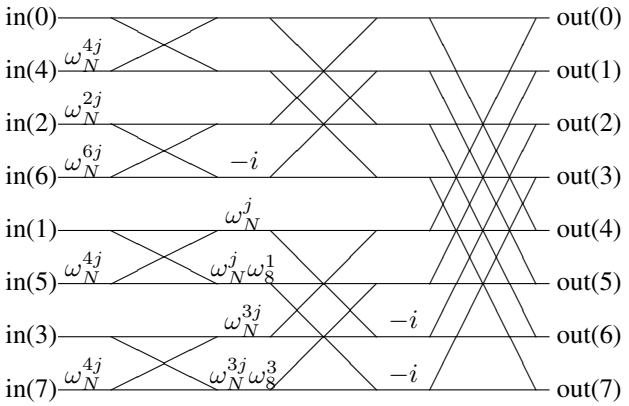

Fig. 2. New radix-8 FFT kernel

```
for (i = 0; i < N1; i++) {
  { load twiddle factors here. }
  { compute kernel. }
  { compute kernel. }
  { compute kernel. }
  { compute kernel. }
}
```

Since these conversions increase the program size, the unrolling for large N2 may cause instruction cache misses. For this reason, this unrolling is limited to the cases of $N2 \leq 4$.

3. SPECIAL INSTRUCTIONS

Processors support special instructions that greatly contribute to the performance improvement. In case of some special instructions, we need to write the code which explicitly use those instructions. The FFTSS library currently supports the following instructions.

3.1. Fused Multiply-Add (FMA) instructions

Many processors such as PowerPC, MIPS and IA-64 support the FMA instructions. An FMA execution unit multiplies two numbers and then adds a number to the result of the multiplication. The FMA unit is occupied even when only addition, subtraction or multiplication is to be computed. For these processors, FFT kernels with a smaller number of FMA instructions are developed [4, 5, 6]. The FFTSS library includes some FFT kernels in which multiplications of complex numbers are converted as follows using Linzer's idea [4].

$$ax \pm by \rightarrow a\{x \pm \frac{b}{a}y\}$$

The multiplication of a is coupled with additions or subtractions of successive butterfly operations.

3.2. Single Instruction Multiple Data (SIMD) instructions

The latest processors support some kinds of SIMD instructions. The FFTSS library supports Intel SSE2/SSE3 instructions. Especially for Intel processors, the use of these instructions is in fact indispensable.

3.3. PoewrPC440 Double Hammer FPU

PowerPC440 Double Hammer FPU [7] is installed in the super computer BlueGene. The processor core has two FPU units and both of them are controlled by a series of instructions like SIMD instructions.

4. RADIX-8 KERNEL

Fig.1 shows conventional radix-8 FFT kernel. In the conventional radix-8 kernel, input data is multiplied by twiddle factors, and then 8-point FFT is computed.

The radix-8 kernels used in the FFTSS library are based on the different radix-8 kernel described in **Fig.2**.

The new radix-8 kernel does not decrease the number of floating-point operations compared with the conventional one. The difference between them is the timings of the multiplications of the twiddle factors. In case of the conventional one, all multiplications concentrate on the first butterfly stage(left in the figure). This causes the congestion of the load instruction. On the other hand, multiplications are distributed to the first and second butterfly stages in case of the new kernel.

Linzer's radix-8 kernel for FMA is based on the conventional radix-8 kernel. The new radix-8 kernel also can

be converted into FMA-style in the same manner. For processors with the FMA instructions, we prepare the kernels based on the new radix-8 kernel.

5. TWIDDLE FACTOR TABLES

In general, we prepare trigonometric tables called 'twiddle factor tables' required in computing FFT. The sizes of the tables are listed in **Table 1**.

The conventional radix-8 kernel requires the twiddle factor tables only for radix-8. On the other hand, the new radix-8 kernel uses the same twiddle factor tables as those of the radix-4 kernels. This is because the new radix-8 kernel consists of a radix-4 kernel, two radix-2 kernels and two split-radix [8] kernels. All of them are subsets of the radix-4 kernel. This makes it easy to use both radix-4 and radix-8 kernels. In the implementation of the FFTSS library, powers of two transforms are computed with radix-4 and radix-8 kernels. In such a case, it becomes the advantage that they can use the shared twiddle factor tables.

The table for the FMA kernels used in the FFTSS library is described below.

$$\begin{aligned} wn(2k) &= \cos(2\pi k/N)/\sin(2\pi k/N) \\ &\qquad 0 \le k < N/2 \\ wn(2k+1) &= \sin(2\pi k/N) \\ &\qquad 0 \le k < N/2 \\ wn(2k+3N/2) &= \cos(6\pi k/N)/\sin(6\pi k/N) \\ &\qquad 0 \le k < N/4 \\ wn(2k+3N/2+1) &= \sin(6\pi k/N)/\sin(2\pi k/N) \\ &\qquad 0 \le k < N/4 \end{aligned}$$

In the radix-4 and radix-8 kernels for FMA, $\cos(6\pi k/N)/\sin(6\pi k/N)$ and $\sin(6\pi k/N)/\sin(2\pi k/N)$ are required at the same time. Therefore, this table is designed to load the pair from a continuous address. That is important not only for the processors with packed load instructions, but also for the efficient use of the cache memory.

6. LIBRARY INTERFACES

The interfaces of the FFTSS library are designed to be almost compatible with those of FFTW library. We provide a header file "fftw3compat.h" for FFTW users. If you have application programs written for fftw3, the porting to the FFTSS library is very simple. All you have to do is to change "fftw3.h" into "fftw3compat.h". Unfortunately, the current version does not support all of the FFTW features.

The limitations of the current version are:

- The size of transform must be powers of two.
- Only one-dimensional transform is supported.
- Only double precision complex-to-complex transform is supported.

As long as only the supported function is used, your application will work well.

The package of the FFTSS library contains various FFT kernel sets such as normal, FMA, SSE2, SSE3, etc. In the subroutine "fftss_plan_dft_1d()", which creats a plan of one-dimensional transform, the best kernel sets are selected by measuring the performance of them actually.

To compute the powers of two transforms, the FFTSS library only uses radix-4 and radix-8 kernels. In case of latest processors, one of them will be the best radix size, considering the bandwidth of the cache memory. We concern the following four about the combination of them.

- priority on radix-4, radix-4 before radix-8.
- priority on radix-4, radix-8 before radix-4.
- priority on radix-8, radix-4 before radix-8.
- priority on radix-8, radix-8 before radix-4.

For example, suppose that the radix-4 is better if the data is in L1-cache, and the radix-8 is better if the data is in L2-cache. In this case, the better radix is automatically selected by measuring the performance of those combinations.

The other combinations are ignored to save the cost of creating plan.

7. PERFORMANCE EVALUATIONS

Fig.3 shows the performances of the FFTSS library, IBM ESSL for Linux version 4.2, and FFTW library version 3.0.1. The machine used in the evaluations is IBM OpenPower 710, running SuSE Linux 9 (Linux kernel 2.6.5). Two Power5 1.65GHz (DualCore) processors and 1GB memory are installed in the system. But only one processor (one core) is used in the executions

For each library, the best compiler and compiler options are selected. In case of FFTSS, IBM XL C compiler 7.0.1 is used and CFLAGS is set to '-O3 -qarch=auto -qtune=auto -qansialias -qlanglvl=extc99'. In case of FFTW, gcc 4.0.1 is used and CFLAGS is set to '-O3 -fomit-frame-pointer -fno-schedule-insns -fstrict-aliasing -mcpu=powerpc', which is defined by the configure script.

Using these libraries, the times required to repeat the backward and forward transforms $300/log_2 N$ times are measured. We assume the number of floating-point operations is $5N \log_2 N$. The performances in Gflops are calculated from them.

Since the Power5 processor has two FMA units, 1.65GHz processor can execute 6.6G floating-point operations per second.

The size of transform in **Fig. 3** is a range from 256 to 8192. In this range, the performance of the FFTSS library outperforms the other libraries by maximum 18%.

Table 1. The sizes of twiddle factor tables for computing an N-point FFT($N = 2^m$)

	Normal		Optimized for FMA		
	Radix-4	Radix-8	Radix-4	Linzer's radix-8	New radix-8
$\sin(x)$	3N/4	7N/8	N/2	3N/8	N/2
$\cos(x)$	3N/4	7N/8	0	0	0
$\cos(x)/\sin(x)$	0	0	3N/4	7N/8	3N/4
$\sin(3x)/\sin(x)$	0	0	N/4	N/4	N/4
$\sin(5x)/\sin(x)$	0	0	0	N/8	0
$\sin(7x)/\sin(x)$	0	0	0	N/8	0
$\sin(x)/\sqrt{2}$	0	0	0	N/8	0
Total	3N/2	7N/4	3N/2	15N/8	3N/2

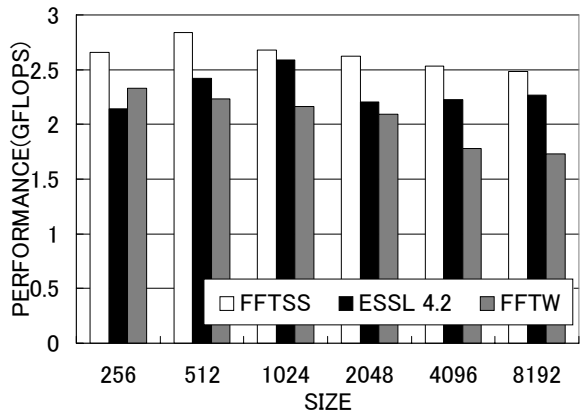

Fig. 3. The performance comparison between FFTSS and ESSL and FFTW.

8. CONCLUDING REMARKS

We have introduced the design and implementation of the FFTSS library. The development of this library is focused on how to prepare the code which compilers can easily optimize.

We can assist the optimization of the compilers by giving a lot of information we have, in various ways. As the result, the library achieved higher performance than the libraries provided by processor vendors.

The FFTSS library is available at the following URL.
`http://ssi.is.s.u-tokyo.ac.jp/fftss/`

ACKNOWLEDGEMENT

This work has been supported by CREST of JST(Japan Science and Technology Agency).

9. REFERENCES

[1] Matteo Frigo and Steven G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[2] *SSI Project*, http://ssi.is.s.u-tokyo.ac.jp/.

[3] G. D. Bergland, "A Fast Fourier Transform Algorithm Using Base 8 Iterations," *Math. Comp.*, vol. 22, pp. 275–279, 1968.

[4] E. N. Linzer and E. Feig, "Implementation of Efficient FFT Algorithms on Fused Multiply-Add Architectures," *IEEE Trans. Signal Processing*, vol. 41, pp. 93–107, 1993.

[5] S. Goedecker, "Fast Radix 2,3,4 and 5 Kernels for Fast Fourier Transformations on C omputers with Overlapping Multiply-Add Instructions," *SIAM J. Sci. Comput.*, vol. 18, pp. 1605–1611, 1997.

[6] H. Karner and et al., "Multiply-Add Optimized FFT Kernels," *Math. Models and Methods in Appl. Sci.*, vol. 11, pp. 105–117, 2001.

[7] C. D. Wait, "IBM PowerPC 440 FPU with complex-arithmetic extensions," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 249–254, 2005.

[8] P. Duhamel and H. Hollmann, "Split-Radix FFT Algorithm," *Electron. Lett.*, vol. 20, pp. 14–16, 1984.