Vol.2015-HPC-148 No.8
2015/3/2

# 整数演算による多倍長浮動小数点演算エミュレーションのGPUでの性能評価

中里 直人[1]

**概要**：様々なアプリケーションにおいて, 仮数部・指数部ともに倍精度より拡張された多倍長浮動小数点演算の必要性が高まっている. GPU システムは通常, 倍精度浮動小数点演算による数値計算・シミュレーションを高速化するアクセラレータとして利用されている. 一方で, 最新の GPU システムでは, ハッシュ演算など整数演算を多用する処理を高速化する超並列プロセッサとしても活用されている. 本論文では, 多倍長浮動小数点演算を整数演算によりエミュレーションするアルゴリズムを, OpenCL カーネルとして実装することで, CPU/GPU システムで性能評価をおこなった. 結果 GPU システムでは, 加算で約 2.3 GFLOPS, 乗算で約 1.8 GFLOPS, 除算で約 0.23 GFLOPS の性能を得た.

**キーワード**：多倍長精度浮動小数点演算, OpenCL, GPU, アクセラレータ

## 1. はじめに

数値計算や数値シミュレーションのための計算機システムが高速化, 並列化, 大規模化されるとともに, 様々なアプリケーションにおいて, 数値計算シミュレーションで標準的に利用されている, IEEE 754-2008 で規定されている倍精度 binary64 フォーマット (仮数部 $n_{\mathrm{man}} = 53$ ビット, 指数部 $n_{\mathrm{exp}} = 11$ ビット) について改めて検討をする必要性が高まりつつある. 一方で, 計算機システムの高度化・大規模化のため, 総消費電力をできるだけ削減する技術が求められている. 例えば, 必要に応じて携帯端末の描画プロセッサ等で利用されている半精度浮動小数点 binary16 フォーマット ($n_{\mathrm{man}} = 11$, $n_{\mathrm{exp}} = 5$) を活用するのはその一例である. 他方では, 倍精度フォーマットより $n_{\mathrm{man}}$ を拡張した四倍精度 binary128 フォーマット ($n_{\mathrm{man}} = 113$, $n_{\mathrm{exp}} = 15$) も IEEE 754-2008 で規定されている. さらには, 様々なアプリケーションにおいて binary128 フォーマットよりもさらに $n_{\mathrm{man}}$ を拡張した多倍長精度演算が必要な場合があり, これまでその手法について様々な研究がおこなわれてきた. 多倍長精度演算を実現する手法として, 今日主に利用されている手法は, (a) 浮動小数点演算 (FP 演算) による多倍長演算手法 (例えば QD Library[5]) と, (b) 整数演算による浮動小数点エミュレーション (例えば GNU MPFR[2]) の二種に分類される.

上記 (a) の手法は, FP 演算の丸め誤差を補償する手法 [1], [7] をベースにしており, 倍精度変数を 2 語利用する double-double(DD) 方式 ($n_{\mathrm{man}} = 105$, $n_{\mathrm{exp}} = 11$ に相当) は, 現在の様々な計算機において高速に実行できる [13], [14]. この手法は語数を増やすことで, 単純に拡張することができるが (quad-double(QD) 方式 [4]), 指数部部分については最上位の変数のみが実効的であり, 残りの変数の指数部部分は冗長である. また, 語数を 2 語より増やした場合には, 変数のソート処理が必要であり, 条件分岐失敗時のコストが高いプロセッサや, SIMD 動作をするメニーコアプロセッサでは, 必ずしも高速に実行できるとは限らない. この手法のもう一つの弱点は, それがプロセッサの FP 演算機構に依存するため, 仮数部のビット長は語単位で任意に拡張できるが, 指数部はプロセッサが採用する FP フォーマットに固定されてしまう. これは通常は $n_{\mathrm{exp}} = 11$ となる.

我々はこれまで, 素粒子物理学で高速な演算が必要とされるファインマン・ダイアグラムの直接計算を, 多倍長精度 FP 計算で高速化するために様々な手法を検討してきた. ファインマン・ダイアグラムの直接計算に二重指数型積分法を適用することで, それは多次元積分の評価に帰着される. これは計算の並列性が大きく, また DD 方式は演算密度が高いため, GPU のようなアクセラレータ型計算機で高性能である [12], [14]. 一方で, 実用的にこの手法を利用する際には, 倍精度 FP 演算をベースとした DD/QD 方式では, $n_{\mathrm{exp}}$ が 11 ビットに制限されることが問題となる. この欠点を解消するために, 上記の (b) の手法による $n_{\mathrm{exp}}$ を拡張した演算が必要である.

[1] 会津大学大学院コンピュータ理工学研究科

FP 演算における主たる処理は仮数部同士の演算であり, 上記 (b) の手法は, 整数演算により複数語からなる仮数部の演算を, 四則演算それぞれの場合ついて筆算と同様のアルゴリズムでおこなう (例えば”The Art of Computer Programming Volume 2”[8](TACP Vol.2) Section 4.3 参照). ただし, 乗算 [9] と除算 [6] について, 筆算と同様の基本アルゴリズムよりも演算数を削減することのできるアルゴリズムが提案されている. この FP 演算のエミュレーションによる多倍長演算手法では, 原理的には指数部, 仮数部ともに任意のビット長を利用することができる. よって, 上記ファインマン・ダイアグラムの直接計算の場合における, 指数部サイズの制限の問題の解決策となる.

本論文では, (b) の手法による FP 演算を C 言語により設計・実装し, それを OpenCL カーネルとして利用可能とすることで, OpenCL でプログラム可能なマルチコア・メニーコアプロセッサ, アクセラレータにおける性能評価について報告する. 特に, 最新の GPU に代表されるメニーコアアクセラレータは, 数値計算や数値シミュレーションのための FP 演算ユニット (FPU) だけでなく, 整数演算機 (Arithmetic Logic Unit, ALU) を内蔵しており, FPU での演算と共にアドレス計算等に利用されている. 実際, GPU に搭載された ALU は, 整数演算超並列プロセッサとしても活用することが可能であり, Bitcoin 採掘などハッシュ演算を高速におこなう必要のある用途で利用されている. (b) の手法に必要な基本演算は, 整数加算, 整数乗算, シフト処理と条件分岐などであり, 現代的な GPU の ALU であれば充分実装可能である. 実際, CUDA Multiple Precision Arithmetic Library(CUMP)[11] は, GNU Multiple Precision Arithmetic Library(GMP)[3] をベースとして, CUDA にて多倍長 FP 演算による加算と乗算を実装し, その性能評価をしている. CUMP では, 効率的に語数が多い多倍長 FP 演算を実装するために, C++テンプレートを利用した実装を採用しており, C++がサポートされていない現行の OpenCL で動作させることは難しい.

本論文では, CUMP とは独立に OpenCL で動作させることを目的として, C 言語にて多倍長 FP 演算を実装し, その実装と性能評価について報告する.

## 2. 実装詳細

本章では OpenCL による多倍長 FP 演算の実装 (以下, MYFP と呼ぶ) について説明する. MYFP を実装する際には, GPU に代表されるアクセラレータをターゲットとして想定した. OpenCL によるプログラミングが可能な主なアクセラレータは, NVIDIA 社および AMD 社の GPU があり, どちらも ALU は 32 ビットである. そのため, 本研究における OpenCL カーネルの実装では基本データ構造として符号なし 32 ビット整数 ( `uint32_t` ) を採用した. 図 1 に, 本研究における多倍長精度変数の定義 (構造体) をし

```
typedef uint32_t  u32;
const u32 NC = 7;
struct my_fp {
  u32 e;
  u32 m[NC];
};
typedef struct my_fp FP[1];
```

**図 1** 本論文における多倍長精度 FP フォーマットの定義

めす. この構造体 `FP` では, `FP->e` に指数部と符号部を保持する. `FP->e` の最上位の 1 ビットは符号部を保持し, 下位 30 ビットに指数部を保持する仕様とした. 指数部は IEEE 754 規格にならって, バイアスを考慮した符号なし 2 進数とした. $n_{\mathrm{exp}} = 30$ より, この場合のバイアスは 16 進数で 0x3fffffff となる. `FP->m[]` は仮数部を保持する. 仮数部も指数部と同様に符号なし 2 進数とし, 各 `m[0]`, `m[1]`... のそれぞれの語において, 下位から 30 ビットに仮数部を分割して保持する. `m[0]` の下位から 30 ビット目が常に 1 となるように正規化する. 本研究では, `FP->m[]` の語数を 7 と固定したので, $n_{\mathrm{man}} = 210$ である.

以下, 加減算, 乗算, 除算の実装アルゴリズムについて説明する. 加減算と乗算は標準的なアルゴリズムを利用しており, IEEE 754-2008 規格との違いは, 全ての丸め処理に, 演算処理が少ない force-1 丸めを採用していることである. force-1 丸め処理と標準的な nearest even による丸め処理を比べると, どちらも誤差が非等方となるバイアスはないが, force-1 丸め処理の誤差は標準丸め処理よりも 0.5 ビット悪い. また, 本研究での実装では, IEEE 754-2008 で定義されている非正規化数や, NaN などの特殊な数値に関する処理, 例外処理は考慮していない.

### 2.1 加減算

FP 加算のアルゴリズムは, TACP Vol.2[8] Section 4.2 に記載されているものと同様の標準的なアルゴリズムを実装している. [*1] 仮数部同士の演算は, 加減算どちらの場合にも, 符号に応じて引数の仮数部を 2 の補数表現に変換したうえで, 符号なし整数の加算として計算している. 複数語からなる仮数部同士の加算は, ループにより下位の語から加算し, 上位 2 ビットをキャリーとして次の語の加算時に加える方法である. 仮数部の演算結果が 2 の補数の場合には, 通常の整数値に変換するため追加のビット反転と加算処理が必要である. また, 計算結果を正規化するために左右のシフトが必要な場合がある. 全ての演算は 32 ビット符号なし整数の加算やシフト処理に帰着される. 以上のことから, 一回の加算に演算に必要な 32 ビット整数の回数は入力値に依存する. 減算は, 符号を反転させた変数を加算ルーチンの入力とすることで実装しており, 減算専用の

[*1] なお Section 4.2 は FP 演算処理について, Section 4.3 は多倍長整数演算処理についてである.

アルゴリズムは採用していない.

### 2.2 乗算

乗算のアルゴリズムも, TACP Vol.2[8] Section 4.2 に記載されているものと同様の標準的なアルゴリズムを実装している. 乗算では, まず仮数部同士を, 32 ビット符号なし整数とみなして乗算し, 部分積を 64 ビット符号なし整数値として一時変数へ保持する. この処理には $7x7 = 49$ 回の乗算処理となる. 49 個の部分積を下位から加算し, キャリー処理を考慮した上で処理し, 7 語の仮数部に格納し, 同時に正規化処理と丸めのためのビットを計算し force-1 丸めをおこなう. 乗算には, 32 ビット符号なし整数演算に加えて, 一部 64 ビット符号なし整数の演算が必要である. 今回の実装では, 語数が 7 語と比較的短いため, 乗算数を削減するアルゴリズムは検討していない. FP 乗算処理は, 加減算とは異なり仮数部の演算結果に応じた演算数の変化はない.

### 2.3 除算

除算は以下の 3 種類の手法を採用し, 性能の比較をおこなった.

#### 2.3.1 仮数部の直接除算

FP 除算のアルゴリズムは, 乗算と同様であり, 仮数部の取り扱いだけが異なる. TACP Vol.2[8] Section 4.3 には, 多倍長整数の同士の標準的なアルゴリズムが既述されている. Huang ら [6] は, Knuth のアルゴリズムよりも演算数が少なく, Knuth の除算アルゴリズムよりも 3 倍高速な多倍長整数の除算アルゴリズムを提案した. Mukhopadhyay と Nandy[10] は, Huang らの誤りを修正したうえで, このアルゴリズムの正しさを数学的に証明した. また, 性能は同じく標準アルゴリズムよりも 3 倍高速であることを示した. 本研究では, [10] のアルゴリズムにより仮数部の除算を実装した. このアルゴリズムに必要な演算は, 32 ビット符号なし整数の処理と, 64 ビット符号なし整数の加算と除算である.

#### 2.3.2 単精度・倍精度変数を初期値とした Newton 法

除算や逆数平方根を求める際には, 問題を非線形方程式の線形化によるニュートン法の処理で求めることも可能である. ニュートン法により多倍長精度 FP 演算で除算を計算するには, ニュートン法で逆数を求めてから被除数と乗算する. この手法の利点は, 初期値の推定を除けば, その実装は多倍長精度 FP 加減算と乗算の組み合わせに帰着されるため, 実装の手間が少ないことである. 今回ターゲットとする計算機は, いずれも単精度, 倍精度の FPU が利用可能なため, それを初期値として利用可能であり, 初期値推定のための特別な処理は必要ない. 標準的なニュートン法は二次収束のアルゴリズムであるため, 初期値が適切であれば, 既述の仮数部の除算をおこなう手法よりも高速となる可能性がある. この手法のために, FP フォーマットと単精度・倍精度変数との相互変換関数を実装した. 初期値の推定を単精度演算でおこなう場合には, 初期値の精度は 24 ビットであるため, 二次収束のアルゴリズムで $n_{\mathrm{man}} = 210$ ビットの結果を得るためには, ニュートン法のループが最低 4 回必要である. 初期値の推定を単精度演算でおこなえば, 初期値の精度は 53 ビットであるため, ニュートン法のループが 3 回必要である. 単精度, 倍精度における除算の計算は, それ自体で必要な時間が異なるため, どちらが高速化は一概にはいえない.

### 2.4 GPU での実装について

本研究では, 様々な x86_64 アーキテクチャの CPU と, 様々な GPU で多倍長精度 FP 演算の性能評価をおこなっている. 評価で利用した計算機システムのひとつは 16 コアを搭載した CPU システム (Xeon E5-2670) で, 単精度 FP 演算での理論性能は約 330 GFLOPS である. 評価で利用した GPU システムの代表 (Radeon R280X) の単精度 FP 演算での理論性能は約 4.2 TFLOPS である. この場合, CPU システムと GPU システムの性能差は 10 倍以上にもなる. 現在の GPU システムは, FPU の演算パイプラインを複数のスレッドで時分割で共有しており (SIMT 動作), 特にメモリアクセス速度を隠蔽するためには, CPU システムに比べて粒度が比較的大きい多くのスレッドが必要である. 多倍長精度 FP 演算は, メモリ読み書き演算数が多い演算粒度が大きい処理であり, GPU システムでは高性能を期待できる. 一方で, FP 演算のエミュレーションには, 条件分岐が何度か必要であるため, この点は SIMD プロセッサ的に動作する GPU にはあまり適していない. よって, 条件分岐を GPU に適した形に最適化するなどの工夫により, 最適化ができると思われる. また, 本研究では基本データ構造として符号なし 32 ビット整数を採用したが, GPU においても 32 ビット整数を基本データ構造として採用した方が, GPU のバックエンドでの最適化がより効果的な可能性がある. 実際に, CUDA での多倍長精度 FP 演算の実装のひとつである CUMP[11] は 32 ビット整数を採用している.

## 3. 性能評価

### 3.1 既存手法の性能評価

多倍長精度 FP 演算を実現する手法には, 様々ものがあり, 既述した QD のように, ライブラリや C++のクラスとして容易に利用可能なものもある.

Multiple Precision Arithmetic Library(GMP)[3] は, 様々なプラットフォーム用に最適化された総合的な多倍長演算ライブラリである. 整数演算や有理数演算とならんで, 多倍長精度 FP 演算ライブラリが実装されている. GMP の FP 演算ライブラリでは, 初期化をする際に変数の演算精度 ($n_{\mathrm{man}}$) を指定することができる. この演算ライブラリのひとつの特徴として, 演算毎に仮数部の大きさが必

| | 加算 | 乗算 | 除算 |
|---|---|---|---|
| Nehalem | 71 | 121 | 242 |
| SandyBridge | 54 | 93 | 206 |
| IvyBridge | 51 | 85 | 185 |
| Haswell | 45 | 84 | 191 |
| Magny-Cours | 92 | 145 | 332 |
| Bulldozer | 100 | 162 | 309 |
| Llano | 115 | 156 | 344 |

**表 1** MPFR 方式 ($n_{\mathrm{man}} = 210$, $n_{\mathrm{exp}} = 64$) の CPU における性能評価. 単位は 1 演算あたりのサイクル数.

| | 加算 | 乗算 | 除算 |
|---|---|---|---|
| Nehalem | 115 | 218 | 1113 |
| SandyBridge | 93 | 193 | 1021 |
| IvyBridge | 76 | 174 | 919 |
| Haswell | 65 | 169 | 1013 |
| Magny-Cours | 211 | 348 | 1572 |
| Bulldozer | 165 | 277 | 1527 |
| Llano | 227 | 374 | 1559 |

**表 2** QD 方式 ($n_{\mathrm{man}} = 209$, $n_{\mathrm{exp}} = 11$) の CPU における性能評価. 単位は 1 演算あたりのサイクル数.

要に応じて拡張されることがある. 例えば減算で演算する数値の絶対値の差が大きい場合, 仮数部が固定された通常の実装では情報落ちが発生する. このような場合に, GMP での FP 演算では, 演算結果の仮数部が動的に適切なサイズに拡張される. そのため, 演算精度が常に適切に保持される一方で, 実用上は演算性能の低下や, 必要以上の演算精度が保持される可能性があることや, 変数の格納形式が動的に変化するという問題がある.

MPFR に代表される整数演算を利用する手法では, FP 演算アルゴリズムを整数演算の組み合わせで実現している. MPFR は, GMP の整数演算ライブラリ部分をベースとして多倍長精度 FP 演算を実装している. MPFR も, GMP の FP 演算ライブラリと同様に, 指定された $n_{\mathrm{man}}$ の仮数部による FP 演算が可能である. GMP の FP 演算ライブラリとの違いは, MPFR は指定した仮数部を常に保持して演算をおこなうことであり, 演算の度に適切な丸め処理がおこなわれる. また四則演算だけではなく各種の数学関数も実装されている. MPFR は, GMP をベースとしており, GMP 自体が幅広い CPU アーキテクチャに最適化された実装のため, それと同様によく各 CPU に最適化されており高速である.

表 3.1 と 3.1 に, MPFR 方式 ($n_{\mathrm{man}} = 210$) と QD 方式 (4 語の倍精度変数によるため $n_{\mathrm{man}} = 209$ に相当) を様々な CPU で実行して計測した基本演算に必要な演算サイクル数を示す. いずれも Linux 上で実行し, ディストリビューションとして Ubuntu 14.04.1 LTS を利用した. プログラムの構築に使ったコンパイラ gcc のバージョンは 4.8.2 である. 各ライブラリは Ubuntu 14.04.1 LTS 附属のものを利用し, QD Library のバージョンは 2.3.11, GMP のバージョンは 5.1.3, MPFR のバージョンは 3.1.2-p3 である. この性能評価で利用した CPU は全て x86_64 アーキテクチャである. 表では, 異なるマクロアーキテクチャをコードネームで示している. x86_64 アーキテクチャで利用可能な ReaD Time Stamp Counter(RDTSC) により演算サイクル数を計測した. どの場合にも, 演算を十分な数だけ繰り返し, 1 演算毎に RDTSC で取得したサイクル数の総計を, 演算ループの回数で割り, 切り上げることで必要な演算サイクル数を推定した. いずれもシングルスレッドで測定している.

QD 方式は倍精度演算と条件分岐のみを利用し, MPFR 方式は 64 ビット整数演算と条件分岐の組み合わせで実装されている. いずれの場合にも, 乗算は加算に比べておおよそ 1.5 - 2 倍のサイクル数が必要であり, 除算は 3 - 10 倍のサイクル数が必要である.

本表における Intel 社の CPU は, Nehalem, SandyBridge, IvyBridge, Haswell の順にマイクロアーキテクチャが更新され, 演算器構成などがよりリッチに進化してきた. その進歩を反映し, より新しいマイクロアーキテクチャのほうが, ほとんどの場合により少ないサイクル数で実行可能である.

AMD 社の CPU は Magny-Cours, Bulldozer, Llano の 3 種であるが, 異なるマイクロアーキテクチャ間で大きな違いはない.

MPFR 方式と QD 方式を比べると, 仮数部のサイズは同等であり, MPFR では指数部がより拡張されている. ストレージサイズはどちらも 1 変数あたり 32 バイトである. この演算精度では, 整数演算を利用した MPFR のほうが全ての場合において高速である. QD 方式の拡張手法は, 語数が増えるほどソート処理が複雑となるため, この性能評価により八倍精度相当より高精度な演算が必要な場合には, QD 方式の拡張を利用するメリットがないことを意味する. ただし, QD 方式では AVX2 などの SIMD 演算を容易に利用することができる余地がある [15] ため, より最適化された実装においてはこの傾向は変化する可能性がある.

### 3.2 MYFP の CPU での性能評価

表 3.2 に, MYFP 方式を様々な CPU で実行して計測した基本演算に必要な演算サイクル数を示す. 測定環境は既述のものと同じである. 除算については, 仮数部を直接除算する手法 (除算と示す), 逆数の初期値を単精度で推定するニュートン法 (除算 F と示す), 逆数の初期値を倍精度で推定するニュートン法 (除算 D と示す) の 3 パターンについて比較をした.

MPFR 方式と比較すると, 加減算, 乗算と除算については, MYFP 方式はおおよそ 1.5 - 2 倍ほど必要なサイクル数が増えている. これは MPFR が内部で利用している GMP ライブラリは, 各 x86_64 のマイクロアーキテクチャ毎にア

| | 加算 | 乗算 | 除算 | 除算 F | 除算 D |
|---|---|---|---|---|---|
| Nehalem | 111 | 167 | 2026 | 2087 | 1155 |
| SandyBridge | 91 | 148 | 1949 | 1986 | 1107 |
| IvyBridge | 80 | 142 | 1822 | 1737 | 970 |
| Haswell | 81 | 136 | 2029 | 1885 | 1049 |
| Magny-Cours | 174 | 275 | 2866 | 2925 | 1654 |
| Bulldozer | 186 | 373 | 3223 | 3044 | 2304 |
| Llano | 189 | 299 | 2884 | 2909 | 1657 |

**表 3** MYFP 方式 ($n_{\mathrm{man}} = 210$, $n_{\mathrm{exp}} = 30$) の CPU における性能評価. 単位は 1 演算あたりのサイクル数.

センブラで記述された最適化ルーチンを採用しているのに対して, MYFP の実装は C 言語で記述されているためと考えられる. QD 方式と比較すると, MYFP 方式は同等か若干高速であることがわかった. MYFP 方式での除算については, 除算 D が他の方式より明らかに高速である.

### 3.3 MYFP の OpenCL での性能評価

最後に, MYFP 方式を OpenCL で動作するように修正し, 各種のアーキテクチャで性能評価をおこなった. C 言語でのオリジナルのソースコードを, OpenCL カーネル化するために必要な修正は必要最低限であり, 実質的に同一コードが動作可能である. ただし, OpenCL カーネルとして動作させるためのラッパー関数が追加で必要である.

OpenCL による性能評価では, 実行サイクル数を計測することはできないため, OpenCL API により得られるカーネル実行時間 (これにはホストと OpenCL デイバイス間のデータ転送時間は含まない) により性能を計測した. 表 3.3 に, 様々な GPU での性能評価の結果を示す. 性能の単位は MFLOPS である. 2 つ目のコラムは各 GPU の単精度 FP 演算による理論演算性能を示す. この性能評価では, GPU の演算ユニットを可能な限り利用するように約 500 万要素の入力値に対してカーネルを実行して, その実行時間を計測し, 総演算数を実行時間で割り, 切り捨てることで演算性能を計算した.

Xeon E5-2670 のみが CPU であり, 他は全て GPU である. Xeon E5-2670 は 2 CPU 構成で, トータルで 16 コアのシステムである. 空欄の箇所は, 演算結果が誤っていたか, 演算自体が実行できなかったことを示す.

実行環境として, NVIDIA 社の GPU は CUDA 6.5.14 を利用した. AMD 社のコンシューマ向け GPU(Radeon HD6970 および Radeon R280X) では, Catalyst 14.40.4, OpenCL Driver 1642.5 を利用した. AMD 社のワークステーション向け GPU(FirePro W8000 および W8100) では, Catalyst 14.40.3, OpenCL Driver 1573.4 を利用した. OpenCL で実装したことで, マルチコア CPU システムや, 様々な GPU で動作することを確認した.

本論文の実装では, 主として 32 ビット整数演算を利用している. 表 3.3 のおおまかな傾向としては, 単精度での演算性能が高い計算機では, MYFP 形式の性能も高いことがわかる. GPU における内部の演算器実装の詳細は明らかではないが, 各種情報から, 単精度浮動小数点演算器と, 32 ビット ALU は内部的に演算リソースを共有していると推測される. ただし, 一部の GPU では性能が不自然に低下している場合がある. 例えば, Radeon R280X と FirePro W8000 のマイクロアーキテクチャはコードネーム Tahiti と呼ばれる同一のチップであるが, 利用可能な OpenCL バックエンドのバージョンが異なるため W8000 の性能は低い. 同様の傾向は, より新しい世代のマイクロアーキテクチャ Hawaii を搭載した FirePro W8100 でも同様である. この両者の GPU では, 除算 F と除算 D が正しく実行できなかった. これらの GPU アーキテクチャ用の, OpenCL バックエンドには改良の余地が多くある. NVIDIA 社の GPU では, おおよそ単精度 FP 演算性能と相関して, 性能が向上している. 現時点で, MYFP 方式の性能が最も高いのは Radeon R280X であり, 全 16 コアの Xeon E5-2670 と比較すると加算, 乗算では約 10 倍, 除算では 5 - 10 倍高速である.

## 4. まとめ

本論文では, 整数演算による多倍長浮動小数点演算エミュレーション手法を, OpenCL カーネルとして実装し, その性能評価を CPU システムと GPU システムでおこなった. 我々の設計・実装した MYFP 手法は, 8 倍精度相当の演算精度を保持し, 指数部は 30 ビットに拡張されているため, 指数部のサイズがクリティカルであるファインマン・ダイアグラムの直接計算に適した手法である. OpenCL カーネルを Radeon R280X で性能評価した結果, 加算では約 2.3 GFLOPS, 乗算では約 1.8 GFLOPS, 除算では約 0.23 GFLOPS の性能を得た. この性能は, 16 コアのマルチコア CPU システムより約 10 倍高速である. 一方で, MYFP 手法と MPFR 手法の性能を CPU システムで比較すると, おおよそ 2 倍 MPFR のほうが高速である. このことから, GPU システムでの MYFP 手法の性能は, 最も最適化された CPU システムと比べると 5 倍の性能差となった. 今後, MYFP を利用することで, GPU によりファインマン・ダイアグラムの直接計算を実用的に計算することをめざす.

### 参考文献

[1] Dekker, T.: A Floating-Point Technique for Extending the Available Precision, *Numerische Mathematik*, Vol. 18, pp. 224–242 (1971).
[2] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P. and Zimmermann, P.: MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding, *ACM Trans. Math. Softw.*, Vol. 33, No. 2, pp. 1–15 (online), DOI: 10.1145/1236463.1236468 (2007).
[3] GMP: GNU Multiple Precision Arithmetic Library.
[4] Hida, Y., Li, X. and Bailey, D.: Algorithm for quad-double precision floating point arithmetic, *Proc. 15th*

| | SP 性能 | 加算 | 乗算 | 除算 | 除算 F | 除算 D |
|---|---|---|---|---|---|---|
| Xeon E5-2670 | 3.3e5 | 247 | 180 | 21.3 | 20.4 | 38.9 |
| GeForce GTX570 | 1.4e6 | 244 | 105 | 17.1 | 11.0 | 13.3 |
| Radeon HD6970 | 2.7e6 | 1461 | 213 | 11.9 | 22.4 | 9.5 |
| FirePro W8000 | 3.2e6 | 1546 | 82.6 | 35.4 | – | – |
| Tesla K20c | 3.5e6 | 349 | 138 | 22.6 | 15.2 | 15.8 |
| Radeon R280X | 4.2e6 | 2324 | 1835 | 190 | 61.7 | 231 |
| FirePro W8100 | 4.2e6 | 260 | 44.7 | 24.4 | – | – |
| GeForce TITAN | 4.5e6 | 449 | 192 | 31.8 | 21.1 | 20.5 |

**表 4** MYFP 方式 ($n_{\mathrm{man}} = 210$, $n_{\mathrm{exp}} = 30$) の CPU における性能評価. 単位は MPFLOS.

*IEEE Symposium on Computer Architecture*, pp. 287–302 (2001).

[5] High-Precision Software Directory: .

[6] Huang, L., Zhong, H., Shen, H. and Luo, Y.: An Efficient Multiple-Precision Division Algorithm, *Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on*, pp. 971–974 (online), DOI: 10.1109/PDCAT.2005.79 (2005).

[7] Knuth, D.: *The Art of Computer Programming vol.2 Seminumerical Algorithms*, Addison Wesley, Reading, Massachusetts, first edition (1998).

[8] Knuth, D.: *The Art of Computer Programming vol.2 Seminumerical Algorithms*, Addison-Wesley Longman, Inc, Reading, Massachusetts, third edition (1998).

[9] Krandick, W. and Johnson, J.: Efficient multiprecision floating point multiplication with optimal directional rounding, *Computer Arithmetic, 1993. Proceedings., 11th Symposium on*, pp. 228–233 (online), DOI: 10.1109/ARITH.1993.378088 (1993).

[10] Mukhopadhyay, D. and Nandy, S. C.: Efficient multiple-precision integer division algorithm, *Information Processing Letters*, Vol. 114, No. 3, pp. 152 – 157 (online), DOI: http://dx.doi.org/10.1016/j.ipl.2013.10.005 (2014).

[11] Nakayama, T. and Takahashi, D.: Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing, *Proc. 23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, pp. 343–349 (online), DOI: 10.2316/P.2011.757-041 (2011).

[12] Yuasa, F., Ishikawa, T., Hamaguchi, N., Koike, T. and Nakasato, N.: *Acceleration of Feynman loop integrals in high-energy physics on many core GPUs*, *Journal of Physics: Conference Series*, Vol. 454, No. 1, IOP Publishing, p. 012081 (online), DOI: 10.1088/1742-6596/454/1/012081 (2013).

[13] 山田 進, 佐々成正, 今村俊幸, 町田昌彦：4 倍精度基本線形代数ルーチン群 QPBLAS の紹介とアプリケーションへの応用, 情報処理学会研究報告. 計算機アーキテクチャ研究会報告, Vol. 2012, No. 23, pp. 1–6（オンライン), 入手先 ⟨http://ci.nii.ac.jp/naid/110009490634/⟩ (2012).

[14] 中里直人, 石川 正, 牧野淳一郎, 湯浅富久子：アクセラレータによる四倍精度演算, 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], Vol. 2009, No. 39, pp. 1–7（オンライン), 入手先 ⟨http://ci.nii.ac.jp/naid/110007995434/⟩ (2009).

[15] 菱沼利彰, 藤井昭宏, 田中輝雄, 長谷川秀彦：AVX2 を用いた倍精度 BCRS 形式疎行列と倍々精度ベクトル積の高速化, 情報処理学会論文誌コンピューティングシステム（ACS), Vol. 7, No. 4, pp. 25–33 (2014).