2008－HPC－116 (11)
2008／8／5

SR11000/J2 における4倍精度演算を改良したFFTの実装と評価

筒井直機[†1] 吉田仁[†1]
黒田久泰[†2] 金田康正[†2]

浮動小数点演算で扱う数値は計算機上で有限桁の取り扱いとなるため，一般的には演算毎に誤差が発生する．それで計算精度を高めるために4倍精度演算を使用することがあるが，4倍精度演算はソフトウェアによるエミュレーションで実現されることが大半のため，ハードウェアで実現される倍精度演算と比べてかなり遅くなる．本研究は，これまで行ってきているベクトルデータに対する4倍精度演算の高速化で提案した手法を，4倍精度FFTの実現に適応した．その結果実行性能について，インラインアセンブラを用いてレイテンシの隠蔽を行なう事で，元の19%まで計算時間を短縮させることができた．なおこのGCCのアセンブラによる実現は，HITACHI最適化コンパイラでコンパイルした他の方法によるものとほぼ同等の計算速度となっている．

Implementation and Evaluation of FFT Using Improved Quadruple Precision Arithmetic on SR11000/J2

Naoki Tsutsui ,[†1] Hitoshi Yoshida ,[†1] Hisayasu Kuroda [†2] and Yasumasa Kanada [†2]

Floating point operations generate rounding errors in every operation, in general, because numerical operations are done with limited digits. In order to get high precision results, quadruple precision operations are sometimes used. Quadruple precision operations were realized with software emulations on almost all machines. Then, processing speed is rather slow compared to double precision operations which were realized with hardware. We adopted acceleration methods used in quadruple precision operations to quadruple precision FFT. We succeeded in reducing the computing time to 19% to the original non-optimized version by applying in-line assembler with latency hiding technique to the original sources. This method with GCC realizes almost the same performance as another method with HITACHI optimizing compiler.

1. はじめに

FFT(Fast Fourier Transform)[1] は数値計算法の分野においてよく知られた手法であり，音声処理や画像処理などで使われている．また，倍精度演算では丸め誤差の影響が無視できないために，要素数 n が大きければ大きいほど，誤差の影響が大きくなる．その誤差の改善には高精度演算，例えば4倍精度演算が有効であるがハードウェアレベルで演算が実現されていないので，演算をソフトウェアエミュレーションで実行するために計算時間が多くかかってしまう．その実現手段の一つとして，倍精度浮動小数点数を二つ用いた“double-double”精度を使用した4倍精度演算があり，ベクトルデータに対する“double-double”精度演算に関する高速化に関する研究がおこなわれている[2][3][4]，そこで我々はさらなる応用事例として，4倍精度FFTへの適用とその高速化を試みた．

2. 4倍精度のデータ型と演算について

数値計算の分野においては，解析的に困難な問題を数値的に解く計算などに浮動小数点演算を用いる方法が一般的である．ここで浮動小数点演算とは有限桁での計算のことである．したがって，多くの演算を実行

†1 東京大学大学院 新領域創成科学研究科基盤情報学専攻
Dept. Frontier Informatics, Graduate School of Frontier Sciences, The University of Tokyo
†2 東京大学 情報基盤センター
Information Technology Center, the University of Tokyo

すると誤差が蓄積し，最終的に求まる値には多かれ少なかれ誤差が含まれることになる．しかし，その誤差の量については問題及び計算手法依存であるので誤差が許容範囲内に抑えられているとは限らないこともある．そのため，数値計算を行う際には浮動小数点データ型の精度について考慮した上で行わなければならない．

2.1 IEEE 754 浮動小数点規格

IEEE 754 規格[5)6)] はカルフォルニア大学バークレー校の Willain Kahan を中心に 1985 年に制定された．この規格は，2 進浮動小数点演算規格で単精度 (32bit)，倍精度 (64bit) のフォーマットや演算ルール等を定義している．フォーマットは，例えば，符号部，指数部，仮数部のビット数などである．具体的には，**表 1** に示すとおりである．なお，IEEE 754 は 2000 年から IEEE 754r[7)] として，これまで完全に決めることができずあいまいだった部分を減らすために改訂中である．

2.2 4 倍精度データ型

SR11000/J2 上の HITACHI 最適化コンパイラで採用されている 4 倍精度データ型について述べる．この 4 倍精度のフォーマットを表 1 に示す．これは 2 つの倍精度数を用いて実装されることから “double-double” データ型と呼ばれることもある.[6)] 4 倍精度データ型は 128bit 長であるが，前述の単精度データ型や倍精度データ型の規格のように固有のデータ型が用意されているわけではなく，2 つの倍精度データ型を使って表現されている．具体的には，仮数部の桁数を増やすことで精度を上げており，指数部が倍精度数と同じなので絶対値の大きい数は表現できない．この方法を採用することで 4 倍精度用の特別なアーキテクチャが不要で，比較的高速かつ高精度に計算できるようになっている．

2.3 4 倍精度演算

2.3.1 精度確保の方法

まず実数を有限桁で近似する時に発生する丸め誤差について述べる．丸め誤差は実数を扱う演算を行う時には必ず注意しなければならない要素である．

この丸め誤差検出手順を一般化したのが次に述べるアルゴリズムである．丸め誤差を考慮した浮動小数演算は，Dekker[8)] と Knuth[9)] らの手法によってそれぞれアルゴリズムが構成されている．

また，以下全てのアルゴリズムは round to even の丸めモードを想定しており，浮動小数点演算 $\{+,-,\times,\div\}$ に対してそれぞれ $\{\oplus,\ominus,\otimes,\oslash\}$ を用いて記述する．そして浮動小数点同士の加算 $a+b$ は $a+b=fl(a+b)+err(a+b)$ と表され，$fl(a+b)$ は $a+b$ の浮動小数点演算，$err(a+b)$ は $a+b$ の演算で生じる誤差を示すとする．即ち精度の確保は、おおよその値とそれに付随する誤差を合わせることで 4 倍精度の精度を確保していることになる．

Knuth の加算アルゴリズム

TwoSum(a, b){
  $s \Longleftarrow a \oplus b$
  $v \Longleftarrow s \oplus a$
  $e \Longleftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
  return (s, e)
}

図 1 Knuth's addition algorithm

4 倍精度加算のアルゴリズム

Quad-TwoSum(a, b){
  $(t, r) \Longleftarrow$ TwoSum(a_H, b_H)
  $e \Longleftarrow r \oplus a_L \oplus b_L$
  $s_H \Longleftarrow t \oplus e$
  $s_L \Longleftarrow t \ominus s_H \oplus e$
  return (s_H, s_L)
}

図 2 Addition algorithm in quadruple precision

2.3.2 一般加算演算における誤差

任意の二つの数 $a \in R$，$b \in R$ について考える．a と b をそれぞれ計算機上の浮動小数点として表現し，a と b の加算を考える．アルゴリズムは引数 a, b に対し，$s = fl(a+b)$ と $e = err(a+b)$ を計算するものである．Knuth の加算アルゴリズムについて説明する．

Knuth の加算アルゴリズム TwoSum(a, b) は図 1 の通りである．このアルゴリズムは v と e の演算を行う所で，if 文を使うことなく全て算術演算子だけで計算を行っているのが特徴である．

2.3.3 4 倍精度加算

4 倍精度加算は，引数に大小関係の制約の無い浮動小数点演算 TwoSum を用いて構成される．4 倍精度加算アルゴリズム Quad-TwoSum(a, b) は $(s_H, s_L) = fl(a+b)$ を計算する．ただし，a, b, s はそれぞれ倍精度数二つで構成された 4 倍精度数であり，(s_H, s_L) は $s = s_H + s_L$ を表している．アルゴリズムは図 2 に示す．

ただし，上位桁で発生した誤差を下位に加算する時の誤差を考慮しないため，最終的に数ビットの欠落が

表 1 IEEE 754 data type and double-double data type.

Data type	Total bit	Exponent bit length	Exponent range	Significant bit length	Number of significant figure in decimal
IEEE 754 Single	32	8	-126~127	23	about 7.22
IEEE 754 Double	64	11	-1022~1023	52	about 15.95
IEEE 754 Single extended	≥ 43	≥ 11	$(-2^{\geq 10}+2)\sim(2^{\geq 10}-1)$	≥ 31	$\geq$ about 9.63
IEEE 754 Double extended	≥ 79	≥ 15	$(-2^{\geq 14}+2)\sim(2^{\geq 14}-1)$	≥ 63	$\geq$ about 119.26
Double-Double	128	$11 \times 2set$	-1022~1023	$52 \times 2set$	about 31.9

SR11000/J2 の 4 倍精度乗算アルゴリズム

SR11000-Quad-TwoProd(a, b){
$m_1 \Longleftarrow a_H \otimes b_L$
$m_2 \Longleftarrow a_L \otimes b_H$
$t \Longleftarrow m_1 \oplus m_2$
$p_H \Longleftarrow fl(a_H \times b_H + t)$
$e \Longleftarrow fl(a_H \times b_H - p_H)$
$p_L \Longleftarrow e \oplus t$
return (p_H, p_L)
}

図 3 Multiplication algorithm in quadruple precision on SR11000/J2

SR11000/J2 の 4 倍精度乗算アルゴリズムの改良

Improved-SR11000-Quad-TwoProd(a, b){
$m_1 \Longleftarrow a_H \otimes b_L$
$t \Longleftarrow fl(a_L \times b_H + m_1)$
$p_H \Longleftarrow fl(a_H \times b_H + t)$
$e \Longleftarrow fl(a_H \times b_H - p_H)$
$p_L \Longleftarrow e \oplus t$
return (p_H, p_L)
}

図 4 Improved multiplication algorithm in quadruple precision on SR11000/2

ある場合もある.

演算回数は TwoSum の 6Flop と加減算の 5Flop で合計 11Flop となる. 倍精度数演算が 1Flop でハードウェアで計算されるのに対して, 11 倍にもなる演算回数が必要になることが分かる.

2.3.4 4 倍精度乗算

東京大学情報基盤センターにある SR11000/J2[12] 上のコンパイラ, Hitachi 最適化コンパイラに採用されている 4 倍精度加算アルゴリズム SR11000-Quad-TwoProd(a, b) は図 3 の通りであり, $(p_H, p_L) = fl(a \times b)$ を計算する. ただし, a, b, p はそれぞれ倍精度数二つで構成された 4 倍精度数であり, (p_H, p_L) は $p = p_H + p_L$ を表している.

しかし, SR11000-Quad-TwoProd は, 2, 3 行目の演算を FMA を使ってまとめることができるため, 本実験では, その部分を改良した図 4 の Improved-SR11000-Quad-TwoProd を採用した.

3. レイテンシの隠蔽について

浮動小数点演算ユニット (FPU:Floating point number Processing Unit) の性質を考慮してデータ転送やレイテンシの隠蔽を行うことができれば, より最適化することができることになる. レイテンシは, 図 6 を使って説明すると, 1 クロック目の式 $C = A + B$, $D = A - B$ は, 式自体に依存関係がないので実行され, C と D はレイテンシが 6 クロックなので, 演算の引数として使いたい時, 7 クロック目でやっと使えるようになっている. しかし, 1 クロック目と 2 クロック目は変数間依存関係が存在しないので連続で実行可能である. こういった変数間依存関係の無い命令を連続で実行できるように命令を配置可能である. 数値計算アルゴリズムの一つである FFT には特定の形をした計算が非常にたくさん出てくる. FFT における頻出パターンの具体例を図 5 に示す. 特に現れるのが `example001`~`example003` のようなパターンである. これらのパターンは, 加減算は符号が違うだけで同じアルゴリズムであること, 同じレジスタを共有することで, 余計なロードとストアを省くことができる. また同じ形の演算が複数回連続で出てくることを利用し, レイテンシを隠蔽できる. 次によく現れるのが `example004` のようなパターンの乗算である. ここも加減算と同じように並列化し, レイテンシを隠蔽した.

4. 実　　験

4.1 実験環境

実験環境として東京大学情報基盤センターにある並列計算機 SR11000/J2 を利用した. CPU のハードウェア構成は**表 2** の通りである. L2 および L3 キャッ

FFT における頻出パターン

example001(c_0, c_1, a, b){
$c_0 = a + b$, $c_1 = a - b$
}
example002(a, b){
$t_0 = a + b$, $t_1 = a - b$
$a = t_0$, $b = t_1$
}
example003(c, a, b, f, d, e){
$c = a \pm b$, $f = d \pm e$
}
example004(c, a, b, f, d, e){
$c = a \times b$, $f = d \times e$
}

図 5 Frequent pattern examples on FFT

シュは 2 つのプロセッサで共有されており，L3 キャッシュは L2 キャッシュからキャッシュアウトしたデータが保存される．よってデータはメモリから L2，L1 キャッシュを経由してレジスタへ転送される．また，SR11000/J2 で使われている CPU は POWER5+ であり，CPU1 つあたりの FPU は 2 個となっている．

表 2 Specification of SR11000/J2.

OS	IBM AIX version 5.3
CPU	POWER5+ 2.3GHz
L1 Cache	32kB
L2 Cache	1920kB/2CPU
L3 Cache	36MB/2CPU
Number of FPU	2

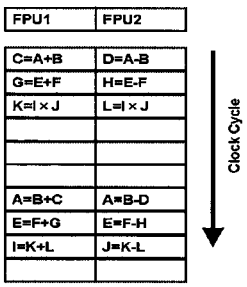

図 6 FPU operation flow

また実験に用いたコンパイラは Hitachi 最適化コンパイラ cc と gcc 4.1.1 を使用した．コンパイルは，`gcc -O3 -maix64 -mcpu=power5+ -mlong-double-128` と，`cc -Os +Op -noparallel` で行った．cc のオプションの説明は次の通りである．

- -Os 並列化，ループ融合，インライン展開などの最適化の指定．
- +Op 関係引数のポインタに依存関係が無いことを仮定する．
- -noparallel 自動並列化を行わない．
- -(no)roughquad 精度を落として高速な 4 倍精度加減算をする (しない)．

4.2 予備実験 (既存 FFT の選定)

高速 4 倍精度演算を利用した FFT を作成するに当たり，既存のライブラリに高速演算アルゴリズムを適用させることにした．候補としたライブラリは**表 3** にある FFTW[13]，FFTE[14]，FFTSS[15]，OOURA FFT[11] の 4 つである．しかし，FFTE と FFTSS は Intel SSE に特化したライブラリであり，今回の実行環境に適合しなかったために条件から外した．そして，OOURA FFT は倍精度しか対応してなかったが，手を加え，4 倍精度に対応させることで候補に残した．手を加えたものは以後 Extended OOURA FFT とする．

比較対象として，FFTW の最初に時間をかけて速いアルゴリズムを選択する `FFTW_MESURE` モードと，アルゴリズムがすでに決められている `FFTW_ESTIMATE` モードの 2 つ，Extended OOURA FFT では Radix 2, 4, Radix 2, 4, 8, Split-Radix の 3 つの合計 5 つで比較した．比較対象の FFTW と Extended OOURA FFT については，`cc -Os +Op -noparallel` でコンパイルした．データはそれぞれ 16 回実験を行い計測時間の最小値を利用した．単精度，倍精度，4 倍精度の 3 種類の型に対して実験を行ったが，いずれも Extended OOURA FFT の方が速い結果となった．4 倍精度についての結果は，図 7 の示すとおりである．グラフは，FFTW の `FFTW_MEASURE` モードでの計算時間を基準とし，それぞれの計算時間の相対比を表している．この結果より，4 倍精度演算の高速化を Extended OOURA FFT に施すことを決めた．

4.3 実験内容

本実験では，Radex 2, 4, Radix 2, 4, 8, Split-Radix のうち Radix 2, 4 のソースコードを元に long double の加減算と乗算部分を前述の double-double アルゴリズムに展開したもの，さらにアセンブラでレイテンシを埋める形で展開したものを用意した．

- `normal` 版
  long double の演算部分に何も手を加えていないもの．
- `double-double` 版
  加算部分と乗算部分を double-double アルゴリズ

表 3 Features of FFT Libraries.

Library	Program Language	Optimization Target	For Single	For Double	For Quadruple	For MPI
FFTW	C	Multi Architecture	○	○	○	×
FFTE	Fortran	Especially Intel SSE	×	○	×	○
FFTSS	C	Especially Intel SSE	×	○	×	○
OOURA FFT	C	Algorithm	×	○	×	×
Extended OOURA FFT	C	Algorithm	○	○	○	×

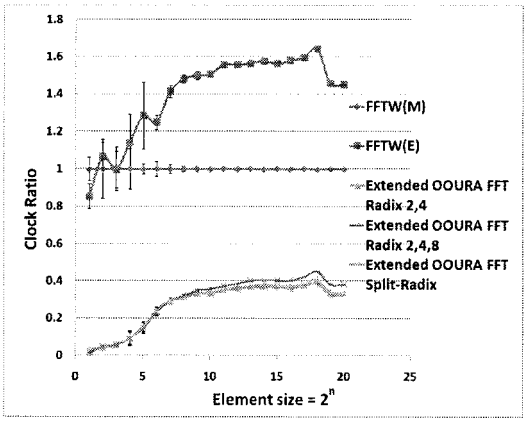

図 7 Long Double Precision Relative Comparison based on FFTW(M) clock.

ムで展開したもの．ただし，乗算部分は一部 FMA を使わなければならない箇所があり，その部分は関数化して使用した．

- asm 版
  加算部分と乗算部分をレイテンシを隠蔽するようにインラインアセンブラで展開したもの．ただし，cc ではインラインアセンブラが使えないので gcc でのみ実験を行った．

実験は前述の 3 種類のソースコードとコンパイラで条件を変え，計 6 種のプログラムを作成し，$2^1 \sim 2^{20}$ の要素データに FFT をかけるのに要した時間を計測する実験を各 1024 回行った．6 種類のプログラムについては，以下の次の通りである．

- `gcc normal`
  `normal` 版ソースを `gcc` でコンパイルしたもの．
- `gcc double-double`
  `double-double` 版ソースを `gcc` でコンパイルしたもの．
- `gcc asm`
  `asm` 版ソースを `gcc` でコンパイルしたもの．
- `cc -roughquad normal`
  `normal` 版ソースを `cc -roughquad` でコンパイルしたもの．
- `cc -roughquad double-double`
  `double-double` 版ソースを `cc -roughquad` でコンパイルしたもの．
- `cc -noroughquad normal`
  `normal` 版ソースを `cc -noroughquad` でコンパイルしたもの．

4.4 実験結果と考察

各実験の計測時間平均値を図 8 に示し，実行時間比で見れるようにしたものを図 9 に示す．ここで，実行時間比のグラフとは，`gcc normal` の実行時間に対する各実験の実行時間を比で示したものである．

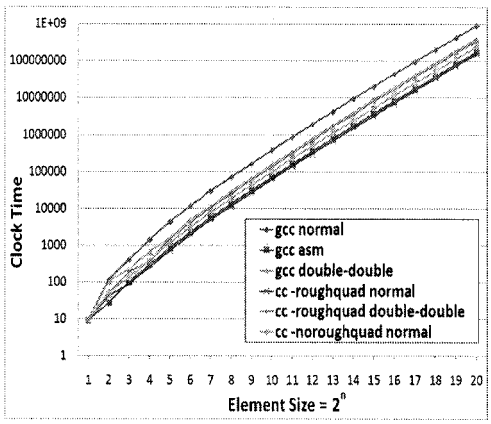

図 8 Computation time of quadruple precision

まず，図 9 において，n が小さい時はばらつきがあるものの，$n \geq 5$ では各実験の実行時間の比率がほぼ一定であることがわかる．`gcc normal` に対する実行時間の比率で考えると，$n = 20$ の時においては，`gcc double-double` は，原則 C 言語で書かれているにもかかわらず，42% にすることができ，高性能な `cc` でコンパイルした `cc double-double` は 37% となった．また，`gcc asm` は `cc -roughquad normal` の 17% とほぼ同等の 19% となった．最終的に今回の実験では，既存の Hitachi 最適化コンパイラより提案

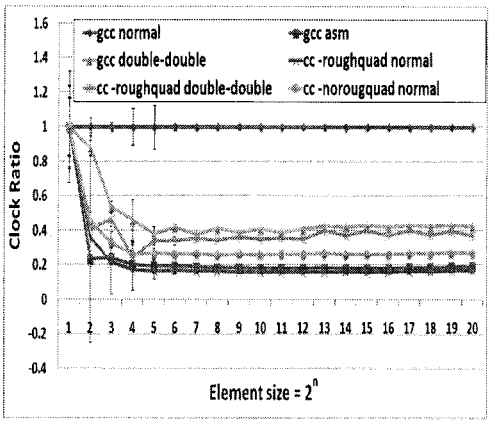

図 9 Relative comparison based on GCC normal

手法を速くできなかったが，gcc normal 版と比べて，double-double の演算を展開するだけでも 42% の時間にすることができ，また，4 倍精度演算を局所的にインラインアセンブラで レイテンシの隠蔽を意図した gcc asm 版では，19% の時間にできた．元々5 倍程度遅かった gcc を用いて Hitachi 最適化コンパイラとほぼ同程度の性能を引き出すことができた．

5. おわりに

今回 SR11000/J2 において高速 4 倍精度演算を採用した FFT を実装した．手法として，double-double アルゴリズムを C 言語で展開したり，頻出部分をインラインアセンブラでレイテンシの隠蔽を意図して演算の並列化を行った．その結果，インラインアセンブラを採用したものは，Hitachi 最適化コンパイラと比べて性能面で劣るコンパイラ gcc においてでも，ほぼ同等の性能まで引き上げることができた．今後の課題としては，異なるアーキテクチャにおいての同手法の適用や，今回より大きな範囲でレイテンシの隠蔽が考えられる．

参 考 文 献

1) J.W.Cooley and J.W.Tukey. An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation, 19:297-301, 1965
2) 小武守, 藤井, 長谷川, 西田：倍精度と 4 倍精度の混合型反復法の提案，HPS2007, 論文集 pp.9-16, 2007
3) 永井貴博，吉田仁，黒田久泰，金田康正：SR11000 モデル J2 における 4 倍精度関和演算の高速化，情報処理学会論文誌：コンピューティングシステム,Vol.48 No.SIG13(ACS19)pp.214-222, 2007
4) Takahiro Nagai, Hitoshi Yoshida, Hisayasu Kuroda1,and Yasumasa Kanada：Fast Quadruple Precision Arithmetic Library on Parallel Computer SR11000/J2：ICCS 2008, Part I, LNCS 5101, pp. 446-455, 2008.
5) Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)
6) IEEE. 2001. 754 Revision. http://grouper.ieee.org/groups/754/.
7) IEEE 754r draft. http://754r.ucbtest.org/drafts/archive/2006-10-04.pdf
8) T. J. Dekker. A floating-point technique for extending the available precision. Numerische Mathematik,10:224-242, 1971.
9) D. E. Knuth. The art of computer programming Vol:2 Seminumerical algorithms Third Edition. Addison-Wesley, 1998.
10) The GNU Compiler Collection. http://www.gnu.org/software/gcc/index.html.
11) 京都大学数理解析研究所，大浦拓也's HP. http://www.kurims.kyoto-u.ac.jp/~ooura/.
12) HITACHI SR11000/J2 http://www.hitachi.co.jp/.
13) FFTW http://www.fftw.org/.
14) FFTE http://www.ffte.jp/.
15) FFTSS http://www.ssisc.org/fftss/.