Vol.2011-ARC-197 No.25
Vol.2011-HPC-132 No.25
2011/11/29

GPU 上における多倍長精度浮動小数点演算の実装

中 山 空 星[†1]　高 橋 大 介[†2]

本研究では，GPU 上で高速な多倍長精度浮動小数点演算の実装を行った．本実装は GMP ライブラリと互換性があり，CUDA 上で動作する．NVIDIA Tesla C2050 上での性能評価において，先行研究の GPU 用多倍長精度演算ライブラリである GARPREC と比較し，本実装は最大約 2.6 倍の性能で動作することを確認した．また，既存の CPU 用多倍長精度演算ライブラリと性能を比較した結果も示す．

1. はじめに

数値計算においては，一般に倍精度浮動小数点演算が使用される．倍精度浮動小数点数は 10 進約 16 桁の数値を表わすことができるが，科学技術計算の分野ではより高い精度が必要となることが多くある．また，浮動小数点演算には丸め誤差があり，大規模計算では無視することができないほど蓄積することがある．このような背景を受け，様々な高精度演算の試みが古くから行われてきたが，その一つに多倍長精度演算がある．多倍長精度演算は，任意の精度での計算を実現するものであり，通常ソフトウェアによって実装される．計算コストが高い一方，実質的な精度限界がないため，アルゴリズムを変更せずに精度改善を図ることができる．

一方，GPU（Graphic Processing Unit）による計算加速が近年注目されている．数百のコアを持つ GPU は CPU を大きく上回る演算性能を持つため，多くの計算やアプリケーションにおいて GPU による高速化が行われており，GPGPU（General Purpose computing on GPU）の研究が盛んである．しかし当然のことながら，これらの研究のほとんどには単精度または倍精度演算が用いられている．GPU コンピューティングにおいて，高精度演算のサポートは実装コストが高い．

そこで，本研究では，GPU 上で動作する高速な多倍長精度浮動小数点演算の実装を行った．CPU 上で多倍長精度浮動小数点演算が可能なライブラリは多く存在するが[1)2)3)4)]，本実装はこれを GPU 上で実現し，高精度が要求される計算やアプリケーションの GPU を用いた加速を可能とする．本稿では，本研究で行った多倍長精度浮動小数点演算の設計と実装を述べ，既存の CPU 用多倍長精度浮動小数点演算の実装および先行研究に対する優位性を示す．

2. 関連研究

GMP（The GNU Multiple Precision Arithmetic Library）[1)] は，C 言語とアセンブリ言語で実装された，C 言語および C++用の高速な多倍長精度演算ライブラリである．多倍長精度浮動小数点演算だけでなく多倍長精度整数演算もサポートしているため，多くのシステムが内部で GMP を利用している．

ARPREC および QD[2)] は，Bailey らによる Fotran および C++用の高精度演算ライブラリである．ARPREC は，内部で倍精度演算を用いた多倍長精度演算ライブラリである．高速な FPU を搭載したマシン上では特に性能が良い．QD は，4 倍・8 倍精度浮動小数点演算ライブラリである．4 倍・8 倍精度の仮数部は，それぞれ 2 つおよび 4 つの倍精度数の組み合わせによって表現される．

GPU 上での多倍長精度整数演算に関しては，いくつかの研究が報告されている．Zhao らは，GPU 用多倍長精度整数演算ライブラリを実装し[5)]，GMP によるマルチコア CPU 上での多倍長精度整数演算を大きく上回る性能を達成したと報告している．Niall らは，高精度整数演算（加算，減算および乗算）の GPU を用いた並列実装を提案している[6)]．そのアプローチは，桁数が大きい 2 つの整数間の計算を，GPU で高速化するものである．

また，Graça らは double-float 型倍精度浮動小数点演算を実装している[7)]．Thall も double-float 型倍精度および quadruple-float 型 4 倍精度浮動小数点演算を実装している[8)]．これらの研究は，倍精度をハードウェアでサポートしていない初期の GPU コンピューティングにおいては有用であった．しかし，計算によってはより高精度な演算が求められているものもある．

Lu らは，ARPREC および QD を基にして，GPU 用の多倍長精度演算ライブラリ GARPREC および 4 倍・8 倍精度浮動小数点演算ライブラリ GQD を実装している[9)]．GARPREC において GPU 上での多倍長精度浮動小数点演算は実現されているが，その評価は NVIDIA Tesla アーキテクチャの GPU 上で行われたものであるため，GARPREC を NVIDIA Fermi アーキテクチャの GPU で再評価し，本実装と比較する必要がある．

†1 筑波大学大学院システム情報工学研究科
†2 筑波大学システム情報系

© 2011 Information Processing Society of Japan

Vol.2011-ARC-197 No.25
Vol.2011-HPC-132 No.25
2011/11/29

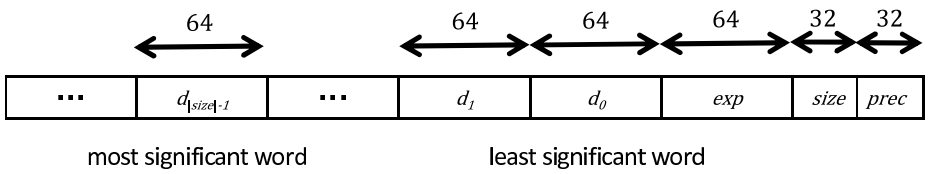


図 1 多倍長精度浮動小数点数のデータ構造

3. 設計および実装

3.1 概　　要

本研究では，GPU コード中で GMP の代替として使用できる多倍長精度浮動小数点演算ライブラリとして実装を行った．多倍長精度演算，特に GMP の関数を含むコードの GPU における並列化を支援するのが目的である．すべてのインタフェースに GMP のインタフェースとの互換性を持たせたことによって，わずかなコード変更で GMP を使用して書かれたコードが GPU 上で動作するようにできる．また，GMP との通信インタフェースを用意したことから，GMP との協調計算も容易である．

ARPREC の GPU 移植である GARPREC は，GPU 用多倍長精度浮動小数点演算の先駆けとなったライブラリである．GARPREC は，内部で倍精度演算を使用して多倍長精度演算を実現する．NVIDIA Tesla アーキテクチャの GPU は FPU のみ搭載しており，整数演算はエミュレーションによって実現されていたため，これは妥当な設計であると考えられる．しかし，倍精度数は多倍長精度演算には不要な符号と指数部のためのビットを保持しているため，倍精度演算を使用するとメモリに無駄が生じ，メモリアクセス回数が増える恐れがある．また，NVIDIA Fermi アーキテクチャの GPU には ALU も搭載されており，整数演算も浮動小数点演算と同等の速度となっている．

本実装では，メモリを効率的に使用するために，多倍長精度演算の実現に内部で 64 ビット整数演算を使用した．ライブラリとしては，型定義，データ操作関数（メモリ確保，データ転送等）および多倍長精度演算関数を提供する．これらは，C 言語，C++および CUDA[10] から利用可能である．データ操作関数は CPU 上で動作し，多倍長精度演算関数は GPU 上で動作する．

本実装は，NVIDIA 社製の Compute capability 2.0 以上の GPU を搭載した 64 ビットオペレーティングシステムのマシン上で動作する．

3.2 データ構造

多倍長精度浮動小数点数は，**図 1** で示すように，可変長の仮数部，64 ビットワードの符号付き指数部，および仮数部の大きさを表す 2 つの 32 ビット整数で表現される．

図 1 において，$prec$ と $size$ は 32 ビット整数である．$prec$ は仮数部の精度を示し，$size$ は多倍長精度数の符号と現在使用している仮数部の長さを示す．指数部 exp は 64 ビット符号付き整数で，exp の底は 2^{64} である．$\{d_n\}$ で表される仮数部の絶対値は，64 ビット符号無し整数の配列であり，リトルエンディアン方式で格納される．すなわち d_0 が最下位ワードであり，$d_{|size|-1}$ が最上位ワードである．図 1 が表す値は，

$$\mathrm{sign}(size) \times 2^{64exp} \times \sum_{n=1}^{|size|} d_{|size|-n} \times 2^{-64n}$$

である．ただし，

$$\mathrm{sign}(x) = \begin{cases} 1 & (x > 0) \\ -1 & (x < 0) \\ 0 & (x = 0) \end{cases}$$

である．

実装にあたっては，ベクトルや行列のような多数の多倍長精度数を処理することを仮定している．例えば，長さ m のベクトル

$$\boldsymbol{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix} \ \text{および} \ \boldsymbol{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix}$$

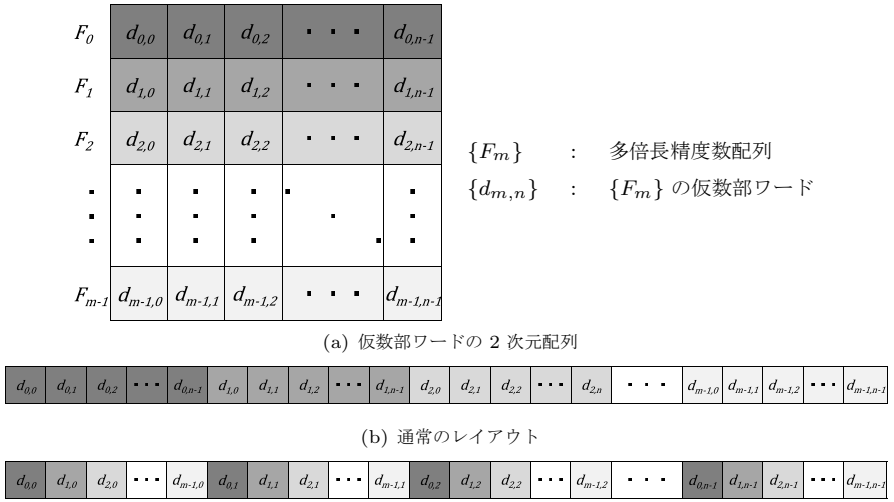
における，要素どうしの加算 $\boldsymbol{c} = \boldsymbol{a} \oplus \boldsymbol{b}$ は，

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} a_0 & + & b_0 \\ a_1 & + & b_1 \\ & \vdots & \\ a_{m-1} & + & b_{m-1} \end{bmatrix}$$

である．

© 2011 Information Processing Society of Japan

Vol.2011-ARC-197 No.25
Vol.2011-HPC-132 No.25
2011/11/29



(a) 仮数部ワードの 2 次元配列

(b) 通常のレイアウト

(c) 転置されたレイアウト

図 2 多倍長精度数配列の 2 つの異なるメモリレイアウト

多数の多倍長精度数を表現するために配列型を定義した．複数の GPU スレッドがそれぞれの多倍長精度数を読み書きするとき，各スレッドはその仮数部ワードへの同時アクセスを何度も繰り返す．通常の多倍長精度数配列では，データの並びは，**図 2(b)** に示すような 2 次元配列と同等である．この場合，スレッドが同時にアクセスするデータは連続的に配置されていない．一方で，この 2 次元配列を転置する場合，**図 2(c)** に示すように，同時にアクセスされるデータが連続的に配置される．GPU は，各スレッドがオフチップメモリ（ビデオ RAM）のデータに同時にアクセスする際に，より効率的に動作するので，後者のレイアウトを用いて配列型を定義した．

GARPREC も図 2(c) に示すレイアウトの配列型をサポートしている．Lu らは，図 2(b) に示すレイアウトよりも図 2(c) に示すレイアウトを用いた方が，最大で 3 倍速い結果が得られたと報告している[9]．

3.3 実　　装

実装は C 言語および CUDA を用いて行い，GMP 5.0.1 をベースとした．現在までに 3 つの多倍長精度演算（加算，減算および乗算）を実装している．そのすべてに筆算アルゴリズムを使用しているため，n ワード精度の数において，加算と減算は $O(n)$ の演算量であるが，乗算は $O(n^2)$ の演算量である．これらの演算は，GPU カーネル内からのみ使用される`__device__`関数として実装されている．各 GPU スレッドはこれらの関数を呼び出し，それぞれの多倍長精度演算の結果を得る．このことから分かるように，各多倍長精度演算は並列化されていない．

演算の途中結果を格納する一時バッファとして，ローカルメモリを使用した．ローカルメモリは，各スレッドが GPU のオフチップメモリに保持するプライベートな記憶領域である．GPU コード内で大きなサイズの配列を宣言すると，自動的にローカルメモリに割り当てられる．これはつまり，一時バッファは静的な固定サイズの配列であるということである．配列サイズは必要ならばコンパイル前に指定できる．一般には，メモリを節約するために一時バッファは動的に確保されるべきである．そのような実装にしなかったのは，CUDA のカーネル内 `malloc()` 関数，すなわち GPU カーネル内動的メモリ確保は非常にコストが高いためである．

多倍長精度乗算の実装には，いくつかのテクニックを使用している．多倍長精度乗算で中心となる処理は，128 ビット符号無し整数の積を算出する 64 ビット符号無し整数の乗算である．純粋な C 言語における演算はオペランドの精度で行われるため，128 ビット整数積を求める乗算は**図 3** のように複雑な処理によって実現する必要がある．しかし，CUDA には，2 つの 64 ビット符号無し整数からその積の上位 64 ビットを求める組み込み関数`__umul64hi()`が用意されている．これを利用することで，**図 4** のように簡単な処理で実現することができた．また，メモリアクセス削減のためにレジスタブロッキングを使用した．このテクニックを使わない場合，被乗数に乗数を 1 ワードずつ掛けていく．例えば，4 段のレジスタブロッキングを行うと，被乗数に乗数を 4 ワードずつまとめて掛けることができる．これによって，被乗数の読み込み回数および一時バッファへのアクセス回数を 4 分の 1 まで削減できる．このように段数を増やすことでオフチップメモリへのアクセス回数は減少していくが，レジスタの使用量は増加する．GPU スレッドが十分なレジスタを利用できないとき，データはローカルメモリに待避されるため（Register spilling），レジスタが不足すると，逆にオフチップメモリへのアクセス回数が増加する．最適な段数を求めるため実験を行ったところ，4 段を超えるレジスタブロッキングを行うと性能は低下した．そこで本実装では，4 段

© 2011 Information Processing Society of Japan

Vol.2011-ARC-197 No.25
Vol.2011-HPC-132 No.25
2011/11/29

Require: u and v are 64-bit unsigned integers
Ensure: $W = w_1 \times 2^{64} + w_0$ is the 128-bit product.
$u_l \Leftarrow \texttt{lower32bits}(u)$
$u_h \Leftarrow \texttt{higher32bits}(u)$
$v_l \Leftarrow \texttt{lower32bits}(v)$
$v_h \Leftarrow \texttt{higher32bits}(v)$
$x_0 \Leftarrow u_l \times v_l$
$x_1 \Leftarrow u_l \times v_h$
$x_2 \Leftarrow u_h \times v_l$
$x_3 \Leftarrow u_h \times v_h$
$x_1 \Leftarrow x_1 + \texttt{lower32bits}(x_0) + x_2$
if x_1 overflows **then**
  $x_3 \Leftarrow x_3 + 2^{32}$
end if
$w_1 \Leftarrow x_3 + \texttt{higher32bits}(x_1)$
$w_0 \Leftarrow \texttt{lower32bits}(x_1) \times 2^{32} + \texttt{lower32bits}(x_0)$

図 3 128 ビット符号無し整数の積を算出する 64 ビット符号無し整数乗算

までのレジスタブロッキングを使用した．

一方，GARPREC の乗算は，レジスタブロッキングを使用していない．しかし，n ワード精度の浮動小数点数どうしの乗算において，本実装では $2n$ ワード精度の積をすべて求めてから丸めを行うが，GARPREC では計算量を減らすために要求される精度よりも下位のワードの計算を省略している．これは，GMP に準拠する本実装と ARPREC に準拠する GARPREC の実装方針の違いである．計算を省略しない本実装の方が，GARPREC に比べ演算の精度は高い．

また，$O(n^{\log_2 3})$ の Karatsuba 乗算[11] や $O(n \log n \log\log n)$ の FFT（高速フーリエ変換）乗算[12] などの高速乗算法も存在するが，本実装および GARPREC では実装されていない．

仮数部ワードは，本実装ではリトルエンディアン方式で格納されるが，GARPREC ではビッグエンディアン方式で格納される．つまり最上位ワードは，本実装においては仮数部ワードの最後尾に，GARPREC においては先頭に位置する．そのため，最上位ワードでキャリーが発生した場合，互いに異なる処理が行われる．本実装では最上位ワードの後ろにキャリーを格納するだけであるが，GARPREC ではすべてのワードを隣に移してキャリーを先頭に格納しなければならない．このワードの移動処理は，ワード数が増えるに従って時間がかかってしまう．

Require: u and v are 64-bit unsigned integers
Ensure: $W = w_1 \times 2^{64} + w_0$ is the 128-bit product.
$w_1 \Leftarrow u \times v$
$w_0 \Leftarrow \texttt{__umul64hi}(u, v)$

図 4 `__umul64hi()` を利用した 128 ビット符号無し整数の積を算出する 64 ビット符号無し整数乗算

仮数部ワードの大きさは，一つのワードあたりの有効なビット数に依存する．仮数部ワードが整数型であれば，ワードのすべてのビットを有効に使用できる．倍精度浮動小数点型のワードは，その仮数部である 53 ビットが最大で使用できる有効なビットである．GARPREC の仮数部ワードは，キャリーを考慮して 48 ビットを有効ビットとしている．多倍長精度数を省メモリに表現するために，本実装では可変長の仮数部の格納に 64 ビット整数の配列を用いた．これによって冗長な仮数部ビットを発生させることなくデータを格納することができる．

本実装では，10 進 k 桁の数を表現するのに最低で $\lceil k \log_2 10 \rceil$ ビット必要であり，そのため $\lceil \lceil k \log_2 10 \rceil / 64 \rceil$ 個の 64 ビットワードが必要である．ヘッダーワード，指数部ワード，要求精度と計算中のキャリーのための 2 つの余分なワードを含めると，$32 + 8 \cdot \lceil \lceil k \log_2 10 \rceil / 64 \rceil$ バイトのメモリが要求される．これは，10 進 100 桁の数を格納するのに 80 バイト必要であること意味する．10 進 1,000 桁では 448 バイト，10 進 10,000 桁では 4,192 バイト必要であり，10 進 10,000 桁，65,536 要素のベクトルにおける要素どうしの加算 $\boldsymbol{c} = \boldsymbol{a} \oplus \boldsymbol{b}$ を行うためには，786MB が必要である．

4. 性能評価

4.1 評価方法

本実装の性能を評価するため，現在利用可能な複数の多倍長精度演算ライブラリと比較した．**表 1** に使用したライブラリを示す．また，**表 2** に評価を行った環境を示す．

加算，減算および乗算の性能をベクトルの要素どうしの計算で測定した．正確な測定のために，計算を 1 秒以上繰り返し実行し，1 秒あたりに実行された多倍長精度演算の回数を算

© 2011 Information Processing Society of Japan

Vol.2011-ARC-197 No.25
Vol.2011-HPC-132 No.25
2011/11/29

表 1 性能評価に用いた多倍長精度演算ライブラリ

Library	Version	Hardware	Internal arithmetic
ARPREC	2.2.7	CPU	Double-precision floating-point
GARPREC	1.2	GPU	Double-precision floating-point
GMP	5.0.2	CPU	64-bit integer
CUMP（本実装）	–	GPU	64-bit integer

表 2 評価環境

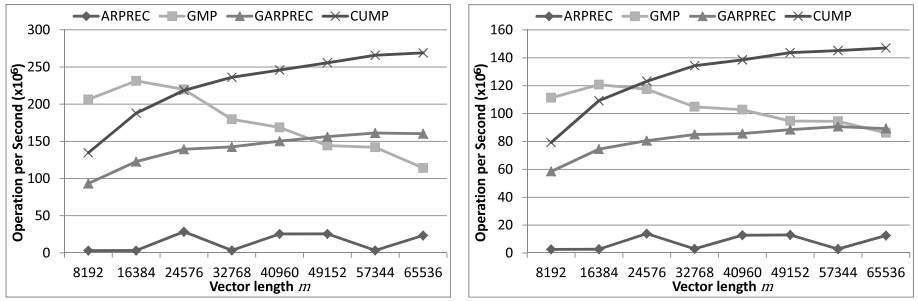
CPU	AMD Opteron 6134 2.3GHz 8cores×2
RAM	DDR3 SDRAM 1333MHz 2GB×2 (Single-channel)
GPU	NVIDIA Tesla C2050 1.15GHz 448 CUDA cores
Video RAM	GDDR5 SDRAM 384bit 1.5GHz 3GB (ECC on)
OS	CentOS release 5.5
Language	C, C++, OpenMP, CUDA
Compiler	nvcc 3.2 (-O2 -arch=sm_20), g++ (gcc) 4.1.2 (-O2)

出した．評価で使用したすべてのベクトルは長さ m であり，先に生成された同じ多倍長精度の一様乱数で構成されている．評価においては，ベクトル長 m を変化させて性能を測定した．

CPU 上で動作するライブラリの評価の際には，評価に使用したマシンの CPU コア数と一致するように 16 スレッドを使用した．実際の計算時間を反映した評価を行うために，スレッド生成時間は実行時間に含めなかった．

GPU はスレッド構成によって性能が変化することが知られている．CUDA では，スレッドはスレッドブロックという単位でまとめられ，スレッドブロックの数とスレッドブロックあたりのスレッド数を指定することで，多量のスレッドを管理する．GPU 上での評価では，複数のスレッド構成下で性能測定を行い，最も良い性能を使用した．CUDA 3.2 では，NVIDIA Tesla C2050 利用時には 2 つの L1 キャッシュの構成が選択できるが，より大きい L1 キャッシュサイズ（48KB）の構成を使用した．前述の通り，測定を可能な限り一貫したものにするため，メインメモリと GPU 間のデータ転送時間は実行時間に含めなかった．

減算のアルゴリズムは加算よりも複雑であるが同様のものであるため，減算の性能は加算の性能と同じ傾向を示す．実際に，減算の性能評価結果は，加算に比べ全体的に 1 割から 2 割ほど低かったものの，加算の性能評価結果と比較的同じ傾向であった．そこで冗長な議論を避けるため，本稿では減算の性能評価結果に関しては省略する．



(a) 加算　　(b) 乗算

図 5 10 進 100 桁の数値どうしの加算および乗算の性能評価結果

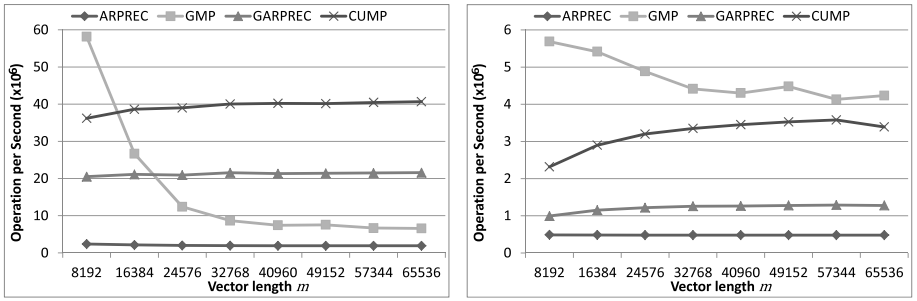
4.2 性能評価結果

この節では，本実装を便宜的に CUMP と呼ぶ．**図 5** に 10 進 100 桁の数値どうしの加算および乗算の性能評価結果を示す．加算，乗算ともに結果は似た傾向であった．ベクトル長 m が 65,536 に達するまで，CUMP および GARPREC の性能は m に従って向上した．その後，性能は飽和する傾向にあり，ベクトル長が増加しても性能が向上するようには見えない．その際，CUMP は GARPREC の約 1.7 倍の性能であった．一方で m が 16,384 を超えると，GMP の性能は低下し始め，最終的に GARPREC を下回った．これは，コアあたり 128KB，16 コア合わせて 2MB の CPU の L1 キャッシュよりもデータサイズが大きくなったためである．ARPREC の性能はベクトル長の増加に従って向上せず，今回評価したライブラリの中では最も低かった．

図 6 に 10 進 1,000 桁の数値どうしの加算および乗算の性能評価結果を示す．ベクトル長の増加に従って，GMP 以外のすべての性能はほぼ同様に向上した．加算の評価結果では，GMP の性能は $m = 8{,}192$ に達した後に急速に低下した．これは，CPU の L2 キャッシュよりもデータサイズが大きくなったためである．使用した CPU には 512 KB の L2 キャッシュが搭載されており，16 コア合わせると 8MB である．乗算の評価結果では，GMP の性能は低下したものの常に最も高かった．GMP は前述の高速乗算法をサポートしているため，計算コストを下げることで乗算をより高速に行うことができる．CUMP は，GARPREC より加算で約 1.9 倍，乗算で約 2.6 倍の性能であった．

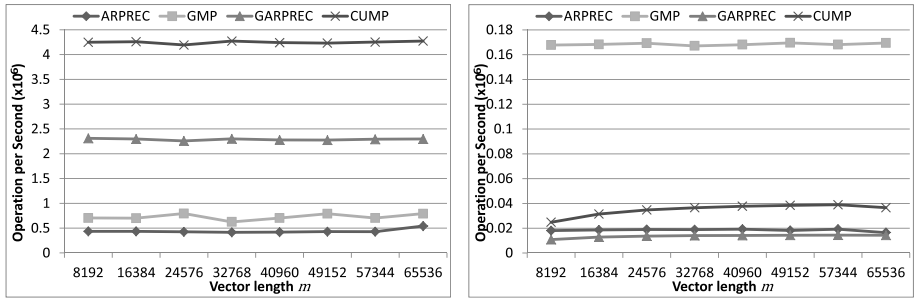
図 7 に 10 進 10,000 桁の数値どうしの加算および乗算の性能評価結果を示す．ベクトル

© 2011 Information Processing Society of Japan



(a) 加算　　(b) 乗算

図 6　10 進 1,000 桁の数値どうしの加算および乗算の性能評価結果



(a) 加算　　(b) 乗算

図 7　10 進 10,000 桁の数値どうしの加算および乗算の性能評価結果

長の増加に関わらず，すべてにおいて結果はほとんど変化しなかった．加算の評価結果では CUMP の性能が最も高く，GARPREC の約 1.9 倍であった．一方，乗算の評価結果では，高速乗算法の高い効果により GMP の性能が最も高かった．CUMP は GARPREC の約 2.6 倍の性能であった．

4.3 考　　察

性能評価の結果，本実装の性能が GARPREC より有意に高いことが分かる．これについては，メモリアクセス回数の違いで説明することができる．両方の実装で同じ筆算アルゴリズムを使用しているが，本実装は GARPREC よりもオフチップメモリにアクセスしない．メモリアクセス回数と比較して計算のために必要な命令数が少ないため，オフチップメモリへのアクセス時間は GPU の実行時間に重大な影響を及ぼす．

まず，両方の実装の多倍長精度加算の性能について議論する．n ワードの精度において，本実装ではオフチップメモリに $5n$ 回，GARPREC では $7n$ 回のメモリアクセスが必要である．共通しているのは，2 つのオペランドをロードするための $2n$ 回，一時バッファに加算の結果を格納するための $2n$ 回，一時的バッファからの出力へ結果を移すための n 回のメモリアクセスである．それらに加えて，GARPREC ではキャリーによるワード移動処理のために，さらに $2n$ 回のメモリアクセスが発生する．また，本実装において，仮数部に n ワード必要な多倍長精度数は，GARPREC では $(64/48)n$ ワード必要となる．したがって，GARPREC は本実装よりも $7 \cdot (64/48)n/5n \approx 1.87$ 倍多くメモリアクセスしている．加算の性能評価結果を見ると，この値とほぼ一致していることが分かる．

次に，両方の実装の多倍長精度乗算の性能について議論する．n ワードの精度において，n が十分に大きい場合，n^2 に比例した回数のメモリアクセスが必要である．要求精度より下位ワードの計算を省略する GARPREC では，オペランドの読み込みにおよそ $n^2/2$ 回，一時バッファの読み書きにおよそ n^2 回，合わせておよそ $3n^2/2$ 回のメモリアクセスが発生する．一方，下位ワードの計算を省略しない本実装では，通常，オペランドの読み込みにおよそ n^2 回，一時バッファの読み書きにおよそ $2n^2$ 回，合わせておよそ $3n^2$ 回のメモリアクセスが発生する．実際には 4 段までのレジスタブロッキングを使用し，最小でメモリアクセスを 4 分の 1 に削減できるため，およそ $3n^2/4$ 回で済む．したがって，仮数部ワードの大きさを考えると，GARPREC は本実装よりも $\left(3 \cdot (64/48)^2 n^2/2\right)/(3n^2/4) \approx 3.56$ 倍多くメモリアクセスしている．しかしながら，多倍長精度乗算の性能は複雑な要因が関係している．乗算でのメモリアクセスパターンには，参照の局所性があるのでキャッシュが有効に働く．さらに，乗算は計算のための命令数が多いため，メモリアクセスが減少するほど計算時間が実行時間に占める割合が増大する．そのため，GARPREC と本実装との性能比は，メモリアクセス回数比の 3.56 よりも低くなってしまう．実際，性能評価の結果は，性能比が最大で約 2.6 にとどまっていることを示している．

5. ま　と　め

本稿では，GPU コンピューティングのための多倍長精度浮動小数点演算の設計と実装を示した．現在までに，加算，減算および乗算を実装している．

© 2011 Information Processing Society of Japan

Vol.2011-ARC-197 No.25
Vol.2011-HPC-132 No.25
2011/11/29

本実装では，省メモリに多倍長精度数を表現するために 64 ビット整数配列を使用し，キャリーのコストを抑えるために仮数部ワードをリトルエンディアン方式で格納している．さらに，乗算の実装には，メモリアクセスを減少させるためにレジスタブロッキングを使用した．

GPU 上での多倍長精度浮動小数点演算について，加算と減算は桁数が大きく計算規模が大きいほど効果的であり，乗算は 10 進 1,000 桁未満においては効果が期待できる．また，NVIDIA Tesla C2050 上で，本実装は GARPREC の最大約 2.6 倍の性能であった．

今後は，データ構造を見直し，いくつかの高速乗算法を実装することで，巨大な桁数における高速化を予定している．さらに，多倍長精度除算や初等関数群を実装し，ライブラリとして充実させることを検討している．最終的には，実アプリケーションに適用し，性能を評価することで GPU 用多倍長精度演算の実用性を議論したいと考えている．

謝辞　本研究の一部は，文部科学省科学研究費補助金「新学術領域研究」（課題番号 22104003）による．

参　考　文　献

1) Granlund, T.: GMP: The GNU Multiple Precision Arithmetic Library, Free Software Foundation, Inc (online), available from ⟨http://gmplib.org/⟩ (accessed 2011-10-25).
2) Bailey, D.H.: High-Precision Software Directory, LBNL Technology Transfer Department (online), available from ⟨http://crd.lbl.gov/~dhbailey/mpdist/⟩ (accessed 2011-10-25).
3) Hanrot, G., Lefevre, V., Pelissier, P., Theveny, P. and Zimmermann, P.: The GNU MPFR Library, INRIA (online), available from ⟨http://www.mpfr.org/⟩ (accessed 2011-10-25).
4) Smith, D.: Multiple Precision Computation, Loyola Marymount University (online), available from ⟨http://myweb.lmu.edu/dmsmith/FMLIB.html⟩ (accessed 2011-10-25).
5) Zhao, K. and Chu, X.: GPUMP: A Multiple-Precision Integer Library for GPUs, *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, pp.1164–1168 (2010).
6) Emmart, N. and Weems, C.C.: High Precision Integer Addition, Subtraction and Multiplication with a Graphics Processing Unit, *Parallel Processing Letters*, pp. 293–306 (2010).
7) Graça, G.D. and Defour, D.: Implementation of float-float operators on graphics hardware, *In 7th conference on Real Numbers and Computers, RNC7*, pp. 23–32 (2006).
8) Thall, A.: Extended-precision floating-point numbers for GPU computation, *ACM SIGGRAPH 2006 Research posters* (2006).
9) Lu, M., He, B. and Luo, Q.: Supporting Extended Precision on Graphics Processors, *Proc.Sixth International Workshop on Data Management on New Hardware (DaMoN 2010)* (2010).
10) NVIDIA: CUDA, NVIDIA Corporation (online), available from ⟨http://www.nvidia.com/object/cuda_home_new.html⟩ (accessed 2011-10-25).
11) Karatsuba, A. and Ofman, Y.: Multiplication of multidigit numbers on automata, *Doklady Akad.Nauk SSSR 145*, pp.293–294 (1962).
12) Knuth, D.E.: *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*, Addison-Wesley Longman Publishing Co., Inc. (1997).

© 2011 Information Processing Society of Japan