Vol.2011-HPC-130 No.45
2011/7/28

高性能な 8 倍精度浮動小数点演算機構の実現

泊　　久　信[†1]　　平　木　　敬[†1]

計算機が高性能になったことにより，より大きな問題を解くことができるようになった．入力が計算結果として出力されるまでに演算器を通る回数も，問題の規模と反復回数に応じて大きくなった．計算アルゴリズムの中には，演算器を通る回数が増えると誤差が蓄積していくものがある．このようなアルゴリズムを，より高性能な計算機を用い大規模な問題に対して適用するためには，より高精度な浮動小数点演算が必要である．

ところが，高精度な浮動小数点数を扱うハードウェアは市販品としては少なく，結果としてソフトウェア実装を用いるのが一般的であった．ソフトウェアによる実装は幅広い環境で動作させることができる利点がある一方，性能を出しにくいという欠点がある．性能が出ない場合，そもそも高精度な浮動小数点数を扱う必要性は低い．

本研究では，IEEE 754 規格を拡張して，8 倍精度 (256-bit) 浮動小数点数を定義した．評価では，POWER7 マシンでの倍精度の演算と，8 倍精度演算の 64 ビット PowerPC アセンブリでの実装との性能を比較し，8 倍精度が倍精度の 1/44 程度の性能の劣化になることを確認した．ハードウェア実装として，CPU の FSB に FPGA が結合された，Convey HC-1 を用いて，高性能な演算器を実装した．この FPGA ベースの実装を用いた場合，POWER7 の 8 コアのシステムに比べ，約 4.5 倍の 8 倍精度浮動小数点処理性能を実現した．

High-performance Octuple Precision Floating Point Processor

Hisanobu Tomari[†2] and Kei Hiraki[†2]

The faster the processor becomes, the larger grows the size of the problem that the processor is capable of solving. The number of operations that are applied to input data is subject to the size and the number of iterations. There are algorithms where the error accumulates as the size or the number of iterations increases. To apply these algorithms to the larger set of problems that are solved on the next-generation computers, a higher-precision floating point format is required.

Notwithstanding the need, there are little support for arithmetic on floating point numbers of quadruple or more precisions. When they really needed it they tend to implement them using software. Using software to process higher-precision floating point number benefits from portability, but at the grave cost of the performance. When the performance is limited, we often do not need higher precision floating point numbers in the first place.

We propose an extension to the IEEE 754 floating point number formats to define a octuple-precision (256-bit) floating point numbers. We compared the performance of our octuple precision implementation to the double-precision operations on IBM POWER7. On POWER7, octuple precision operations take about 44 times more processing time than double-precision counterparts. We implemented FPGA-based arithmetic unit for the data format on Convey HC-1 system, where FPGA chips are connected to the host using the front side bus. On this system, octuple precision operations are 4.5 times faster than those on the 8-core POWER7 system.

1. はじめに

処理装置の性能は，浮動小数点演算性能を含めて，時間に対して指数関数で表される上昇をしている．この上昇した計算性能を，数値的な問題を解くうえで弱スケーリングに用いると，初期値が結果に至る過程で通る演算器の数は増加する．丸め・情報落ちがあっても，反復の回数を増やして演算器を通せば通すほど誤差が小さくものもある一方，ここで生まれた誤差が結果に降り積もるアルゴリズムも存在する．また，問題サイズを大きくすることで条件数が大きくなるアルゴリズムがあり[12)]，この場合マシンイプシロンによる誤差が拡大される．高速な計算機を用いることで，大規模な問題を解けるようになった一方，その分誤差が増える場合があるということである．

IEEE 754-2008 のうち，よく使われている浮動小数点形式は単精度，倍精度，4 倍精度

表 1　IEEE 754 の主な浮動小数点形式の比較

	語長	指数部長	仮数部長	最小の正の正規化数	最大値
単精度	32	8	23	2^{-126}	$\sim 2^{128}$
倍精度	64	11	52	2^{-1022}	$\sim 2^{1024}$
4 倍精度	128	15	112	2^{-16382}	$\sim 2^{16384}$

†1 東京大学大学院情報理工学系研究科
†2 The University of Tokyo

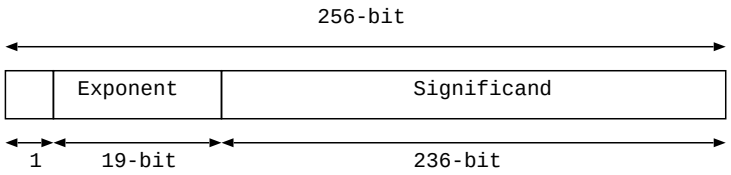

図 1 本研究で扱う 8 倍精度浮動小数点形式

である[6]．浮動小数点数の処理速度の向上を考慮すると，将来的には 4 倍精度でも不足すると考えられる．4 倍精度より高い精度が必要な場合，主記憶での配置の利便性および将来の更なる演算精度の向上に耐えうる精度として，256-bit の幅を持つ，8 倍精度が有力である．8 倍精度は IEEE 754 では定義されていないので，以下のように定義する．IEEE 754 規格の表現 (**表 1**) の際に考慮されたことを生かすため，8 倍精度の表現がその延長となるようにした．指数部の長さを決めると，その他の欄の長さが決まる．IEEE 754-2008 の Binary interchange floating-point format に従い，指数部の長さは $\mathrm{round}(4\times\log_2(k))-13$ に $k=256$ を代入して得られる指数部の長さは 19 ビットである．以上のように定義したのが，本研究で使用する 8 倍長浮動小数点形式である (**図 1**)．IEEE 754 形式に合わせて，指数部はオフセット付きの正数で表現する．仮数部の扱いも IEEE 754 形式に準ずる．この形式の正数の値域は，$2^{-262142}$ から約 2^{262144} までである．

2. 処 理 方 法

8 倍精度浮動小数点数の演算のハードウェア実装を述べる前に，どのような処理を行う必要があるのかまとめる．IEEE 754 の倍精度浮動小数点数の演算と手順は同じだが，ワード幅が広いため定数部分が異なる．IEEE 754 の浮動小数点演算では，Gradual underflow が定義されている．これは，指数部が 0 で仮数部が 0 でないとき，正規化数の場合に仮数部の最上位に付加されているとみなすビット 1 をつけず，正規化数で表現できる最小の正値以下の正値を扱えるようにしたものである．今回の実装は，そもそも指数部が相当分以上大きくなっているので，Gradual underflow はサポートしていない．丸めモードも IEEE 754 では複数定義されているが，今回の実装は切り捨てのみサポートしている．

加算は，まず仮数部の桁の位置を合わせるため，指数部の差を求める．つぎに，求めた差の数だけ，仮数部をシフトする．これによって，対応する桁が同じ場所に移動する．このシフトで失われるビットが情報落ちする．IEEE 754 では先頭のビット 1 が省略されているが，指数部が 0 の場合，正規数でないか表現されている値が 0 で，いずれも最上位のビットが 1 ではない．よって，指数部が 0 でなければ，仮数部の最上位ビットの 1 ビット上に 1 を補う．入力された浮動小数点形式の符号を見て，符号ビットを補う．負の数の場合は 2 の補数表現に直す．以上の操作により，桁合わせされた仮数部は 238 ビットの数になっている．この 2 つの仮数部を 238 ビットの加算器に通す．この結果は 2 の補数表現になっている．符号ビットが最下位ビットから数えて 238 ビット目になる．IEEE 754 では，仮数部は常に正数なので，符号ビットが 1 になっている場合は 2 の補数表現から正の数に戻す．加算器を通る値が両方正なら，最上位ビットの 1 の場所が 237 ビット目あるいは 236 ビット目だが，負の数の場合は最上位の 1 のビットを探す必要がある．最上位の 1 のビットが，237 ビット目になるようにシフトし，シフトした量だけ，指数部に値を加算する．最後に，238 ビット加算器からの出力の符号ビットを結果に追加して，最終的な出力にする．以上をまとめた擬似コードを**図 2** に示す．

乗算は，まず指数部を加算し，指数部がオフセット表現のために加算により 2 倍になるオフセット部分を引く．次に，符号を入力の符号の排他論理和をとることで求める．次に，仮数部の表現で省略されている最上位ビットの 1 を指数部が 0 より大きい場合は付加し，237 ビットの指数部を得る．この 237 ビットの指数部を乗算器に通す．乗算器の出力の最上位の 2 ビットが 00 の場合は出力は 0 である．最上位のビットが 0 の場合は，最上位ビットと次のビットを飛ばして上から 236 ビットが仮数部である．この際，出力される指数部は求めた指数部より 1 小さい．最上位のビットが 1 なら，最上位のビットを飛ばして上から 236 ビットが仮数部である．出力する指数部が 0 の場合，あるいはオーバーフローしている場合は，それぞれ 0 または符号に応じた無限大を返す必要がある．以上をまとめた擬似コードを**図 3** に示す．

倍精度浮動小数点数から変換は，仮数部についてはビット幅が広くなるだけなので，倍精度浮動小数点数の仮数部の下位に 0 を補うことで処理する．指数部は，オフセットが違うため，差を補正する．倍精度浮動小数点数への変換は，値域が小さくなるため，オーバーフロー・アンダーフローさせる場合がある．整数からの変換は，絶対値をとり，最高位の 1 ビットの場所を指数部に表現し，仮数部は最高位の 1 ビットを削った表現を格納することで実現できる．本研究では除算をサポートする命令 $(1/\sqrt{x})$ は対象としていないので，除算は和・積を使って実現する必要がある．

Require: $\text{input}_1 \geq \text{input}_2$
$s_1, e_1, f_1 \leftarrow \text{input}_1[255], \text{input}_1[254{:}236], \text{input}_1[235{:}0]$
$s_2, e_2, f_2 \leftarrow \text{input}_2[255], \text{input}_2[254{:}236], \text{input}_2[235{:}0]$
$b_1, b_2 \leftarrow (e_1 > 0)?1 : 0, (e_2 > 0)?1 : 0$ { ゼロと非正規化数なら 1 を補わない }
$F_1[237:0], F_2[237:0] \leftarrow 0\&b_1\&f1[235:0], 0\&b_2\&f2[235:0]$
$F_2 \leftarrow \text{ShiftRight}(F_2, (e_1 - e_2))$ { 指数部の差だけシフトし，桁を F_1 と合わせる }
$F_1 \leftarrow e_1?\text{negate}(F_1) : F_1$
$F_2 \leftarrow e_2?\text{negate}(F_2) : F_2$ { 通常の加算器を通すため，負の数を 2 の補数表現に }
$F[237:0] \leftarrow F_1[237:0] + F_2[237:0]$
$S \leftarrow F[237]$ { 符号ビットを抽出 }
$F \leftarrow S?\text{negate}(F) : F$ { 仮数部は正数として格納する }
$l \leftarrow 1 - \text{CountLeadingZeros}(F[237:0])$ { 最上位ビットの位置 }
if $F = 0$ **then**
  return $S\&255'\text{d}0$ { 仮数部が 0 になったら結果は 0}
else if $l > 0$ **then**
  $F \leftarrow \text{ShiftLeft}(F, l)$ { 仮数部の桁を合わせる }
else
  $F \leftarrow \text{ShiftRight}(F, -l)$
end if
$E \leftarrow e_1 + l$ { 仮数部のけた合わせに応じて，指数部を更新 }
return $S\&E[18:0]\&F[235:0]$

図 2 8 倍精度の加算アルゴリズム

$s_1, e_1, f_1 \leftarrow \text{input}_1[255], \text{input}_1[254{:}236], \text{input}_1[235{:}0]$
$s_2, e_2, f_2 \leftarrow \text{input}_2[255], \text{input}_2[254{:}236], \text{input}_2[235{:}0]$
$b_1, b_2 \leftarrow (e_1 > 0)?1 : 0, (e_2 > 0)?1 : 0$ { ゼロと非正規化数なら 1 を補わない }
$F_1[236:0], F_2[236:0] \leftarrow b_1\&f1[235:0], b_2\&f2[235:0]$
$S \leftarrow \text{xor}(s_1, s_2)$ { 積の符号は排他論理和 }
$f[473:0] \leftarrow F_1[236:0] \times F_2[236:0]$
if $f[473] = 1$ **then**
  $F[235:0] \leftarrow f[472:236]$
  $E[19:0] \leftarrow (0\&e_1) + (0\&e_2) - 20'\text{h3FFFF}$
else if $f[472] = 1$ **then**
  $F[235:0] \leftarrow f[471:235]$
  $E[19:0] \leftarrow (0\&e_1) + (0\&e_2) - 20'\text{h3FFFF} - 1$
else
  return $S\&255'\text{d}0$ { 最上位ビットが以上 2 通りでない場合は結果は 0}
end if
if $E[19] = 1$ **then**
  return infinity, sign=S { オーバーフローしたら無限大を返す }
else if $E = 0$ **then**
  return $S\&255'\text{d}0$
else
  return $S\&E[18:0]\&F[235:0]$
end if

図 3 8 倍精度の乗算アルゴリズム

3. 実　　装

3.1 ソフトウェア実装

まず，ソフトウェア実装では，データ構造や倍精度浮動小数点数との変換部分は C 言語で記述した．C 言語にはキャリーフラグを効率的に使う方法がないため，加算・積算でプロセッサのキャリーフラグを使うことで高速化が見込める部分をアセンブリで記述した．今回の実装においては現在高性能計算で用いられることが多い 64 ビット PowerPC と，AMD64 での実装を行った．キャリーが有効に使える部分は，加算では，238 ビット入力が 2 つある加算器で，積算では，236 ビット積算機である．これらの部分は整数演算でよいため，一般的な多倍長演算と同じように，プロセッサに実装されている 64 ビット加算器を使うことができる．この部分をアセンブリで記述すると，AMD64 および 64 ビット PowerPC の両方でレジスタのスピルのない記述ができる．積については，AMD64 では 64 ビット数 2 入力で，128 ビット出力を得ることができる．ところが，初期の POWER では MQ レジスタでこの機能が実現されていたものの，PowerPC からは削除されたため[4]，64 ビット PowerPC 実装では 32 ビット 2 入力で 64 ビット出力の積算機を使う必要があった．キャリーフラグがないアーキテクチャ(e.g. IA64) および C 言語のみで多倍長加算を実現するためには，出力が入力のいずれよりも小さいかどうかでオーバーフローが発生したかを確かめる方法が使える．

3.2 ハードウェア実装

CPU で非標準なデータの処理を記述すると，固定的な処理を行うにも拘らず動的にデータパスを切り替えて演算器を利用するので，非効率である．FPGA を用いると，固定された演算を CPU に比べて効率よく処理することができる．FPGA の問題点は，計算アルゴリズムの一部として 8 倍精度演算を行うとき，従来の CPU も通常は同時に必要になるため，CPU との接続が必要になる点である．通常，FPGA と CPU との接続には PCI-Express などの規格が用いられるが，8 倍精度演算は細粒度の演算も行われることを想定しているため，PCI-Express のような高レイテンシ・低バンド幅な接続では，命令の発行コストおよびデータ移動のコストの影響で利用効率が下がってしまう．

これらの問題を解決するため，ハードウェアの実装には，Convey HC-1 を用いて行った．Convey HC-1 は，主 CPU (Intel Xeon 5138) と FPGA 群が FSB で接続されている計算機である[1]．FPGA 群は，Application Engines (AE) と呼ばれる 4 つの FPGA と，8 つの Memory Controllers (MC)，Application Engine Hub (AEH) によって構成されている

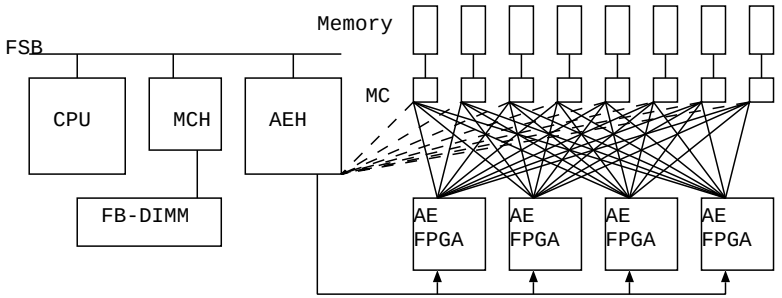

図 4　Convey HC-1 のブロック図

(**図 4**)．AE は，利用者が必要に応じて再構成できるようになっている．今回，この AE に 8 倍精度演算器を実装した．MC は，それぞれが専用の DIMM と接続されている．すべての MC を同時に利用した場合，FPGA からメモリまでのバンド幅は 80 GB/s である．すべての AE からすべての MC に配線されている．AEH は，主 CPU の FSB とのインターフェースとして動き，AE で仮想アドレスによるデータアクセスを実現している．主 CPU からは，AEH を通してすべてのメモリに直接アクセスできるが，MCH に接続されているメモリよりもアクセスが遅いという NUMA 構成になっている．

Convey のアーキテクチャでは，8 つあるメモリコントローラーからアクセスできる領域はそれぞれ限定されている．本実装では空間全体を 1 KB ごとに区切り，それをメモリコントローラーに順番に割り付けていく Binary Interleave を利用した．演算器の 2 つの入力と出力がそれぞれ異なるメモリコントローラに跨っていると，それぞれの演算器の入出力を，8 つのメモリコントローラーからルーティングする必要が生じる．回路を単純化するために，2 つの入力と出力すべてが同じメモリコントローラーから読み書き可能になるようにソフトウェアで入出力を再配置するようにした．MC からのインターフェースはクロックの立ち上がりと立ち下がりでそれぞれ要求・返答を受けとるため，インターフェースあたり 2 つずつ演算器を実装することになる．8 倍精度浮動小数点数の 1 語が 256 ビットと広く，読み込みに 4 サイクル必要なため，演算器の最大の利用効率は 1/4 である．

FPGA 側で実行する命令を発行するのにオーバーヘッドが予想されるため，ある程度まとめて計算を実行するのが望ましい．そのため，今回実装した命令はベクトル命令で，メモリから直接データを読み込み，演算を適用し，メモリに直接データを書き戻す動作をする．

以上，演算器を載せている AE FPGA が 4 つあり，それらはそれぞれ 8 つの MC と接続

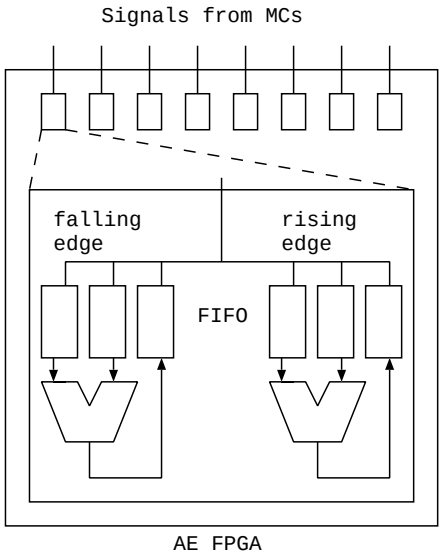

図 5 AE FPGA 内部の構成

しており，MC に対してそれぞれ 2 つずつ演算器が載せられている構成になる (**図 5**)．プログラムから演算命令を呼ぶ際には，FPGA 群の命令列が書かれたアセンブリの関数を呼び出し，その中で関数のパラメータを AE のレジスタにセットし，計算が終了するのを待つようになっている．

4. 評　　価

まず，ソフトウェア実装の 8 倍精度浮動小数点数処理の性能を評価するため，高性能な計算機を用いて，CPU 内蔵の倍精度浮動小数点計算機構と性能を比較した．2 つの入力ベクトルを与え，ベクトルの各要素に演算を適用し，結果を結果ベクトルに書き込む動作を，ベクトル長を変えて実行し，処理時間を計測した．計測に利用したプロセッサは IBM POWER7 である．実験時の構成は**表 2** の通りである．8 コアプロセッサのすべての演算器を利用するため，OpenMP を用いてループの並列化を行った．測定時には，numactl を使って，1 つのダイのみを使い，各コア 1 スレッドのみ利用した．

図 6 が測定結果である．倍精度の浮動小数点数の処理が，ベクトル長を長くしていくと性

表 2 ソフトウェア実装の性能測定に利用した計算機の諸元

機種名	IBM Power 740 Express
CPU	2× POWER7, 3550 MHz, 計 16 コア, 64 スレッド
キャッシュ	L1I/D=32/32 KB, L2=256 KB, L3=4+32 MB[7)]
主記憶	64 GBytes
ソフトウェア	Linux 2.6.32.36, GCC 4.6
フラグ等	-O3 -mcpu=power7 -fopenmp, OMP_NUM_THREADS=8

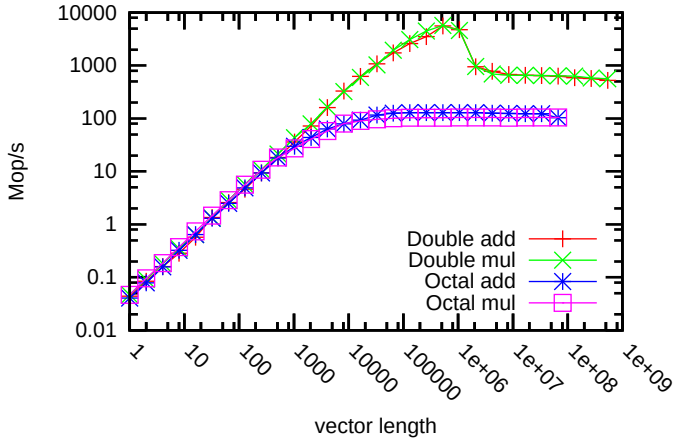

図 6 ベクトル長に対するソフトウェア 8 倍精度演算と倍精度演算の性能　横軸は要素数

能低下しているのは，入力及び結果がキャッシュに収まらなくなり，倍精度浮動小数点数の処理はメモリからの読み出しより早いため，入力データが届かないことによるものである．8 倍精度の浮動小数点数では，ベクトル長を長くしても性能が低下することはない.8 倍精度浮動小数点数は, 256 ビットで, 倍精度の 4 倍のデータ量があるが, 演算をソフトウェアで行っている．これによって、データ転送ではなく，演算が律速しているからである．ベクトル長が短いときに 8 倍精度演算と倍精度演算の性能に差が小さいのは，スレッド作成・同期のオーバーヘッドが大きいことによると考えられる[5]．今回の実装では，ベクトル演算の度に OpenMP でスレッドを作成しているため，演算処理時間が短い倍精度計算では，OpenMP の処理時間が支配的になっている．倍精度演算の最高性能は 5,624 Mop/s で，8 倍精度演算の最高性能は 129 Mop/s なので，8 倍精度演算は倍精度の演算の約 44 分の 1 の性能を達成している．

ハードウェアでの実装は，ソフトウェア実装との性能の比較によって行った．ソフトウェアでの評価と同様，ベクトル長を変化させた場合の性能をプロットしている (**図 7**)．ソフトウェアでの処理は，図 6 のものと同じ値である．ベクトル長が短いと，今回の実装では，演算機がアクセス可能なメモリ領域が限定されているため，一部の演算機しか利用することができず，性能が低い．ベクトル長が 64K 要素より長くなると，利用可能な演算機の数が増えていき，それに応じて性能が向上する．今回の実装では，メモリからのデータの読み出しに少なくとも 4 クロックかかっているため，80 MHz で演算機が 64 個の合計で 5120 Mop/s の利用可能な演算能力の 1 割にあたる 537 Mop/s になっている．1 つの 8 倍精度演算は，2 つの入力を読み込んで 1 つの出力をメモリに書き戻しているので，96 バイトの入出力が発生する．537 Mop/s を実現するのにメモリバンド幅を 51.5 GB/s 利用しており，これは Convey 社が提供している単精度浮動小数点演算のデザインで利用可能なメモリバンド幅と一致していることから，ハードウェアを利用した場合のボトルネックはメモリバンド幅であるといえる．命令発行コストが依然高く，細粒度の演算で利用する場合はソフトウェア実装を利用してハイブリッドに処理を行うことで，最適な性能を発揮できる．

5. 関 連 研 究

4 倍精度は，SPARC アーキテクチャでは表現形式として利用可能なものの[9]，実装ではこの形式はハードウェアでは処理されず[8]，コンパイル時にライブラリコールに置き換えるか，OS が代替して処理を行っている．また，GCC には，4 倍精度演算をソフトウェアで行う libquadmath が含まれたため，今後利用する機会が増えると考えられる．高精度

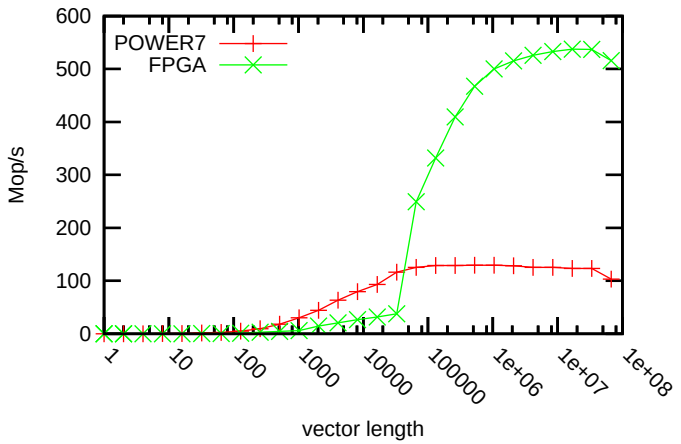

図 7 ベクトル長に対するソフトウェア 8 倍精度演算と FPGA 実装の性能　横軸は要素数

な演算を専用ハードウェアによって高性能に実現するものとして，GRAPE-MP がある[3]．GRAPE-MP は 4 倍精度演算を扱う．GRAPE-MP は PCI-Express を介してホスト計算機と接続するので，利用者がアルゴリズムを GRAPE-MP の計算方式に合うように改変する必要がある．本手法は，高速な FSB を利用して，より細粒度で，より高精度な 8 倍精度での演算を提供するものである．

256 ビットの幅を持つ浮動小数点形式は，Crandall らにより実装されている[2]．この実装では，ソフトウェアでの高速処理が目標のため，指数部を 32 ビットにし，仮数部は 2 の補数表現になっている．将来的に IEEE 754 でもより高精度な表現が標準化されることを見込むと，現在の IEEE 754 規格の延長として 8 倍精度を定義した我々の方がデータ/精度互換性の点で有利である．

GPGPU は，すぐれた並列化コンパイラと価格の低さで近年注目されている．楠木らにより，GPGPU を用いた 8 倍長を含む BLAS が実現されている[10]．本稿でのソフトウェア実装は，アセンブリを用いて最適化したことにより，楠木らの Core i7-920 で OpenMP を用いた 8 倍精度の実装の 3 から 4 倍の性能を実現した．また，FPGA を用いた実装では，Tesla C1060 の 3 倍，Tesla C2050 の 7 割程度の性能を実現している．今後の実装で演算機の利用効率を向上させることで，GPU の性能を越すことが可能である．

FPGA を用いた 4 倍精度浮動小数点演算機構は中里らにより実現されている[11]. 本稿で

は，ホストシステムとの接続を含めて，より高精度である 8 倍精度浮動小数点数の演算性能を評価し，中里らの評価の 2 倍の性能を実現している．また，本研究は今後の改良で性能をさらに改善する見込みである．

6. おわりに

本稿では，8 倍精度の浮動小数点表現を，一般に用いられている IEEE 754 の延長として定義した．また，将来的に有用性が見込まれるこの表現を用いて，現在のハードウェアで高精度な演算を実現するため，高速な演算処理機構を，アセンブリを用いた高速なソフトウェアと，FPGA を FSB に接続した高速汎用ハードウェアの 2 通りで実現した．ソフトウェアの実装では，CPU 内蔵の FPU を用いた倍精度の演算に比べ，約 1/44 の性能で 8 倍精度の演算を実現した．これは, 既存の実装の 3-4 倍の性能である．ハードウェア実装では，命令発行のオーバーヘッドを減らすため，メモリから直接データを読み込み，直接メモリに結果を書き出す形のベクトル命令を実装した．このハードウェア実装では，高性能計算機上でソフトウェアを用いて同等の処理を行う場合に比べて最大で約 4.5 倍の命令実行性能を実現した．粒度の大きい演算では, プロセッサに 8 倍精度浮動小数点数演算回路が組み込まれない限り,FPGA を用いた処理は有用なアプローチである．プロセッサ外部のハードウェアでは細粒度の演算のオーバーヘッドが大きい場合でも，計算の粒度に応じてソフトウェアとハードウェアによる実装を切り替えて，より高精度な浮動小数点演算を現在の計算機システムで高性能に実現することができる．

ハードウェアの実装の最適化が不十分で，メモリのアラインメントの制限があること，および演算器の利用効率が低いことが課題である．メモリのアラインメントの制限に関しては，演算器ブロックとメモリ・コントローラーのブロックの間にネットワークを実装することで解決できる．これによって，演算機の利用効率も上げることができる．また，除算サポート $(1/\sqrt{x})$ も今後実装したい．最後に，こういった高精度の要求は，計算機の演算性能の向上に伴って増えることが予想されるため，4 倍精度より高い精度を標準化する必要がある．

参 考 文 献

1) Brewer, T.: Instruction Set Innovations for the Convey HC-1 Computer, *Micro, IEEE*, Vol.30, No.2, pp.70 –79 (online), DOI:10.1109/MM.2010.36 (2010).
2) Crandall, R. and Papadopoulos, J.: Octuple-precision floating point on Apple G4 (2002).
3) Daisaka, H., Nakasato, N., Makino, J., Yuasa, F. and Ishikawa, T.: GRAPE-MP: An SIMD Accelerator Board for Multi-precision Arithmetic, *Procedia Computer Science*, Vol.4, pp.878 – 887 (2011). Proceedings of the International Conference on Computational Science, ICCS 2011.
4) Diefendorff, K., Oehler, R. and Hochsprung, R.: Evolution of the PowerPC architecture, *Micro, IEEE*, Vol. 14, No. 2, pp. 34 –49 (online), DOI:10.1109/40.272836 (1994).
5) Fredrickson, N. R., Afsahi, A. and Qian, Y.: Performance characteristics of openMP constructs, and application benchmarks on a large symmetric multiprocessor, *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, New York, NY, USA, ACM, pp. 140–149 (online), DOI:http://doi.acm.org/10.1145/782814.782835 (2003).
6) IEEE: IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2008*, pp.1 –58 (online), DOI:10.1109/IEEESTD.2008.4610935 (2008).
7) Sinharoy, B., Kalla, R., Starke, W.J., Le, H.Q., Cargnoni, R., VanNorstrand, J.A., Ronchetti, B.J., Stuecheli, J., Leenstra, J., Guthrie, G.L., Nguyen, D.Q., Blaner, B., Marino, C.F., Retter, E. and Williams, P.: IBM POWER7 multicore server processor, *IBM Journal of Research and Development*, Vol.55, No.3, pp.1:1 –1:29 (online), DOI:10.1147/JRD.2011.2127330 (2011).
8) Tirumalai, P., Greenley, D., Beylin, B. and Subramanian, K.: UltraSPARC: compiling for maximum floating-point performance, *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, pp. 408 –416 (online), DOI:10.1109/CMPCON.1996.501803 (1996).
9) Weaver, D.L. and Germond, T.(eds.): *The SPARC Architecture Manual version 9*, SPARC International, Inc.
10) 楠木大地，中山星空，越智浩之：GPU による多倍長精度 BLAS の開発，ICT ソリューションアーキテクト育成プログラム: ソリューション型特別プロジェクト最終報告書 (2010).
11) 中里直人，石川　正：高精度浮動小数点演算器の FPGA での実装 (応用 1)，電子情報通信学会技術研究報告. RECONF, リコンフィギャラブルシステム， Vol.108, No.48, pp. 7–11（オンライン），入手先⟨http://ci.nii.ac.jp/naid/110006881681/⟩ (2008-05-15).
12) 名取　亮，塚本敦子：行列の条件数の推定，数理解析研究所講究録， Vol.483, p.212 (1983).