Vol.2015-HPC-152 No.4
2015/12/16

短尺浮動小数点形式の検討

椋木 大地[1,a)]　今村 俊幸[1,b)]

概要：本稿では計算速度の向上と省電力化を目的とした短尺浮動小数点形式（short length floating-point formats）を提案する．今日，ほとんどの計算機上では IEEE 規格の単精度・倍精度の 2 種類の浮動小数点表現が使用されている．しかしユーザが真に必要としている精度と IEEE 規格の単精度・倍精度との間にはズレがあり，計算において無駄なデータのやりとりが生じている可能性がある．もし必要な情報のみを格納することができれば，処理速度の向上と省電力化が期待できる．本研究では，従来の単精度・倍精度型から仮数部ビットを 8 ビット刻みで削った短尺浮動小数点形式の実装を検討し，x86 CPU および NVIDIA GPU 上の基本的な線形計算カーネルに適用して，演算性能と電力性能を評価する．

1. はじめに

ポストペタスケール時代に向けた高性能計算分野の研究課題は，これまでの演算コスト削減からデータアクセスコスト削減へと関心が移りつつある．その理由として，(1) プロセッサあるいはシステム全体の演算器数が急増する一方で，相対的に進化速度の遅いメモリ・通信の性能が不足し，計算速度がメモリ・通信の性能に律速される傾向となる．また演算能力に対してメモリ容量も不足する，(2) 計算性能の向上と手法の確立によって，ビッグデータ解析，データマイニング，機械学習といったデータアクセス中心の処理が高性能計算のアプリケーションとして増加傾向にあり，これらの処理の高性能化が求められている，(3) プロセッサあるいはシステムの大規模化・高集積化で電力コストと廃熱の問題から電力削減が課題となっている．特に DRAM メモリは電力消費が大きいコンポーネントである，といった背景が存在する．

一方，高性能計算における主たるデータ表現形式は浮動小数点表現であるが，ほとんどのプログラムは IEEE 754-2008 規格 [1] に基づく 32/64 ビットの浮動小数点型（binary32/binary64，いわゆる単精度/倍精度，あるいは FP32/FP64）を用いて実装されている．IEEE 754-2008 は 1985 年に制定された IEEE 754 を改訂したものであり，FP32/FP64 は IEEE 754 当時からの規格であるから，実に誕生から 30 年近くが経過している．IEEE 754-2008 においてはこのほかに 16 ビット浮動小数点型（binary16，FP16，いわゆる半精度）と 128 ビット浮動小数点型（binary128，FP128，いわゆる 4 倍精度）が定義されているが，x86 をはじめとする汎用プロセッサにおいてハードウェアにおけるサポートはされておらず，現時点で広く普及しているとは言いがたい．

このように，一般的な計算機のプログラムは FP32 か FP64 の 2 段階の精度で記述されていることがほとんどである．しかしこれらの 2 段階の精度が，ユーザーが必要とする精度の計算結果を得るために，そのプログラム中で真に必要となる精度と，常に完全に合致しているとは考えにくい．多くの計算では，プログラムは無駄な情報を含みながら処理を行っており，そのために無駄なデータアクセスや電力消費が発生していることが考えられる

そこで本研究ではこの問題を解消すべく，ユーザが必要とする桁数だけを無駄なく格納するために，IEEE 754-2008 の浮動小数点形式（以下，IEEE 形式と呼ぶ）に対して仮数部ビット長を多段階で短縮した新しい浮動小数点形式を提案し，その有効性を検討する．本稿ではこの新しい浮動小数点形式の総称を，IEEE 形式に対して長さを短縮した浮動小数点形式という意味で，短尺浮動小数点形式（Short Length Floating-Point formats: SLFP）と呼ぶ．我々が短尺浮動小数点形式の導入によって期待する具体的な効果を以下に示す：

(1) 無駄なメモリアクセス量を削減することにより，メモリアクセス性能に律速されていた処理で計算の高速化を実現する．また無駄なメモリアクセスにより発生する電力消費を削減する．
(2) MPI によるノード間，GPU・MIC 等のアクセラレータとホスト間で発生していた無駄な通信の削減により，これらに律速されていたプログラムでは計算の高

[1] 理化学研究所計算科学研究機構
[a)] daichi.mukunoki@riken.jp
[b)] imamura.toshiyuki@riken.jp

速化を実現し，また電力消費の削減を図る．
(3) 使用メモリ量の削減によりノードあるいはアクセラレータあたりの計算問題サイズを大きく取ることで，同じシステム上でより大きな計算を可能にする．もしくは同じ規模の計算をより少ない並列数で計算することにより高速化を達成する．

本稿では短尺浮動小数点形式の検討の第一段階として，まず現在の計算機上で短尺浮動小数点形式を実現する方法を検討し，上記 (1) に関して，CPU（x86）と GPU（NVIDIA）上に短尺浮動小数点形式を適用した簡単な計算カーネル（ベクトルの加算とスカラ倍，行列積）を実装して，演算性能と電力性能を評価する．そして理想的な短尺浮動小数点形式の実現に向けた課題を整理することを目的とする．

2. 関連研究

本節では本研究と同様に浮動小数点データ量の削減に着目した研究や技術動向を紹介する．

FP16（半精度）：FP16 は IEEE 754-2008[1] において定義されている格納専用の浮動小数点形式である．GPU においては以前から画像処理を主目的とした格納専用形式として一部サポートされていたが，2015 年に NVIDIA 社の Maxwell アーキテクチャに基づくエンベデッド向け製品 Tegra X1 において初めて FP16 演算器が搭載され [2]，その後プログラミング言語（CUDA）におけるサポート強化，および次世代 Pascal アーキテクチャにおいて FP16 演算器の搭載を表明している [3]．この背景には昨今 GPU の主要ターゲットアプリケーションの一つと目される機械学習の一種である深層学習（deep learning）に活用可能である [4] ことや，車載コンピュータ等における画像認識処理需要の高まりなどが挙げられる．CUDA 7.5 に添付されている CUBLAS[5] においては，FP16 で演算を行う cublasHGEMM ルーチンと，FP32 の単精度行列積ルーチン（SGEMM）において FP16 フォーマットでの入出力に対応した cublasSgemmEx ルーチンをサポートしている．

混合精度演算：近似計算等のあまり精度を必要としない箇所で，他の部分の計算よりも低い精度で計算を行う混合精度演算（[6] など）が提案されている．多くの計算機環境は FP64 と比べて FP32 の浮動小数点演算性能が高速であるため，演算律速な計算では FP32 演算器の活用による高速化が，データ律速な計算においてもデータアクセス量が削減されることによる速度向上が期待できる．また，電力性能を含めた議論も行われている [7]．混合精度演算の研究は FP64 の計算において部分的に FP32 を用いた事例が多いが，多倍長精度演算に適用した研究 [8] も行われている．一方で，入出力のデータ型と演算に用いるデータ型が異なる混合精度 BLAS ルーチンの実装も行われている（XBLAS[9] や，前述の CUBLAS における cublasSgemmEx）．また，Albert ら [10] は，FP64/FP32/FP16 を用いた混合精度演算に適したプロセッサ設計を行い，データサイズの削減による演算速度と電力性能を向上を試みている．

浮動小数点圧縮：圧縮により浮動小数点データ量の削減を試みた研究が行われている（[11][12] など）．我々の知る限りでは多くの研究は可逆圧縮である．一方で多階層精度圧縮数値記録 JHPCN-DF[13] は可逆圧縮と非可逆圧縮の両方の側面を持つと同時に，本研究が目指しているような任意長の多段階精度を実現できる．JHPCN-DF の原理は IEEE 形式の浮動小数点型の仮数部を任意の箇所で分割し，それぞれの不要部分をゼロ埋めしてから既存のハフマン圧縮を適用するというものである．分割したデータをそれぞれ保持すればあとで復元可能であるという点では可逆圧縮ととらえることもできるが，分割した上位ビットだけを用いる場合には非可逆圧縮の一種ととらえることもできる．しかしこのように何らかの可逆圧縮の使用は変換コストが大きい．JHPCN-DF はメインターゲットを可視化データの圧縮としているほか，多倍長精度演算の通信部分に適用した事例がある．

その他：多倍長精度計算においては，ユーザが必要とする任意の精度で計算を行う手段として任意長浮動小数点演算が実現されており，例えば ARPREC[14] や MPFR[15] などの実装が存在する．しかし既存手法はいずれも FP64 を超える高精度に対するものであり，それ以下の精度を実現していない．一方，本稿の著者の一人である椋木と高橋は，Double-Double 型の 4 倍精度演算 [16] において，より小さなデータ型として 3 倍精度型 [17] を実現する方法を提案した．本稿で提案する短尺浮動小数点形式はこの手法をベースとしている．また，指数部を可変長とした新しい浮動小数点表現の提案 [18] を行った論文も存在するが，実装方法や性能に関する議論は行われていない．一方で，IEEE 形式の FP64/FP32/FP16 は順に指数部長が短く，数の表現範囲が狭くなるため，これらを用いた混合精度演算や，計算全体を低精度フォーマットに移行する際には，問題が生じる可能性があった．

3. 短尺浮動小数点形式の提案と実装

本節では短尺浮動小数点形式のソフトウェアによる実現方法を提案し，C 言語による CPU と GPU（CUDA）での実装について述べる．

短尺浮動小数点形式の理想としては，1 ビット単位で任意の仮数部長・指数部長をユーザが設定可能な格納形式と，それに対応する算術演算が実現され，ビットが短いほど処理速度が高速で消費電力も小さくなることが望ましい．しかしまず，プロセッサが持つ FP32/FP64 演算器より高速な任意精度の算術演算をソフトウェア的に実装することは不可能であるから，今回は算術演算には既存の FP32/FP64 演算器を使用することを前提とする．つまり何らかの計算を行う際には，メモリ上にある短尺浮動小数点形式のデー

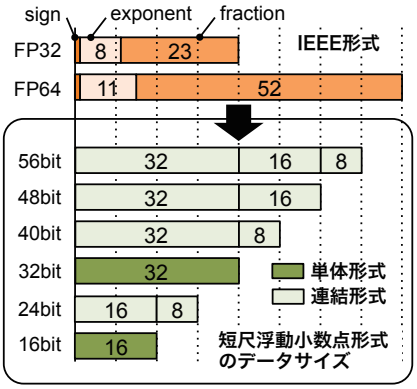

図 1 短尺浮動小数点形式の概要

タをレジスタ上で FP32/FP64 に変換し，FP32/FP64 演算器で計算を行い，計算結果を短尺浮動小数点形式に変換してメモリ上に書き込む，という方式を採る．この方式は椋木・高橋による 3 倍精度演算 [17] と同様の方法である．また，計算カーネル単位で実装すると，CUBLAS の cublasSgemmEx や XBLAS のルーチンのような実装となり，一種の混合精度演算と言うこともできる．

3.1 格納形式

格納方法を検討した結果，FP32/FP64 の仮数部を 8 ビット単位で短縮（切り捨て）し，これを 16–64 ビットで 8 ビット刻みのワードサイズを持つ “入れ物”に格納するという方法を考えた（**図 1**）．8 ビット単位の入れ物は，8/16/32 ビットの整数型（C 言語の stdint.h における uint8_t, uint16_t, uint32_t）の組み合わせで実現できるため，計算機上での表現としては合理的である．**表 1** にこの方法で実現できる FP64 および FP32 ベースの短尺浮動小数点形式を示す．本稿では仮にこれらの形式を，SLFP64in48b（FP64 を 48 ビットの入れ物に格納）というように命名した．この方法によって新たに 8 種類のフォーマットが生まれ，IEEE 形式の FP16/32/64 を含めると 11 段階の精度を表現できる．

この方法では指数部長の変更や仮数部長の 1 ビット単位の指定はできないが，FP32/FP64 の符号部＋指数部＋仮数部の形式をそのまま保持するため，後述する変換関数の実装が容易である．なお，16/32/64 ビットは 1 ワードで実現できるため，本稿ではこれを単体形式と呼ぶ．一方で 24/40/48 ビットは 2 ワードの組み合わせ，56 ビットは 3 ワードの組み合わせによる表現となり，本稿ではこれらを連結形式と呼ぶことにする．

この方法は FP64 の組み合わせで 4 倍/8 倍精度を表現する DD/QD 演算（QD[16] など）への適用も可能であり，そうすれば 64 ビット以上の精度も多段階化できる．DD 演算における 4 倍精度表現を FP64+32 ビット整数型に格納したものが，椋木・高橋による D+I 型の 3 倍精度型 [17] である．

3.2 IEEE 形式との変換

提案手法の短尺浮動小数点形式は IEEE 形式の符号部＋指数部＋仮数部の形式をそのまま保持しているから，IEEE 形式との変換はこれらのビット列を論理シフトで操作することで容易に実現できる．**図 2** に，x86 CPU における短尺浮動小数点形式（SLFP64in48b）と IEEE 形式（FP64）の変換関数の実装例を示す．C 言語の共用体機能と，論理シフト演算を活用している．なお変換関数は関数呼び出しのオーバーヘッドを排除するためにインライン関数として実装する．GPU（CUDA）の場合には，これらの関数をデバイス関数として実装し，__forceinline__修飾子でインライン化する．

3.3 配列の格納方式の検討

連結形式による短尺浮動小数点形式は 1 要素のみの場合，構造体として表現すれば良い．しかしその構造体を配列として形成すると，メモリアラインメント条件を満たせなくなり，メモリアクセス性能が大幅に低下する恐れがある．そこで連結形式を配列として格納する場合には，構造体の配列（Array of Structures：AoS）形式ではなく，配列の構造体（Structure of Arrays：SoA）形式を用いる．SoA では構造体の中に連結形式を構成する要素それぞれの配列のポインタを格納する．**図 3** に SoA 形式による短尺浮動小数点形式（SLFP64in48b）と IEEE 形式（FP64）の変換関数の実装例を示す．なお，単体形式では SoA 形式の格納というものは存在しないが，今回は便宜上，SoA 形式と同様の方法でプログラムを記述した．例えば SLFP64in32b では，uint32_t 型のポインタ 1 つからなる slfp64i32bArray 構造体で管理する．

4. 短尺浮動小数点形式を用いた計算カーネルの実装

本稿では CPU および GPU において，短尺浮動小数点形式を適用した Level-1 BLAS の AXPY（$y = \alpha x + y$）と，Level-3 BLAS の GEMM（$C = \alpha AB + \beta C$）を実装し，演算性能と電力性能を評価する．本節ではその実装を示す．

今回は単純なコード変換で短尺浮動小数点形式を適用した場合の性能を評価するという意味も込めて，通常の倍精度（FP64）の実装をベースに，メモリアクセスを行う箇所に短尺浮動小数点形式と IEEE 形式の変換関数を，コンパイラのプリプロセッサで機械的に適用して，各形式の実装を生成した．また，型の違いごとに異なるチューニングは行わず，スレッド数やブロックサイズなどのパラメータは FP64 実装において検討したものと同一の値を用いた．

なお，演算を FP64 で行うカーネルについては，入出力データに FP32 型を用いて，通常の FP64 と FP32 のキャスト（型変換）による変換を用いた実装も行った．本稿ではこれを FP64inFP32 と呼ぶ．

表 1 IEEE 形式と短尺浮動小数点形式（10 進桁数は正規化数に基づいて算出した桁数を小数点以下第 3 桁で四捨五入した値）

名称	ワードサイズ	指数部長	仮数部長	10 進桁数	備考
FP64	64 bits	11 bits	52 bits	15.95	IEEE 形式
SLFP64in56b	56 bits (32+16+8)	11 bits	44 bits	13.55	
SLFP64in48b	48 bits (32+16)	11 bits	36 bits	11.14	
SLFP64in40b	40 bits (32+8)	11 bits	28 bits	8.73	
SLFP64in32b	32 bits	11 bits	20 bits	6.32	
SLFP64in24b	24 bits (16+8)	11 bits	12 bits	3.91	
SLFP64in16b	16 bits	11 bits	4 bits	1.51	
FP32	32 bits	8 bits	23 bits	7.22	IEEE 形式
SLFP32in24b	24 bits (16+8)	8 bits	15 bits	4.82	
SLFP32in16b	16 bits	8 bits	7 bits	2.41	
FP16	16 bits	5 bits	10 bits	3.31	IEEE 形式

```
union union64 {
    uint64_t i;
    double f;
};

struct slfp64i48b {
    uint32_t i32;
    uint16_t i16;
};

__inline__ slfp64i48b fp64_to_slfp64i48b_rz
(double f64) {
    slfp64i48b sa;
    union union64 u64;
    u64.f = f64;
    sa.i32 = (uint32_t)(u64.i >> 32);
    sa.i16 = (uint16_t)(u64.i >> 16);
    return sa;
}

__inline__ double slfp64i48b_to_fp64
(slfp64i48b sa) {
    union union64 u64;
    uint64_t i64h, i64l;
    i64h = (uint64_t)sa.i32[i];
    i64h = i64h << 32;
    i64l = (uint64_t)sa.i16[i];
    i64l = i64l << 16;
    u64.i = i64h | i64l;
    return u64;
}
```

図 2 AoS 形式による短尺浮動小数点形式（SLFP64in48b）と IEEE 形式（FP64）の変換関数

```
struct slfp64i48bArray {
    uint32_t *i32;
    uint16_t *i16;
};

__inline__ void fp64_to_slfp64i48bArray_rz
(double f64, slfp64i48bArray sa, size_t i) {
    union union64 u64;
    u64.f = f64;
    sa.i32[i] = (uint32_t)(u64.i >> 32);
    sa.i16[i] = (uint16_t)(u64.i >> 16);
}

__inline__ double slfp64i48bArray_to_fp64
(slfp64i48bArray sa, size_t i) {
    union union64 u64;
    uint64_t i64h, i64l;
    i64h = (uint64_t)sa.i32[i];
    i64h = i64h << 32;
    i64l = (uint64_t)sa.i16[i];
    i64l = i64l << 16;
    u64.i = i64h | i64l;
    return u64.f;
}
```

図 3 SoA 形式による短尺浮動小数点形式（SLFP64in48b）と IEEE 形式（FP64）の変換関数

4.1 AXPY

AXPY はベクトル長 N の計算において，$3N$ 要素のメモリアクセスに対して $2N$Flops の演算が生じるメモリインテンシブな処理である．さらにメモリアクセスは単純な連続アクセスであるため，多くの環境ではシステムのメモリ帯域を使い切る処理である．したがって，短尺浮動小数点形式を適用した場合には，メモリアクセス量の削減による演算性能，電力性能の向上が期待できる．

CPU における SLFP64in48b（FP64 ベース 48 ビット短尺浮動小数点形式）の AXPY の実装例を**図 4** に示す．このコードにおいて，FP_TYPE/FP_TYPE_ARRAY はレジスタ上で用いられる IEEE 形式，SL_TYPE/SL_TYPE_ARRAY はメモリ上のデータ表現に用いられる短尺浮動小数点形式を表しており，短尺浮動小数点形式はプリプロセッサで機械的に適用される．なお，for 文に対しては OpenMP による並列化を指示している．

GPU 版の実装も CPU 版と同様にマクロ展開によって短尺浮動小数点形式を適用する．CUDA 化するにあたり CPU 版の実装において for 文を回しているインデックス (i) をスレッド ID に置き換えた．また，スレッドブロックあたりのスレッド数は 128 とした．

```
#define FP_TYPE double
#define SL_TYPE slfp64i48b
#define TO_SL fp64_to_slfp64i48b_rz
#define TO_FP slfp64i48b_to_fp64
#define SL_TYPE_ARRAY slfp64i48bArray
#define TO_SL_ARRAY fp64_to_slfp64i48bArray_rz
#define TO_FP_ARRAY slfp64i48bArray_to_fp64
#define SLAXPY slfpAxpyFp64i48b

int32_t SLAXPY (size_t n, SL_TYPE a,
SL_TYPE_ARRAY x, SL_TYPE_ARRAY y) {
    size_t i;
    register FP_TYPE ra, rx, ry;
    #pragma omp parallel for private (ra, rx, ry)
    for (i = 0; i < n; i++) {
        ra = TO_FP (a);
        rx = TO_FP_ARRAY (x, i);
        ry = TO_FP_ARRAY (y, i);
        ry = ra * rx + ry;
        TO_SL_ARRAY (ry, y, i);
    }
    return 0;
}
```

図 4 CPU における SLFP-AXPY の実装（SLFP64in48b の場合）

4.2 GEMM

GEMM は行列積 $C = \alpha AB + \beta C$ を計算する Level-3 BLAS ルーチンである．$N \times N$ 要素からなる正方行列の場合，$4N^2$ 要素のメモリアクセスに対して $2N^3 + 3N^2$Flops の演算を行う演算インテンシブな処理である．メモリアクセス時間が全体の実行時間に占める割合はごくわずかであるから，短尺浮動小数点形式を適用しても演算性能の大幅な向上は期待できないが，メモリアクセスに要する電力次第では電力性能が改善される可能性がある．GEMM は適切な最適化を施すとプロセッサの理論ピーク演算性能に近い性能が得られることが知られているが，本稿では GEMM の実装そのものが目的ではないため，最低限の最適化のみを行った．短尺浮動小数点形式の適用方法は AXPY の場合と同様である．

CPU 版の実装では，内積形式の一般的な 3 重ループによる実装をベースに，ルーチン内において行列 A の転置によるメモリアクセス方向の最適化，ブロッキング（ブロックサイズ=256），ブロック内の i,j ループに対してそれぞれ 4 段のループアンローリングを適用した．ブロックサイズに対して問題サイズが端数となる場合の処理は省略した．また，それぞれの最外側ループにおいて OpenMP による並列化を行った．

GPU 版の実装は著者らが Maxwell アーキテクチャ向けに行った過去の実装 [19] をベースに，スレッドブロックサイズを 16×16，共有メモリブロッキングを 128×16，レジスタブロッキングを 8×8 とした実装を用いた．キャッシュモードは cudaFuncCachePreferShared として，Texture メモリは使用していないが，行列 A, B の読み込みには組込関数__ldg() を用いて Read Only Data Cache を適用した．また展開可能な for 文は#pragma unroll により自動展開を指示している．またスレッド数およびブロックサイズに対して問題サイズが端数となる場合の処理は省略している．

表 2 実験環境（Flops, GB/s 等の数値はカタログスペックである）

CPU	Intel Core i7-4790
	(4 cores, 3.6GHz)
Flops (FP32/FP64)	460.8 / 230.4 GFlops
Host Memory	DDR3 1600MHz 16 GB
Host Memory Bandwidth	25.6 GB/s
GPU	NVIDIA Tesla K20c
Flops (FP32/FP64)	3.52 TFlops / 1.17 TFlops
Device Memory	GDDR5 5 GB (ECC Enabled)
Device Memory Bandwidth	208 GB/s
OS	CentOS 7.1.1503
	(3.10.0-229.4.2.el7.x86_64)
CUDA	7.5
GPU Driver	352.39
Compiler	gcc 4.8.3, nvcc V7.5.17

5. 評価実験

CPU と GPU において AXPY と GEMM の演算性能と電力性能を評価した．評価方法と実験結果について述べる．

5.1 評価環境と設定

実験に用いた計算機の概要を**表 2** に示す．CPU の Intel Core i7-4790 は Haswell アーキテクチャの 4 コア CPU であり，Hyper Threading は無効としている．GPU の NVIDIA Tesla K20c は Kepler アーキテクチャ（Compute Capability 3.5）であり，CUDA Toolkit に含まれる nvidia-smi コマンドにより Persistent mode を有効にして，GPU Boost 機能の最大クロックをメモリ：2600MHz，コア：758MHz に設定している．

CPU・GPU 向けプログラムはともにコンパイルの最適化オプションとして O3 を設定した．GPU 向けプログラムはコンパイルオプションで Compute Capability 3.5 向けのコード生成を行った．CPU 向けプログラムの実行前には OpenMP のスレッド数（OMP_NUM_THREADS）をコア数と同じ 4 に設定した．

なお，GPU における測定では，CPU-GPU 間のデータ転送における時間・電力は測定対象としていない．GEMM の測定においては行列は正方行列であるとし，入力データはスカラ変数 α，β も含めてすべて乱数で初期化している．また参考データとして，CPU においては OpenBLAS-0.2.15[20]，GPU においては CUBLAS 7.5[5] の性能も測定した．

5.2 問題サイズに対する演算性能

問題サイズに対する演算性能の測定では，ルーチンを最

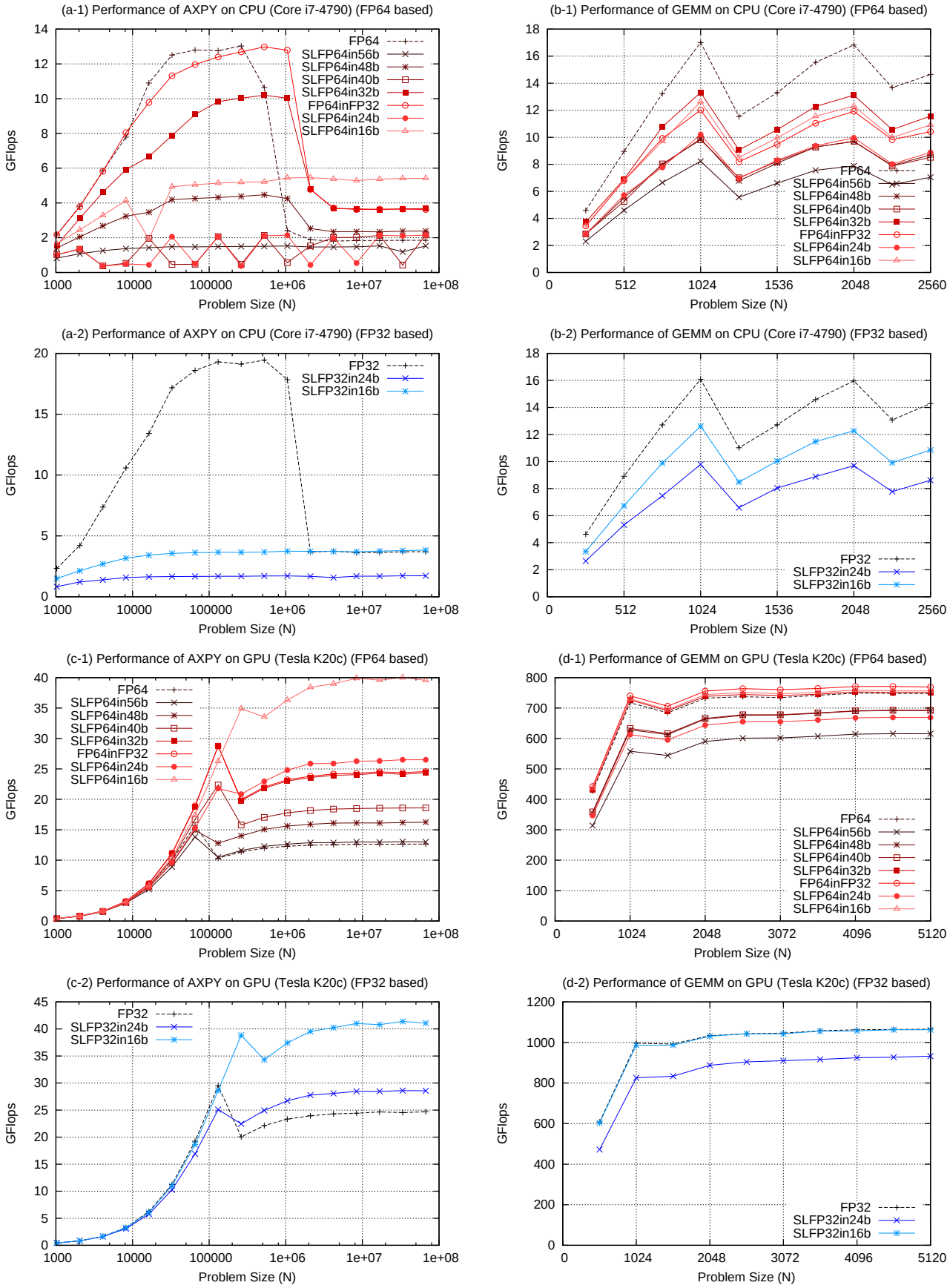

図 5 問題サイズに対する演算性能

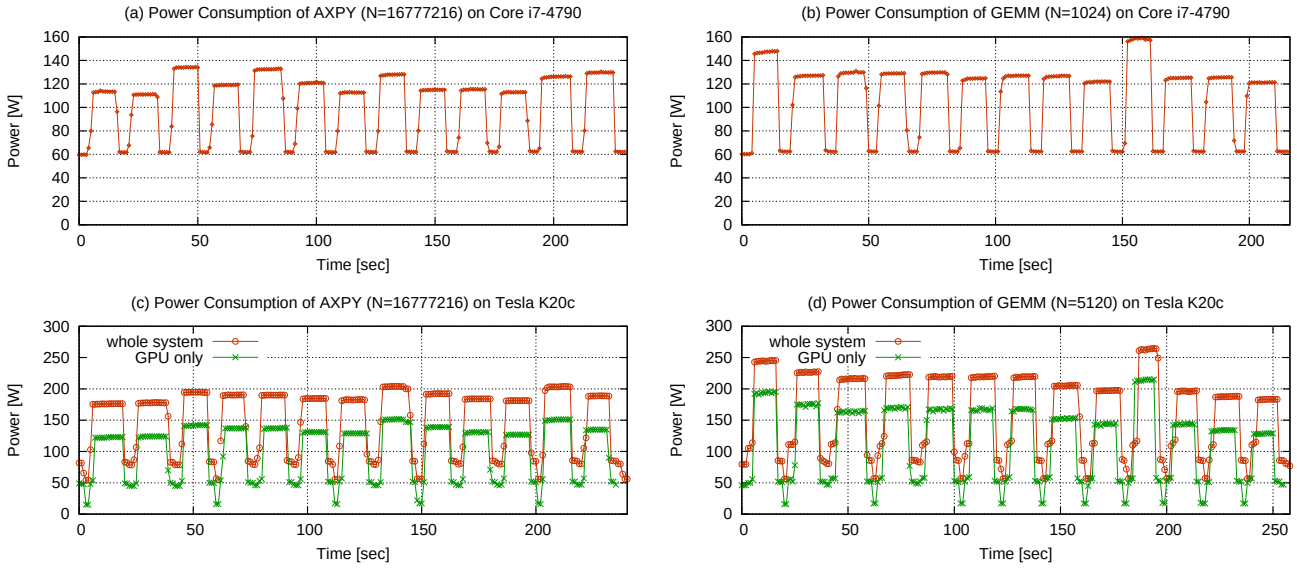

図 6 電力測定中の時間方向に対する電力変化

表 3 電力性能（GPU の結果では nvidia-smi で取得した GPU ボード単体の電力および電力性能を括弧内に示す）

(a) AXPY（N=16777216, CPU: Core i7-4790）

Type	GFlops	W	GFlops/W
FP64-OpenBLAS	1.84	114	16.1
FP64	1.84	111	16.5
SLFP64in56b	1.53	134	11.4
SLFP64in48b	2.39	119	20.0
SLFP64in40b	2.16	133	16.2
FP64inFP32	3.61	122	29.7
SLFP64in32b	3.64	113	32.2
SLFP64in24b	2.15	128	16.7
SLFP64in16b	5.40	115	46.9
FP32-OpenBLAS	3.65	116	31.6
FP32	3.66	113	32.4
SLFP32in24b	1.86	127	14.7
SLFP32in16b	4.19	130	32.1

(b) GEMM（N=1024, CPU: Core i7-4790）

Type	GFlops	W	MFlops/W
FP64-OpenBLAS	159.9	148	1080.4
FP64	17.1	128	134.3
SLFP64in56b	8.7	131	66.5
SLFP64in48b	10.8	129	83.8
SLFP64in40b	10.9	130	83.9
FP64inFP32	14.2	125	114.0
SLFP64in32b	13.1	127	102.6
SLFP64in24b	11.1	127	87.5
SLFP64in16b	13.3	122	108.8
FP32-OpenBLAS	328.7	159	2064.7
FP32	17.7	125	140.9
SLFP32in24b	10.6	126	84.5
SLFP32in16b	13.2	122	108.3

(c) AXPY（N=16777216, GPU: Tesla K20c）

Type	GFlops	W	GFlops/W
FP64-CUBLAS	12.6	176 (123)	71.4 (102.4)
FP64	12.6	178 (124)	70.8 (101.8)
SLFP64in56b	13.0	195 (142)	66.5 (91.3)
SLFP64in48b	16.1	191 (138)	84.6 (116.7)
SLFP64in40b	18.6	190 (138)	97.5 (134.4)
FP64inFP32	24.3	185 (131)	131.6 (185.6)
SLFP64in32b	24.5	183 (129)	134.0 (190.1)
SLFP64in24b	26.3	204 (152)	128.9 (173.1)
SLFP64in16b	39.6	193 (139)	205.4 (284.8)
FP32-CUBLAS	24.0	184 (131)	130.2 (183.1)
FP32	24.7	181 (127)	135.9 (194.1)
SLFP32in24b	28.5	204 (151)	139.6 (188.5)
SLFP32in16b	40.8	189 (135)	215.8 (302.3)

(d) GEMM（N=5120, GPU: Tesla K20c）

Type	GFlops	W	GFlops/W
FP64-CUBLAS	1108	246 (195)	4.51 (5.68)
FP64	748	228 (177)	3.29 (4.23)
SLFP64in56b	616	217 (165)	2.84 (3.73)
SLFP64in48b	692	223 (171)	3.10 (4.05)
SLFP64in40b	693	220 (169)	3.15 (4.10)
FP64inFP32	751	221 (169)	3.41 (4.45)
SLFP64in32b	769	220 (168)	3.50 (4.58)
SLFP64in24b	669	206 (153)	3.25 (4.37)
SLFP64in16b	757	198 (145)	3.82 (5.22)
FP32-CUBLAS	2621	265 (215)	9.90 (12.19)
FP32	1066	197 (144)	5.42 (7.40)
SLFP32in24b	933	188 (134)	4.95 (6.96)
SLFP32in16b	1063	184 (129)	5.79 (8.24)

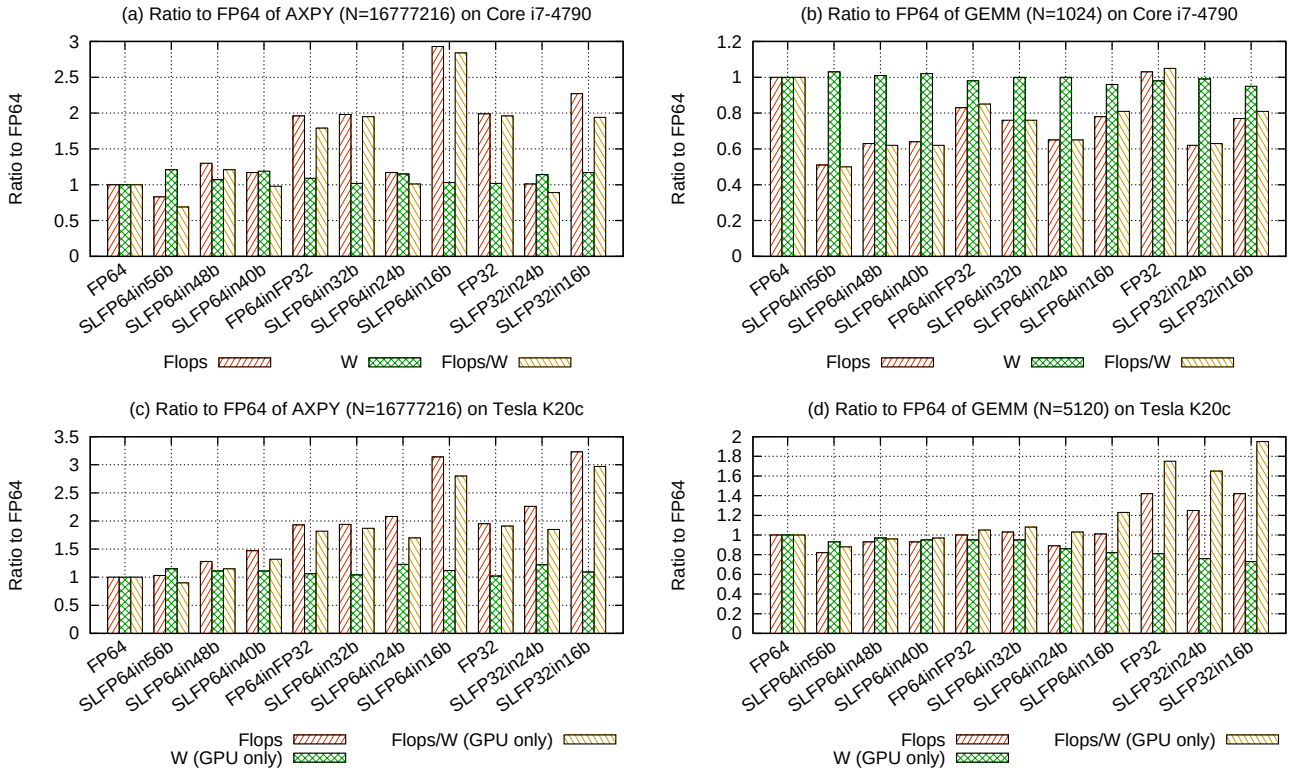

図 7 FP64 を基準に正規化した演算性能，電力，および電力性能

低で 10 回以上かつ全体の実行時間が 1 秒を超えるような回数で繰り返し実行し，その全体の実行時間をタイマー（gettimeofday）で測定して，繰り返し回数で割った平均実行時間を元に Flops 値を算出した．結果を**図 5** に示す．それぞれ上側に FP64 演算器を用いるルーチン，下側に FP32 演算器を用いるルーチンの結果を掲載した．

5.3 電力性能

電力測定にはいずれも三和電気計器の製品で，デジタルマルチメータ PC720M，クランプ電流計 CL-22AD，ラインセパレータ LS11 を使用した．測定対象の計算機をラインセパレータ経由で家庭用の 100V AC 電源に接続し，ラインセパレータは感度倍率を 10 倍にして測定した．今回の測定環境では電圧と電流を同時に測定することができないため，電圧は 100V で一定であると仮定している．GPU ボード単体の電力は nvidia-smi を使用して取得した．GPU ではある程度の時間アイドル状態であった後に初めて GPU カーネルを実行した場合に，ホスト側の消費電力が上昇する傾向が確認されたため，この影響を避けるためにダミー実行を行ってから測定を行っている．

電力性能の測定ではある問題サイズでルーチンを繰り返し実行し，平均実行時間から算出した演算性能 [Flops] と，そのとき計測された最大電力 [W] から，電力性能 [Flops/W] を算出した．問題サイズはキャッシュから十分にはずれる大きさで，ルーチンの性能が十分に発揮できる適当な問題サイズ（AXPY は CPU・GPU ともに N=16777216，GEMM は CPU が N=1024，GPU が N=5120）を用いた．ルーチンの繰り返し実行回数は最低で 10 回以上かつ全体の実行時間が 10 秒を超えるように設定した．電力は繰り返し実行中に 1 秒間隔で測定したが，得られた値はほぼ一定であったため，その中の最大値を採用した．

表 3 に結果を示す．この測定では表 3 に示す実装を上から順に 5 秒間隔で実行しており，**図 6** に測定中の電力の履歴を示す [*1]．

5.4 結果から分かること

結果の理解を助けるため，**図 7** に，表 3 の結果を FP64 を基準として正規化した図を載せる．なお，OpenBLAS と CUBLAS の結果は除き，GPU プログラムの結果についてはシステム全体と GPU 単体の電力消費はほぼ同じ傾向であると判断し，GPU 単体の結果のみを掲載した．

まず，AXPY はメモリインテンシブな計算であるため，データ型の大きさが小さいほど高い演算性能・電力性能が得られることを期待した．GPU では実際にほぼ期待通りの演算性能・電力性能を示したが，SLFP64in56b や SLFP64in24b などの連結形式による短尺浮動小数点形式

[*1] GPU プログラムの電力測定はシステム全体と GPU ボード単体の測定を同時に行っているが，それぞれの計測システムの時刻は同期していないため，この図は個々のデータを後から重ねてプロットしている．しかしなんらかの理由により時間方向に若干のズレが生じている．

は，単体形式によるものと比べると消費電力が微増しており，短縮されたデータサイズの割に電力性能は良いとはいえない．連結形式では，連結形式を構成するワード数に比例してメモリアクセスと変換のための必要命令数が増加することが原因であると考えられる．一方 CPU でも，単体形式では GPU と同等の演算性能と電力性能の改善が得られたが，連結形式では基準となる FP64 と同等か，悪化しているものがある．また図 5 の CPU における AXPY の結果を見ると，FP64 などで問題サイズが小さいところではキャッシュの効果により演算性能が向上しているが，連結形式はいずれもキャッシュの恩恵が得られていない．また，SLFP64in40b と SLFP64in24b では問題サイズによってランダムに性能低下が起きている．これらの結果は連結形式による命令増加だけでなく，SoA のメモリレイアウトそのものに原因がある可能性がある．そのほか興味深い結果として，SLFP64in32b（短尺浮動小数点形式）と FP64inFP32（キャストによる一般的な混合精度）はデータサイズが同一で性能はほぼ同じであるにも関わらず，CPU・GPU ともに電力性能は前者の方が良いことがわかる．

一方，GEMM は演算インテンシブであるため，演算性能はデータ型の大きさとは無関係にほぼ同一となり，短尺浮動小数点形式の導入による演算性能・電力性能の改善効果は小さいと予想したが，GPU では単体形式の場合に演算性能が FP64 と同等かそれ以上でありながら，消費電力が低下し，その結果電力性能がわずかながら改善しているものが見られた．しかし CPU においてはベースとなる IEEE 形式と比べて，短尺浮動小数点形式あるいは FP64inFP32 を用いた実装は演算性能が大きく低下し，電力性能もその影響で低下していると考えられる．この結果については，今回の実装が FP64 の実装をベースに他形式の実装を機械的な変換で生成したため，実装によって最適化が不十分であった可能性がある．

6. まとめと今後の課題

本稿では計算速度の向上と省電力化を目的として，仮数部ビット長を 8 ビット単位で短縮した短尺浮動小数点形式の提案を行った．予備評価として，CPU と GPU において簡単な計算カーネル（AXPY と GEMM）を実装し，演算性能と電力性能を評価した．

短尺浮動小数点形式の適用によりほぼ期待通りに演算性能と電力性能が改善されたのは GPU 上の AXPY のみであり，他の実装ではなんらかの問題が生じた．まず短尺浮動小数点形式を FP64 のコードに対して機械的に適用するのみでは，演算性能が大幅に低下する場合があり，短尺浮動小数点形式を適用したカーネルの実装最適化手法を検討する必要がある．また，電力性能の検討はその次のステップであると言える．さらに，連結形式では単体形式と比べて期待した性能が得られなかったものが多く，特に GPU と比べて CPU において，性能上の問題が見られた．

今後は短尺浮動小数点形式を用いた計算カーネルの最適化手法を検討するとともに，より複雑なメモリアクセスが生じる疎行列計算カーネルにおける評価や，MPI 通信やアクセラレータにおけるホスト-デバイス通信における評価を行いたいと考えている．また今回の実装では IEEE 形式から短尺浮動小数点形式への変換を単純な切り捨て処理としたが，IEEE 標準の最近接偶数丸めを実装することが望ましく，実装方法と性能を検討する必要がある．一方で，仮に短尺浮動小数点形式によって実行時間の削減あるいは電力の削減が可能であったとしても，その有効性が主張できるのは，あくまで既存のプログラムが精度過多であるケースのみである．したがって，実際のアプリケーションにおいて精度過多が生じているケースを探し，そこに短尺浮動小数点形式を適用して，実行時間・消費電力の削減あるいは省メモリ化による利点を示す必要がある．現時点で我々が適用を検討しているのは，混合精度前処理付き反復解法や混合精度反復改良法における低精度演算部分である．このほか数値計算以外の，FP16 や JHPCN-DF がターゲットとしているような分野のアプリケーションについても，応用の可能性を検討したいと考えている．

我々は短尺浮動小数点形式を利用するためのソフトウェアライブラリ（SLFP）を開発中であり，著者らのウェブサイト（http://www.aics.riken.jp/labs/lpnctrt/index.html）において，本稿の実験に使用したソースコードを含む SLFP 0.0.1 を公開中である．

謝辞 本研究は公益財団法人計算科学振興財団 研究教育拠点（COE）形成推進事業の助成を受けたものである．

参考文献

[1] IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2008*, pp. 1–70 (2008).
[2] NVIDIA Corporation: Whitepaper NVIDIA Tegra X1 NVIDIA'S New Mobile Superchip, V1.0, http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf (2015).
[3] Hisa Ando: マイナビニュース GTC Japan 2015 - NVIDIA の Maxwell アーキテクチャと CUDA7.5, http://news.mynavi.jp/articles/2015/09/25/gtc_japan_2015_maxwell/ (2015).
[4] Gupta, S., Agrawal, A., Gopalakrishnan, K. and Narayanan, P.: Deep Learning with Limited Numerical Precision, *CoRR*, Vol. abs/1502.02551 (online), available from ⟨http://arxiv.org/abs/1502.02551⟩ (2015).
[5] NVIDIA Corporation: The NVIDIA CUDA Basic Linear Algebra Subroutines, https://developer.nvidia.com/cublas.
[6] Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A. and Dongarra, J.: Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems), *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 113 (2006).
[7] Anzt, H., Heuveline, V., Rocker, B., Castillo, M., Fer-

nandez, J., Mayo, R. and Quintana-Orti, E.: Power Consumption of Mixed Precision in the Iterative Solution of Sparse Linear Systems, *Proceedings of 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pp. 829–836 (2011).

[8] 幸谷智紀：倍精度と多倍長精度浮動小数点数を用いた反復改良法による連立一次方程式の高精度高速解法について，日本応用数理学会論文誌，Vol. 19, No. 3, pp. 313–328 (2009).

[9] Li, X. S., Demmel, J. W., Bailey, D. H., Hida, Y., Iskandar, J., Kapur, A., Martin, M. C., Thompson, B., Tung, T. and Yoo, D. J.: XBLAS – Extra Precise Basic Linear Algebra Subroutines, http://www.netlib.org/xblas/.

[10] Ou, A., Nguyen, Q., Lee, Y. and Asanovic, K.: A Case for MVPs: Mixed-Precision Vector Processors, *2nd International Workshop on Parallelism in Mobile Platforms (PRISM-2)* (2014).

[11] O'Neil, M. A. and Burtscher, M.: Floating-point Data Compression at 75 Gb/s on a GPU, *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*, pp. 7:1–7:7 (2011).

[12] Lindstrom, P. and Isenburg, M.: Fast and Efficient Compression of Floating-Point Data, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 12, No. 5, pp. 1245–1250 (2006).

[13] Hagita, K., Omiya, M., Honda, T., Murotani, K., Takeda, T., Kato, T. and Ogino, M.: Study of Efficient Data Compression by JHPCN-DF, *Proceedings of Annual Meeting on Advanced Computing System and Infrastructure (ACSI) 2015* (2015).

[14] Bailey, D.: ARPREC (C++/Fortran-90 arbitrary precision package), http://crd.lbl.gov/ ~ dhbailey/mpdist/.

[15] Hanrot, G., Lefèvre, V., Pélissier, P., Théveny, P. and Zimmermann, P.: MPFR : GNU MPFR Library, http://www.mpfr.org/.

[16] Bailey, D. H.: QD (C++/Fortran-90 double-double and quad-double package), http://crd.lbl.gov/~dhbailey/mpdist/.

[17] 椋木大地，高橋大介：GPU における 3 倍・4 倍精度浮動小数点演算の実現と性能評価，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 6, No. 1, pp. 66–77 (2013).

[18] 須田礼仁，小柳義夫：新しい可変長指数部浮動小数点数表現形式の提案，情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 1997-HPC-066, No. 37, pp. 31–36 (1997).

[19] 椋木大地，今村俊幸：Maxwell アーキテクチャ GPU における疑似倍精度演算を用いた DGEMM の実装と評価，情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2014-HPC-147, No. 26, pp. 1–6 (2014).

[20] Xianyi, Z.: OpenBLAS, http://www.openblas.net.