

High Performance FFT on SGI Altix 3700

Akira Nukada^{1,2}, Daisuke Takahashi^{3,1}, Reiji Suda^{2,1}, and Akira Nishida^{2,1}

¹ Japan Science and Technology Agency, Saitama 332-0012, Japan

² The University of Tokyo, Tokyo 113-8685, Japan

{nukada,reiji,nishida}@is.s.u-tokyo.ac.jp

³ University of Tsukuba, Ibaraki 305-8573, Japan

daisuke@cs.tsukuba.ac.jp

Abstract. We have developed a high-performance FFT on SGI Altix 3700, improving the efficiency of the floating-point operations required to compute FFT by using a kind of loop fusion technique. As a result, we achieved a performance of 4.94 Gflops at 1-D FFT of length 4096 with an Itanium 2 1.3 GHz (95% of peak), and a performance of 28 Gflops at 2-D FFT of 4096² with 32 processors. Our FFT kernel outperformed the other existing libraries.

1 Introduction

The Fast Fourier Transform (FFT) [1] algorithm is now used in many fields, and its fast adaptation to computer systems continues to be expected. We developed a high-performance FFT on SGI Altix 3700 [2], which is a hardware distributed shared memory parallel computer with Intel Itanium 2 processors.

The performance of floating-point operations in modern microprocessors has significantly matured. The next issue is how to use the FPUs efficiently. In this paper, we focus on the efficiency of executing instructions. Several optimization techniques for nested loops have been researched [3]. But they are not adequate for achieving perfect performance of processors such as the Itanium 2, because the overheads of the loop initializations are too large. Various kinds of loop fusion methods for FFTs on vector processors have been proposed [4,5]. Those methods require index arrays, which are usually larger than the capacity of the cache memory. For this reason, conventional loop fusion methods are not suitable for non-vector processors like Itanium 2. Therefore, we propose a new loop fusion method without index arrays, and implemented highly optimized FFT kernels for the Itanium 2.

2 IA-64

Intel Architecture 64 (IA-64) [6] is a new 64-bit processor architecture developed by Intel and Hewlett Packard. As a major step forward from the IA-32, Intel's 32-bit architecture, the IA-64 is equipped with many advanced and challenging technologies.

Table 1. The capacities of the cache memories

	Itanium	Itanium 2	
	Merced	McKinley	Madison
Level-1 Inst.	16kB	16kB	16kB
Level-1 Data	16kB	16kB	16kB
Level-2 Unified	96kB	256kB	256kB
Level-3 Unified	2MB	1.5/3MB	3/6MB

Itanium is the first product of the IA-64 architecture, codenamed 'Merced', and was released in 2001. The second product, codenamed 'McKinley', was released as Itanium 2 in 2002, and the third product, codenamed 'Madison' was released in 2003.

2.1 EPIC

The IA-64 uses a kind of VLIW (Very Long Instruction Word) [7] architecture. A 16-byte 'bundle' usually contains three instructions, and the Itanium and the Itanium 2 processor can execute two bundles, i.e., up to six instructions, in each cycle. This number is large compared with other processors. EPIC (Explicitly Parallel Instruction Computing) [8] technology plays an important role in exploiting its performance.

Checking the dependencies between instructions in order to execute many instructions in one cycle requires a great deal of hardware resources. For this reason, the number of instructions to be executed in a single cycle is limited. In the case of EPIC, the dependencies between the instructions are checked at the compilation time, and independent instructions are explicitly grouped to avoid dependency. Thus, the grouped instructions can be subsequently executed without checking dependencies.

Itanium 2 processors have four memory units, two integer units, two floating-point units, and three branch units. Up to six instructions are sent to those execution units.

2.2 Memory Hierarchy

The IA-64 processors have a level-1 instruction cache, a level-1 data cache, a level-2 unified cache, and a level-3 unified cache. Their capacities depend on the models as listed in Table 1.

In the IA-64 architecture, the floating-point data do not pass the level-1 cache, but are directly transferred from the level-2 cache. The throughput of the level-2 cache is large, but the latency is long.

2.3 Predicate Register

In case of the IA-64 instruction set architecture, predicate registers can be specified to almost all instructions. The predicate register is a single-bit register, and 64 registers in total are available. If a predicate register is specified to an instruction, the instruction will be executed only if the value of the predicate register

is 1. The values of the predicate registers are changed by compare instructions, and logical operations between the predicate registers are also available.

Implementations with branch instructions always have the risk of penalties for the branch prediction misses. But implementations with predicate registers have no such risk; therefore, even more complex operations including conditional statements can be executed efficiently. Although an instruction is not executed if the value of the specified predicate register is 0, the instruction nonetheless occupies an instruction slot. In practice, it is a rare case that this matters, because the IA-64 processors can execute six instructions in each cycle.

2.4 Register Rotation

The IA-64 architecture uses the mechanism of register rotation. The relations between the logical register numbers and the physical register numbers can be rotated mainly by the branch instructions for loops described later. The IA-64 architecture has 128 floating-point registers, and 96 registers from the number 32 can be rotated. In general, a long sequence of instructions is required to utilize such a large number of registers. But the mechanism of the register rotation enables a short compact sequence of the instructions to use many physical registers. There are 64 predicate registers, where 48 registers from the number 16 can be rotated. The general purpose (integer) registers also can be rotated partially.

2.5 Branch Instructions for Loop

The IA-64 instruction set defines several kinds of branch instructions, including a normal branch instruction, a branch instruction for calling functions, a branch instruction for returning from functions, and branch instructions for loops. The conditional branches are implemented by the normal branch instruction with the predicate register.

The branch instructions for loops reference the Loop Counter register (LC) and the Epilogue Counter register (EC) to control the iteration. The LC is set to (iteration count - 1) of the original loop. The EC is set to the number of extra iterations required by the software pipelining. The '**br.cloop**' instruction only references the LC. On the other hand, the branch instructions for the modulo-scheduled loops [9] such as '**br.ctop**' reference both the LC and EC, and the registers are rotated.

Those branch instructions reference the value of the LC, and jump to a specified address as long as the value is positive, and the LC is decreased by one. When the LC becomes zero, the branch instructions for the modulo-scheduled loops also reference the value of the EC and decrease it. In case of the modulo-scheduled loop, the predicate registers are rotated by the branch instructions. After rotating, the predicate register number 16 (p16) is set to 1 as long as the value of the LC is positive. When the LC becomes 0, p16 is set to 0. Using the features of the predicate registers, the register rotation, and the branch instruction for loops, we can describe an efficient modulo-scheduled loop very simply.

3 Conventional Optimizations of Nested Loop of FFT

Many of the traditional implementations of the FFT algorithm have nested loops. An example of the radix-2 FFT kernel [5] is described below:

```
double complex vin[n1][2][n2], vout[2][n1][n2];
double complex w, table[n1][n2];
for (i = 0; i < n1; i++) {
    w = table[i][0];
    for (j = 0; j < n2; j++) {
        t = w * vin[i][1][j];
        s = vin[i][0][j];
        vout[0][i][j] = s + t;
        vout[1][i][j] = s - t;
    }
}
```

The array ‘**table**’ is a trigonometric table called the twiddle factor table. Stockham’s auto-sort algorithm [4,5] is used in this example. For this reason, the input array ‘**vin**’ can not be overwritten. The results are written to another output array ‘**vout**’.

The iteration counts of the nested loops are n_1 and n_2 , respectively, and they change during the computation of the FFT. For example, using the radix-2 FFT kernel for the input size $N(= n_1 * n_2 * 2) = 64$, (n_1, n_2) becomes $\{(1, 32), (2, 16), (4, 8), (8, 4), (16, 2), (32, 1)\}$. When the iteration count of the inner loop n_2 is too small, the efficiency of executing instructions is generally degraded because of the overhead around the loop. There are several loop optimization methods [3] proposed for such a case.

3.1 Loop Interchange

If one of the iteration counts of the nested loops is large enough, the performance may be improved by interchanging the inner loop and the outer loop [4,5].

```
if (n1 > n2) {
    for (j = 0; j < n2; j++) {
        for (i = 0; i < n1; i++) {
            w = table[i][0];
            t = w * vin[i][1][j];
            s = vin[i][0][j];
            vout[0][i][j] = s + t;
            vout[1][i][j] = s - t;
        }
    }
} else {
    ....
}
```

In this example, the loop interchange is performed only if the iteration count of the outer loop is larger than that of the inner loop. But the loop interchange does not always improve the performance in the case of the FFT. The loop interchange increases the number of load operations for the twiddle factor table, and the memory accesses to the input and output buffers are no longer contiguous.

3.2 Unrolling of Inner Loop

The efficiency of executing instructions decreases when the iteration count of the inner loop is small. The rate of the performance degradation depends on the overhead of the loops such as the penalty for the branch prediction misses and so on. Assume that branch prediction misses occur once every n_2 iterations. A drastic performance degradation results only if n_2 is too small. Thus, we should unroll the inner loop only in those cases.

```
switch (n2) {
case 1:
  for (i = 0; i < n1; i++) {
    w = table[i][0];
    t = w * vin[i][1][0];
    s = vin[i][0][0];
    vout[0][i][0] = s + t;
    vout[1][i][0] = s - t;
  }
  break;
case 2:
  for (i = 0; i < n1; i++) {
    w = table[i][0];
    t = w * vin[i][1][0];
    s = vin[i][0][0];
    vout[0][i][0] = s + t;
    vout[1][i][0] = s - t;
    t = w * vin[i][1][1];
    s = vin[i][0][1];
    vout[0][i][1] = s + t;
    vout[1][i][1] = s - t;
  }
  break;
  ....
}
```

The improvement in performance achieved by unrolling is especially large in the case of smaller n_2 values. As n_2 becomes larger, the ratio of improvement decreases. The exact threshold at which n_2 should be unrolled should be determined in consideration of the size of the instruction cache memory and so on. In general, the use of unrolling for large n_2 values is not recommended.

3.3 Loop Fusion Using Index Array

When computing on vector computers, the iteration count of the inner loop is a very important factor. To achieve high performance with vector processing, the iteration count must be sufficiently large. Using the loop interchange, the iteration count may become $\sqrt{N/2}$ in the worst case, and this is not sufficiently large for small N values. Pease proposed a method to combine nested loops into a single loop [10]. The method prepares index arrays to realize non-contiguous access in nested loops and performs gather operations using them. The example of $n2 = 4$ is described below.

```
double complex vin[n1 * n2 * 2], vout[n1 * n2 * 2];
double complex w, table[n1 * n2];
long index1[] = {0, 1, 2, 3, 8, 9, 10, 11, 16, ...};
long index2[] = {0, 0, 0, 0, 1, 1, 1, 1, 2, ...};
for (i = 0; i < n1 * n2; i++) {
    w = table[index2[i] * n2];
    t = w * vin[index1[i] + n2];
    s = vin[index1[i]];
    vout[i] = s + t;
    vout[i + n1 * n2] = s - t;
}
```

Although the iteration count of the loop becomes large, the method requires $O(N \log N)$ memory space for the index arrays, and additional memory access to the index arrays is required.

4 Loop Fusion Without the Index Arrays

In the case of the modulo scheduled loop of Itanium 2, the throughput of executing instructions is very high, whereas the overheads of the loops are large. Therefore, the technique of loop interchange is not sufficient to exploit the potential. The conventional loop fusion method requires large index arrays. This causes many cache misses; therefore, those methods are not suitable for non-vector processors including Itanium 2. Thus, we propose a new loop fusion method without index arrays.

At first, we observed the memory access patterns for ‘**vin**’, ‘**vout**’, and ‘**table**’. The addresses for ‘**vin**’ move 16 bytes forward at each iteration; additionally, they move $(n2 * 16)$ bytes forward once every $n2$ iterations. The addresses for ‘**table**’ move $(n2 * 16)$ bytes forward once every $n2$ iterations. The addresses for ‘**vout**’ move 16 bytes forward at each iteration. Using the operations of pointers, they can be described as follows:

```
double complex *vin0 = vin, *vin1 = vin + n2;
double complex *vout0 = vout, *vout1 = vout + n1 * n2;
double complex *ptable = table;
count = n2;
```

```

for (i = 0; i < n1 * n2; i++) {
    w = *ptable;
    t = w * *vin1; s = *vin0;
    *vout0 = s + t; *vout1 = s - t;
    vin0 += 1; vin1 += 1;
    vout0 += 1; vout1 += 1;
    count -= 1;
    if (count == 0) {
        vin0 += n2; vin1 += n2;
        ptable += n2;
        count = n2;
    }
}

```

The variable ‘**count**’ is used to count down from n_2 , and the adjustments of the pointers are performed when the count becomes zero. Although the number of address operations is increased, the nested loops can be combined into a single loop. The latest super-scalar processors can execute many instructions in each cycle. In particular, IA-64 processors can execute six instructions, and the post increment feature of the load and store instructions is usable to increase the addresses by 16 bytes.

4.1 Implementation on Itanium 2

Combining the nested loops of the FFT into a single loop, the iteration count is always sufficiently large. This may improve the efficiency of executing instructions, while the ‘**if**’ statement appears in the loop. If the statement is implemented with branch instructions, the branch prediction misses degrade the performance. For this reason, this kind of transform is usually not performed. But in the case of implementation with the predicate register of IA-64, the transform is effective.

The memory accesses for the twiddle factor table are performed at each iteration. Since the accesses to the same addresses are repeated in n_2 contiguous iterations, the lines are probably stored in the cache memory. The number of load operations actually increases; therefore, the high-performance (cache) memory access such as that of IA-64 is required.

4.2 Implementation on Other Architecture

The loop fusion method can be implemented, not with predicate registers but with generic integer operations. If n_2 is a power of two, it is clearly possible to implement with logical operations as follows:

```

count = 0;
for (i = 0; i < n1 * n2; i++) {

```

```

....
count += 1;
b = count & n2;
count -= b;
vin0 += b; vin1 += b;
ptable += b;
}

```

This is because we can use $n2$ as the bit mask and create the counter easily if $n2$ is a power of two. Even if not, a similar operation is possible. The following example uses the arithmetic shift right operation.

```

count = n2 - 1;
for (i = 0; i < n1 * n2; i++) {
....
count -= 1;
b = count >> 63; shift right arithmetic.
b &= n2;
count += b;
vin0 += b; vin1 += b;
ptable += b;
}

```

The size of the integers is assumed to be 64 bits. As the result of the 63-bit arithmetic shift right operation, the significant bit will be copied to all the other bits. For negative numbers, all bits become one. Otherwise, all bits become zero. Using this procedure for the mask operation, the variable 'b' becomes $n2$ once every $n2$ iterations.

Almost all the processors have the arithmetic shift operation; therefore, the loop fusion method is usable not only for IA-64 but also for other processors.

5 Implementation

The floating-point units of the IA-64 processors execute Fused Multiply-Add (FMA) operations. Several FFT kernels for FMA have been proposed [11,12,13]. We used the FFT kernels proposed by Goedecker and Linzer, which require a minimum number of FMA operations. To compute the FFT of the length of powers of two, we can use the radix-2, radix-4, and radix-8 [14] FFT kernel and so on. Table 2 shows the numbers of FMAs, and the load and store operations required in those kernels. Using the radix-4 and radix-8 kernels, the number of the FMA operations becomes smaller than with the radix-2 kernel. We used the radix-4 mainly, and also used the radix-8 if required. Although the sizes of transforms are limited to powers of two in this paper, our loop fusion method can be used for kernels of radix 3, 5, or any other radix.

An optimized assembly code of the radix-4 kernel is used in later experiments. Using the radix-4 kernel for the FMA, the nested loops are combined into a single

Table 2. The number of FMA and load/store operations of the FFT kernels for FMA

	FMA	load	store
Radix-2	6	6	4
Radix-4	22	14	8
Radix-8	66	30	16

loop, and then the software-pipelined modulo-scheduled loop is generated. The instructions are well scheduled in consideration of the dependencies and latencies. In case of the Itanium 2 (Madison), the latency of the FMA is 4 cycles. The latencies of the floating-point load operations from the level-2 cache and the level-3 cache are 6 cycles and at least 23 cycles, respectively. In our optimized kernel, the load operations are scheduled for 22–33 cycles before the data is used. Each iteration takes 11 cycles, and 4 iterations are added for the epilogue. Thus, $11(N/4 + 4)$ cycles in total are required.

When the twiddle factor is 1.0, the multiplication by it can be skipped. This optimization is often used to reduce the number of floating-point operations. In case of the sample code in Section 4, the twiddle factors become 1.0 in the first $n2$ iterations. The radix-4 kernel has three complex multiplications by the twiddle factors. If they are skipped, the number of the FMA operations is reduced to 16. We also generated an optimized modulo-scheduled loop for them. Its epilogue count is 5, and $8(n2 + 5)$ cycles in total are required. The other part, that is, multiplications for which the twiddle factor is not 1.0, is computed normally and requires $11(N/4 - n2 + 4)$ cycles. As the result of skipping the multiplication, $(3 * n2 - 40)$ cycles are eliminated. Therefore, it is effective only if $n2$ is larger than 13. Otherwise, the overhead of splitting the loop is larger. In our experiments, N is a power of two. We used this optimization only if $n2$ was not less than 16.

For the 2-D FFT, the optimized 1-D FFT kernel is used. And it is parallelized using OpenMP as follows.

```
double complex v[N] [N+1];
double complex work1[N], work2[N];
double complex table[N];

#pragma omp parallel for private(work1)
for (i = 0; i < N; i++)
    FFT1D(N, v[i], work1, table);

#pragma omp parallel for private(j,work1,work2)
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) work1[j] = v[j][i];
    FFT1D(N, work1, work2, table);
    for (j = 0; j < N; j++) v[j][i] = work1[j];
}
```

In the transform along the second axis, the data in non-contiguous addresses are copied into the work space, and they are written back after the transform for the efficient memory access. The 2-D array of $N \times (N + 1)$ is allocated to store $N \times N$ data. This is important to improve the performance of the memory access because N is a power of two in our experiments.

In case of the Altix 3700, the physical memory pages are allocated by the first-touch policy. The 2-D array is initialized in parallel to allocate the physical memory pages of all nodes equally.

6 Performance Evaluations

Using the optimized FFT kernels, the performance on the SGI Altix 3700 was evaluated. Table 3 shows the specifications of the system.

6.1 Performance of 1-D FFT

The performance of the 1-D FFT is shown in Table 4. Only one processor (single thread) was used for computation. The elapsed time required to repeat the forward and backward transforms $\lfloor 300/\log_2 N \rfloor$ times was measured. The performance (Gflops) was calculated from this value. Scaling of the results by $(1/N)$ was not performed. The number of floating-point operations in a 1-D FFT of length N was assumed to be $5N \log_2 N$. To compare the performance of the optimized FFT kernel with that of other existing libraries, the performances of FFTW 3.1 [15], Intel Performance Primitive Signal Processing 5.1.1 (IPPS), Intel Math Kernel Library 8.0 (MKL), and SGI Scientific Computing Software Library 1.5 (SCSL) were also measured. To compile the FFTW library, Intel C/C++ Compiler 9.1.042 and compiler option ‘-O3’ were used.

Table 3. The specification of SGI Altix 3700

Processor	Intel Itanium 2 (Madison)
Clock Frequency/Peak Gflops	1.3 GHz/5.2 Gflops
# of Proc.	32
Level-2/3 cache	256kB/3MB
Memory	32GB
Operating System	SGI ProPack 2.4 for Linux

Table 4. The performance (Gflops) of 1-D FFTs

N	256	512	1024	2048	4096
Optimized	4.18	4.56	4.80	4.86	4.94
FFTW 3.1	2.94	3.11	3.14	2.92	3.01
IPPS 5.1.1	4.10	4.25	4.33	4.37	4.03
MKL 8.0	3.27	3.66	3.72	3.86	2.10
SCSL 1.5	2.87	3.13	3.32	3.44	3.29

Table 5. The use rate of the FPUs

N	256	512	1024	2048	4096
# of FMAs	5152	11712	26144	57920	127008
Time	2.43us	5.04us	10.6us	23.1us	49.7us
Use rate	81.4%	89.2%	94.2%	96.2%	98.2%

Table 6. The performance (Gflops) of parallel 2-D FFT of 4096²

# of PEs	1	2	4	8	16	32
Optimized	1.35	2.52	4.37	8.37	15.8	28.2
SCSL 1.5	0.82	1.44	2.54	4.36	6.91	10.6
MKL 8.0	0.62	0.95	1.51	2.83	4.92	7.99
FFTW 3.1	0.69	0.61	0.91	1.25	2.02	3.20

The optimized kernel (Optimized) we developed highly outperformed other libraries. As the result of the loop fusion, the iteration count became $O(N)$ and the performance increased as N became larger.

Table 5 shows the number of floating-point FMA operations and the use rate of the FPUs. The number of FMA operations only counts the operations which are actually completed; therefore, it does not include the operations retired in the prologue and epilogue parts of the modulo-scheduled loop. Despite that, the use rate of the FPUs is very high, reaching up to 98.2%. The rate was improved by loop fusion.

For FFTs of larger sizes, the block FFT algorithm based on six-step FFT is suitable. In the block FFT algorithm, multicolumn FFTs of the sizes evaluated in the Table 4 are computed in the cache memory.

6.2 Performance of 2-D FFT

Table 6 shows the performance of the parallel 2-D FFT of 4096². The performances with 1,2,4,8,16, and 32 threads were measured. The speed-up of the optimized kernel was about 20.8 with 32 threads. In the case of a single thread, all the data were available in the memory of the local node; therefore, no data transfer between nodes was required. In consideration of that, the speed-up was sufficiently large.

7 Concluding Remarks

We have developed a high-performance FFT on the SGI Altix 3700, which is a hardware distributed shared memory parallel computer with Itanium 2 processors. To exploit the improved performance of the modulo-scheduled loop of the IA-64 architecture, we proposed a loop fusion method for nested loops of the FFT. In contrast to conventional loop fusion methods, our method requires no index arrays, and also can be used for other processor architectures because it can be implemented with a series of generic instructions. This improved the

efficiency of executing instructions, and the use rate of the FPUs reached 98.2%. Using the Itanium 2 1.3 GHz, we achieved a performance of 4.94 Gflops (95% of peak) in the 1-D FFT, and 28 Gflops in the 2-D FFT with 32 processors. The optimized FFT kernel outperformed the other existing libraries.

Acknowledgments. This work was partially supported by CREST of JST.

References

1. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.* 19, 297–301 (1965)
2. Dunigan, T.H., Vetter, J.S., Worley, P.H.: Performance evaluation of the SGI Altix 3700. In: *ICPP*, pp. 231–240 (2005)
3. Wolf, M.E., Lam, M.S.: A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.* 2, 452–471 (1991)
4. Swarztrauber, P.N.: FFT algorithms for vector computers. *Parallel Computing* 1, 45–63 (1984)
5. Van Loan, C.: *Computational Frameworks for the Fast Fourier Transform*. SIAM Press, Philadelphia, PA (1992)
6. Intel Coporation: Itanium Architecture Software Developer’s Manual Revision 2.1 (2002)
7. Colwell, R.P., et al.: A VLIW architecture for a trace scheduling compiler. *IEEE Trans. on Computers* 37, 967–979 (1988)
8. Gwennap, L.: Intel, HP make EPIC disclosure. *Microprocessor Report* 11, 1–9 (1997)
9. Rau, B.R.: Iterative modulo scheduling: An algorithm for software pipelining loops. In: *Proc. 27th Annual International Symposium on Microarchitecture*, San Jose, CA, pp. 63–74 (1994)
10. Pease, M.C.: An adaptation of the fast Fourier transform for parallel processing. *J. ACM* 15, 252–264 (1968)
11. Linzer, E.N., Feig, E.: Implementation of efficient FFT algorithms on fused multiply-add architectures. *IEEE Trans. Signal Processing* 41, 93–107 (1993)
12. Goedecker, S.: Fast radix 2,3,4 and 5 kernels for fast Fourier transformations on computers with overlapping multiply-add instructions. *SIAM J. Sci. Comput.* 18, 1605–1611 (1997)
13. Karner, H., et al.: Multiply-Add Optimized FFT Kernels. *Math. Models and Methods in Appl. Sci.* 11, 105–117 (2001)
14. Bergland, G.D.: A fast Fourier transform algorithm using base 8 iterations. *Math. Comp.* 22, 275–279 (1968)
15. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. *Proceedings of the IEEE* 93, 216–231 (2005) (special issue on “Program Generation, Optimization, and Platform Adaptation)