情報処理学会論文誌　コンピューティングシステム　Vol. 3　No. 2　48–56 (June 2010)

高精度総和計算と高精度内積計算のGPUのための並列アルゴリズム

鈴　木　智　博[†1]

浮動小数点数の加算と乗算の無誤差変換に基づく高精度総和計算と高精度内積計算アルゴリズムが荻田らによって開発された．これらのアルゴリズムは，通常精度の K 倍（$K \geq 2$）の精度で計算されたベクトルの総和または内積の計算結果を通常精度に丸めたものと同等の精度を持っていることが示されている．ベクトルの内積計算は数値線形代数の基本演算であるから，大規模な問題を高精度で計算することが求められることが多い．しかし，高精度総和，高精度内積計算は提案された形式のままでは逐次性が強いアルゴリズムである．これらのアルゴリズムをGPU上で効率的に実行するために，総和に対するparallel reduction演算に基づいた並列アルゴリズムを開発した．

Parallel Algorithms for Accurate Summation and Dot Product on the GPU

Tomohiro Suzuki[†1]

Accurate summation and dot product algorithms of floating point numbers using error-free transformations were proposed by Ogita et al. The accuracy of its computational results is comparable to a K-fold working precision, where $K \geq 2$. Since the dot product is an elementary operation in numerical linear algebra, highly accurate computations for large arrays are frequently required in many applications. However, accurate summation and dot product algorithms seem inherently sequential. To execute the accurate dot product (and summation) efficiently on the GPU, we develop an efficient data parallel algorithm by applying the parallel reduction operation.

†1 山梨大学大学院医学工学総合研究部
Interdisciplinary Graduate School of Medical and Engineering, University of Yamanashi

1. はじめに

高性能計算（High Performance Computing）とは大規模な計算対象を高速，高精度で処理する技術の総称であるといえる．特に計算精度は，ときに計算速度を犠牲にしても担保されるべきであり，電子計算機の黎明期から現在に至るまで研究者はつねに注意を払ってきた．

わずかな誤差の混入が計算結果に大きく反映されてしまうような悪条件問題を扱う際，通常よりも高い精度のデータ型を使用するというアプローチがある．任意精度のデータ型を扱うGMPライブラリ[3]や1つのデータ型にIEEE 754倍精度浮動小数点数を2つ使用するdouble-double演算[5]などがその例である．

これらとは別に，加算，乗算で発生する丸め誤差を補償するアルゴリズムによって高い精度の計算結果を得るアプローチがある[9]．この手法の特徴は，すべての演算が通常の浮動小数点（たとえば，IEEE 754倍精度浮動小数点）演算であり，計算結果もその浮動小数点数として与えられることである．

高精度計算は通常の計算よりも多くの演算を必要とするので，これを高速に実行するために並列化が必要となる．近年の汎用CPUはマルチコア化が進み，今後さらにコア数が増加していく傾向にある．また，最近では科学技術計算のプラットホームとして定着したGPUは汎用CPUに比べはるかに高い並列性を持つ．高い並列性を有効に活用するためには細粒度の並列アルゴリズムが必要となる．

本論文では，文献9)の高精度計算アルゴリズムをGPUなどの並列性の高いハードウェアで効率的に実行するための並列アルゴリズムを提案する．以降，2章では文献9)のアルゴリズムを解説し，3章では高精度総和，内積計算の並列アルゴリズムを提案し，4章では並列アルゴリズムのCUDAによる実装を解説する．5章では逐次アルゴリズムと並列アルゴリズムの比較，CPUとGPUの並列実装を比較する実験を行い，6章をまとめとする．

2. 無誤差変換に基づく高精度計算

$\mathbb{F}$ をIEEE 754倍精度浮動小数点数とする．また，丸めモードは偶数への最近接丸め（round-to-nearest-even）とし，$\varepsilon = 2^{-53}$ とする．$\mathbb{F}$ に対する浮動小数点演算の結果を $\mathrm{fl}(\cdot)$ で表す．このとき括弧内の演算はすべて倍精度浮動小数点演算で行われる．

a，$b \in \mathbb{F}$ の二項演算によって生ずる誤差を以下で定義する．

$$\mathrm{err}(a \circ b) = a \circ b - \mathrm{fl}(a \circ b),\ \circ \in \{+, -, \cdot, /\} \tag{1}$$

浮動小数点数 a，b の加算については次のアルゴリズムで和 $s = \mathrm{fl}(a + b)$ と誤差

$e = \mathrm{err}(a+b)$ を求めることができる[7]．

Algorithm 2.1　加算の無誤差変換

function $[s, e] = \mathtt{TwoSum}(a, b)$
　$s = \mathrm{fl}(a+b)$
　$v = \mathrm{fl}(s-a)$
　$e = \mathrm{fl}((a-(s-v))+(b-v))$
endfunction

このアルゴリズムは 6 回の浮動小数点演算回数を必要とする．ここで，a，b，s，$e \in \mathbb{F}$ について，

$$a + b = s + e,\ |e| \le \varepsilon |s| \tag{2}$$

が成り立つ．式 (2) の意味で Algorithm 2.1 を加算の無誤差変換[10] と呼ぶ．

被演算数の指数に差がある場合，浮動小数点数の加（減）算において絶対値の小さい数は桁合せのためシフトされ，その仮数部の一部（または全部）の情報を失ってしまう．この現象は情報落ちと呼ばれ丸め誤差の原因である．Algorithm 2.1 はこのような丸め誤差を浮動小数点数の加減算のみで誤差 e に抽出することができる．

浮動小数点数の乗算については次のアルゴリズムで積 $p = \mathrm{fl}(a \cdot b)$ と誤差 $e = \mathrm{err}(a \cdot b)$ を求めることができる[2]．

Algorithm 2.2　乗算の無誤差変換

function $[p, e] = \mathtt{TwoProduct}(a, b)$
　$p = \mathrm{fl}(a \cdot b)$
　$[a_1, a_2] = \mathtt{Split}(a)$
　$[b_1, b_2] = \mathtt{Split}(b)$
　$e = \mathrm{fl}((a_2 \cdot b_2 - (((p - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2)))$
endfunction

この中で $[a_1, a_2] = \mathtt{Split}(a)$ は 53 bit の仮数部を持つ浮動小数点数 a を上位 26 bit と下位 26 bit に相当する 2 つの浮動小数点数 a_1，a_2 に誤差なく分割するアルゴリズムである．FMA（Fused Multiply-Add）演算が可能なハードウェアでは Algorithm 2.2 は以下のように記述できる[9]．後述する GPU 実装においてはこちらが使用される．

Algorithm 2.3　乗算の無誤差変換

function $[p, e] = \mathtt{TwoProductFMA}(a, b)$
　$p = \mathrm{fl}(a \cdot b)$
　$e = \mathrm{FMA}(a, b, -p)$
endfunction

Algorithm 2.2 の浮動小数点演算回数が 17 回なのに対し，Algorithm 2.3 では同じ結果を得るのに必要な演算回数はわずかに 2 回である．

ベクトル $p = (p_0, p_1, \ldots, p_{n-1})^T \in \mathbb{F}^n$ に対する次のアルゴリズムを考える．

　for $i = 1$ to $n-1$ **do**
　　$[p_i, p_{i-1}] = \mathtt{TwoSum}(p_i, p_{i-1})$
　end for

このアルゴリズムで，p_{n-1} に通常の浮動小数点演算による総和 $\sum_{i=0}^{n-1} p_i$ が代入され，p_i $(i = 0, \ldots, n-2)$ には各加算における丸め誤差が代入される．つまり，これはベクトルの総和に対する無誤差変換である．この丸め誤差の総和 $\sum_{i=0}^{n-2} p_i$ を総和 p_{n-1} に加えることで計算結果を補償するのが補償付き総和（*compensated summation*[6),⋆1]）である．

Ogita らはこれを繰り返し適用する以下のアルゴリズムを開発した[9]．

Algorithm 2.4　高精度総和計算

function $\mathtt{res} = \mathtt{SumK}(n, p, K)$
　for $k = 1$ to $K-1$ **do**
　　for $i = 1$ to $n-1$ **do**
　　$[p_i, p_{i-1}] = \mathtt{TwoSum}(p_i, p_{i-1})$
　　end for
　end for
　$\mathtt{res} = \mathrm{fl}\left(\sum_{i=0}^{n-1} p_i\right)$
endfunction

Algorithm 2.4 の誤差解析のために，ベクトル総和 $\sum p_i$ の条件数 $\mathrm{Cond}(\sum p_i)$ を以下で定義[9] する．

⋆1 ただし，文献 6) のアルゴリズムでは Algorithm 2.1 ではなく，これより高速だが $|a| \ge |b|$ の制限がある変換を用いている．

定義 2.1　ベクトル総和 $\sum p_i$ の条件数.

$$\mathrm{Cond}\left(\sum p_i\right) = \frac{\sum |p_i|}{|\sum p_i|} \tag{3}$$

これを用いて Algorithm 2.4 の計算結果について次の定理が成り立つ[9].

定理 2.1　C を正の定数として，Algorithm 2.4 の相対精度は以下を満たす.

$$\frac{|\sum p_i - \mathtt{res}|}{|\sum p_i|} \leq \varepsilon + C\varepsilon^K \mathrm{Cond}\left(\sum p_i\right) \tag{4}$$

定理 2.1 は，Algorithm 2.4 の計算結果 `res` は，通常の総和計算の K 倍の精度（右辺第 2 項）で計算した結果を倍精度浮動小数点数に丸めた（右辺第 1 項）ものである，と解釈できる.

ベクトル $x = (x_0, \ldots, x_{n-1})^T \in \mathbb{F}^n$，$y = (y_0, \ldots, y_{n-1})^T \in \mathbb{F}^n$ の高精度内積計算は，Algorithm 2.2 により積を和に無誤差変換した後，長さ $2n$ のベクトル t の高精度総和計算を適用することで実現される[9].

Algorithm 2.5　高精度内積計算

function `res` $=$ `DotK`(n, x, y, K)
  $r = 0$
  for $i = 0$ to $n - 1$ **do**
    $[h, t_i] =$ `TwoProduct`(x_i, y_i)
    $[r, t_{n+i-1}] =$ `TwoSum`(r, h)
  end for
  $t_{2n-1} = r$
  `res` $=$ `SumK`$(2n, t, K - 1)$
endfunction

`DotK` についても定理 2.1 と同様な誤差解析が成り立つ．すなわち，`DotK` の計算結果は，通常の内積計算の K 倍の精度で計算した結果を倍精度浮動小数点数に丸めたものである．ただし，内積の条件数は以下で定義される.

定義 2.2　ベクトル内積 $x^T y$ の条件数.

$$\mathrm{Cond}(x^T y) = 2\frac{|x^T||y|}{|x^T y|} \tag{5}$$

大規模な科学技術計算では巨大なベクトルに対して，高い計算精度が求められることが多い．大規模な計算対象に高精度計算を適用させるため OpenMP [11] による `SumK`，`DotK` の並列化が行われた[12]．これは，適当な数に分割されたベクトル辺をそれぞれ個別の CPU において逐次アルゴリズムで処理するものであるため，必ずしも並列性が高いとはいえない.

マルチコア/メニイコアの性能を十分発揮させるためにはアルゴリズムに十分な並列性を持たせることが要求される．次章ではマルチコア CPU や GPU のデータ並列性を効率的に利用するためのアルゴリズムを提案する.

3. 並列アルゴリズム

ベクトル $p \in \mathbb{F}^n$ の並列総和計算アルゴリズムを以下に示す．ここで n は 2 のベキであることを仮定している．ただし，n が 2 のベキでない場合は padding を行う．このアルゴリズムは parallel reduction [4] として知られている．このアルゴリズムを実行することにより，ベクトル p の総和が p_0 に与えられる.

Algorithm 3.1　並列総和計算

function `DPRed` (n, p)
  for $d = 0$ to $\log_2 n - 1$ **do**
    for all $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
      $a = k;\ b = k + 2^d$
      $p_a = p_a + p_b$
    end for
  end for
endfunction

Algorithm 3.1 の加算を Algorithm 2.1 に置き換えて，次の Algorithm 3.2 を得る．このアルゴリズムは，Algorithm 3.1 同様にベクトル p の総和が p_0 に与えられるとともに，p_i（$i = 1, \ldots, n-1$）には各加算で生じた丸め誤差が代入される．Algorithm 3.2 の概略を**図 1** に示す.

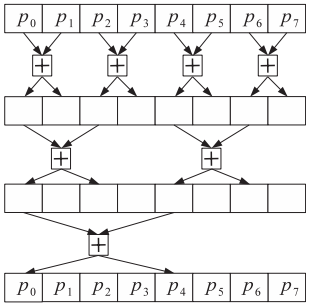

図 1　DPSum の概略（$n = 8$）
Fig. 1　Outline of DPSum ($n = 8$).

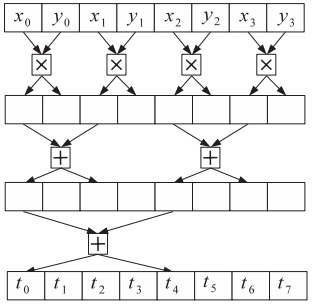

図 2　DPDot の概略（$n = 4$）
Fig. 2　Outline of DPDot ($n = 4$).

Algorithm 3.2　総和の並列無誤差変換

function DPSum (n, p)
  for $d = 0$ to $\log_2 n - 1$ **do**
    for all $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
      $a = k;\ b = k + 2^d$
      $[p_a, p_b] = \mathtt{TwoSum}(p_a, p_b)$
    end for
  end for
endfunction

Algorithm 3.1，3.2 を用いて，Algorithm 2.4 の並列アルゴリズムとして以下を提案する．このアルゴリズムにより p_0 に K 倍の精度を持つ総和が与えられる．

Algorithm 3.3　並列高精度総和計算

function DPSumK (n, p, K)
  for $k = 1$ to $K - 1$ **do**
    DPSum(n, p)
  end for
  DPRed(n, p)
endfunction

次に Algorithm 2.5 の並列アルゴリズムを得るために次のアルゴリズムを提案する．

Algorithm 3.4　内積の並列無誤差変換

function DPDot (n, x, y)
  for $i = 0$ to $n - 1$ **do**
    $t_{2i} = x_i;\ t_{2i+1} = y_i$
  end for
  for $d = 0$ to $\log_2 2n - 1$ **do**
    for all $k = 0$ to $2n - 1$ by 2^{d+1} in parallel **do**
      $a = k;\ b = k + 2^d$
      if $d = 0$ **then**
        $[t_a, t_b] = \mathtt{TwoProduct}(t_a, t_b)$
      else
        $[t_a, t_b] = \mathtt{TwoSum}(t_a, t_b)$
      end if
    end for
  end for
endfunction

Algorithm 3.4 の概略を**図 2** に示す．Algorithm 3.4 はベクトル x，$y \in \mathbb{F}^n$ をベクトル $t \in \mathbb{F}^{2n}$ とした後，要素ごとの積を Algorithm 2.2 で無誤差変換し，Algorithm 2.1 を用いながらその総和を求めることで t_0 に内積が与えられるとともに，t_i（$i = 1, \ldots, 2n-1$）には各演算で生じた丸め誤差が代入される．

Algorithm 3.1，3.2，3.4 を用いて次の並列アルゴリズムを得る．このアルゴリズムによ

り t_0 に $x^T y$ の K 倍の精度の計算結果が与えられる．

Algorithm 3.5　並列高精度内積計算

function DPDotK (n, x, y, K)
  DPDot(n, x, y)
  for $k = 1$ to $K - 2$ **do**
    DPSum$(2n, t)$
  end for
  DPRed$(2n, t)$
endfunction

4. 並列アルゴリズムの GPU 実装

4.1 GPU アーキテクチャと CUDA の概要

ここで，NVIDIA GeForce GTX 285 を例に GPU の構造を簡単に説明する．

GPU の演算コアはストリーミングプロセッサと呼ばれ，8 つのストリーミングプロセッサが 1 つのマルチプロセッサを構成している．ストリーミングプロセッサは単精度浮動小数点演算が可能である．GeForce GTX 285 ではマルチプロセッサ数は 30 ユニットである．ただし，倍精度浮動小数点演算ユニットは各マルチプロセッサに 1 つずつしか搭載されていない．浮動小数点演算は単精度，倍精度とも IEEE 754 標準に準拠している．

GPU が直接参照するメモリ（グローバルメモリ）は DDR3 1 GB をグラフィックボード上に搭載している．グラフィックボードは PCI-Ex. 2.0 でマザーボードに接続されているので，GPU でのプログラム実行は必ずバスを介したデータ転送をともなう．

各マルチプロセッサには 16 KB の共有メモリが搭載されている．同一マルチプロセッサ上で実行されるスレッドは 16 KB の共有メモリを参照できる．共有メモリはグローバルメモリに比べて 100 倍ほど高速なので，効率的な実装のためにスレッドに操作されるデータは共有メモリ上にコピーするべきである．

次に，NVIDIA の並列プログラミングモデル CUDA[8] を概説する．CUDA は C 言語を拡張した記述形式で GPU におけるスレッドグループの階層化，スレッドの同期，共有メモリ操作を可能にする．1 つのマルチプロセッサで実行される処理をブロックとし，1 ブロックで最大 512 スレッドを実行することができる．GPU で実行されるプログラムをカーネルと呼び，カーネルはブロック数，1 ブロックあたりのスレッド数，共有メモリサイズを指定されて CPU プログラムから実行される．

```
__global__ void DPSum( double* p )
{
  extern __shared__ double sm[];
  int tid = threadIdx.x;
  int i = 2*blockIdx.x * blockDim.x + threadIdx.x;

  // データコピー
  sm[tid] = p[i];
  sm[tid + blockDim.x] = p[i + blockDim.x];
  __syncthreads();

  // 共有メモリ上での総和の無誤差変換
  for (int d = blockDim.x; d > 0; d >>= 1) {
    if (tid < d)
      TwoSum(sm[tid], sm[tid + d], sm[tid], sm[tid + d]);
    __syncthreads();
  }

  // データコピー
  p[i] = sm[tid];
  p[i + blockDIm.x] = sm[tid + blockDim.x];
}
```

図 3　DPSum カーネル
Fig. 3　DPSum kernel.

4.2 CUDA によるアルゴリズムの実装

CUDA による実装は，アルゴリズムの並列部分を GPU で実行されるカーネルとして CPU プログラムから呼び出す形式となる．Algorithm 3.3，3.5 の実装においては Algorithm 3.1，3.2，3.4 をカーネルとして記述する．

文献 4) には parallel reduction カーネルの最適化について詳しい記述がある．DPRed カーネルの実装にはこの最適化カーネルをほぼそのまま適用すればよい．しかし，parallel reduction カーネルでは総和が代入される 1 ベクトル要素のみが必要なのに対し，高精度総和，内積計算では無誤差変換が繰り返し適用されるため DPSum，DPDot カーネルでは計算途中のベクトル要素すべてをそのまま保存しておく必要があり，文献 4) に記述された効果的な最適化の多くを適用できない．

ここでは DPSum カーネルの CUDA 実装を**図 3** に示し，これについて詳細に説明する．

この中で sm は共有メモリ上のベクトルであり，サイズはカーネルの呼び出し時に指定される．また threadIdx.x，blockIdx.x，blockDim.x はそれぞれスレッドの識別子，ブ

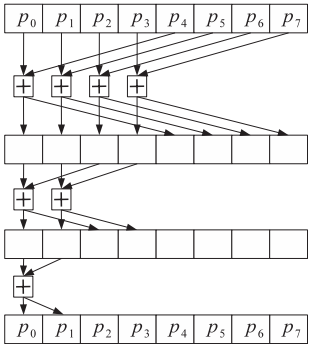

図 4　sequential addressing の概略
Fig. 4　Sequential addressing.

ロックの識別子，ブロック内のカーネル数を表す．`__syncthreads()` は同一ブロック内のスレッドを同期させる命令である．その他の CUDA プログラミングの詳細については文献 1)，8) を参照されたい．

図 3 の `DPSum` カーネルは以下の 3 つの部分に分けられる．

(a)　ベクトルデータをグローバルメモリから共有メモリにコピーする部分
(b)　無誤差変換部分
(c)　ベクトルデータをグローバルメモリから共有メモリにコピーする部分

(a) では 1 つのスレッドが `TwoSum` 演算で使用する 2 つのベクトル要素をグローバルメモリから共有メモリにコピーし，同一ブロック内のすべてのスレッドで作業が終了するのを待つ（作業の同期をとる）．

(b) では総和の無誤差変換を実行する．ベクトル要素が更新された後に次のステップが実行されるようスレッド同期を行う必要がある．

(c) では (a) の逆の操作で共有メモリ上の計算結果をグローバルメモリ上に書き戻している．

図 3 の `TwoSum` 演算が操作するベクトル要素は図 1 に示したものと異なっている．これを**図 4** に図示する．図 1 のアクセス方法を interleaved addressing といい，図 4 の方法を sequential addressing という[4]．GeForce GTX 285 の共有メモリは 32 bit ごとの 16 個のバンクにより構成されている．同一バンクに複数のスレッドがアクセスすることをバンクコンフリクトという．共有メモリへのアクセスは 16 スレッドずつ処理されるので，連続した 32 bit のデータにアクセスする場合にはバンクコンフリクトが発生しない．ベクトル要素が単精度の場合は sequential addressing を採用することでバンクコンフリクトを回避することができる．倍精度の場合は sequential addressing を行ってもバンクコンフリクトは発生してしまうが，同一バンクにアクセスするスレッド数が interleaved addressing の半分になるため (a)，(c) のデータコピー時間が短縮される．

CUDA では 1 ブロックにつき最大 512 スレッドが実行できるので 1 スレッドで `TwoSum` 演算 1 つを実行するとブロックあたり最大で要素数 1,024 のベクトルを扱うことができる．これ以上のサイズのベクトルを扱うにはカーネルを再帰的に適用すればよい．今回の実装ではブロックあたりのスレッド数を 512，再帰の段数を 1 段としたため，扱える最大要素数は $1{,}024 \times 1{,}024 = 1{,}048{,}576$ である（内積計算では 524,288）．

5. 数 値 実 験

並列アルゴリズムの有効性を検証するための実験を行う．ここでは倍精度浮動小数点数における Algorithm 3.5 と 2.5 の比較を行い，Algorithm 3.3 と 2.4 の比較は割愛する．

実験に使用した GPU は NVIDIA GeForce GTX 285，CPU は Intel Core i7（2.67 GHz）である．CUDA コンパイラは nvcc 2.2，C++ コンパイラは Intel C++ compiler 11.0 を使用した．GPU で倍精度浮動小数点数を扱うために nvcc には `-arch sm_13` オプションを使用した．Algorithm 2.1，2.2 は最適化などにより演算順序が変更されると正しく動作しなくなるため，コンパイルする際には注意が必要である．今回，C++ コンパイラオプションとして `-O3 -fp-model source` を指定した．CPU マルチスレッド実装においてはこれに加えて `-openmp` が必要となる．

並列実装については，GPU 実装とマルチコア CPU 実装とを比較するために Algorithm 3.5 の OpenMP によるマルチスレッド実装を行った．具体的には，Algorithm 3.1，3.2 の k ループと，Algorithm 3.4 では i，k ループを for 構文でスレッド分割した．Core i7 は 4 コアの CPU であり，Hyper-Threading により 8 ハードウェアスレッドを実行可能である．実験した範囲では Hyper-Threading を無効にした 4 スレッドでの実行と Hyper-Threading を有効にした 8 スレッドでの実行では前者の方が若干高速だったため，CPU での並列実装は前者による結果を掲載した．

5.1 計 算 精 度

はじめに計算精度の比較を行う．ベクトル長 $n = 8{,}192$ とし，内積の条件数の異なるデータセットを用意し，$K = 2$，4，8 のときのそれぞれのアルゴリズム，実装による内積計算

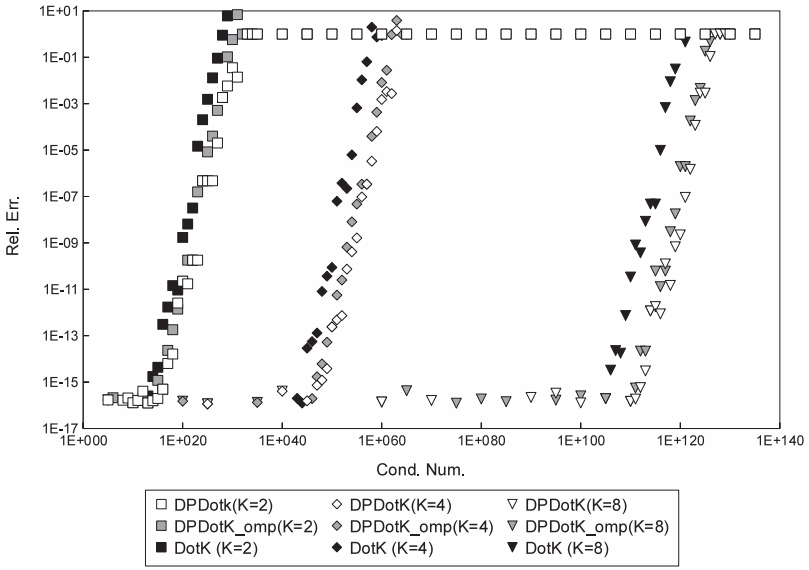

図 5　DotK と DPDotK の精度
Fig. 5　Accuracy of DotK and DPDotK.

を行った．その相対誤差を**図 5** に示す．この中で，DPDotK，DPDotK_omp はそれぞれ Algorithm 3.5 の GPU 実装と CPU マルチスレッド実装の結果を表し，DotK は Algorithm 2.5 の CPU 実装の結果を表している．

データセットは文献 12) の Algorithm 4.2（GenDot2）により生成した．GenDot2 は指定されたサイズと内積の条件数を持つベクトル x，y を生成するアルゴリズムである．このとき生成されたベクトルの内積の値は指定した条件数の逆数となる．実験では条件数を 10^c（$c = 5, \ldots, 135$）とした．

図 5 の相対誤差における真値として DotK$(n, x, y, 10)$ を用いた．また，真値と計算結果 res の差は $x_n =$ res，$y_n = -1.0$ とし DotK$(n+1, x, y, 10)$ を用いている．

図 5 の Algorithm 3.5 のプロットを見ると，$K = 2$ のとき，条件数が ε^{-1} 付近から相対誤差が増加し，ε^{-2} 付近ではまったく精度がなくなってしまっていることが分かる．$K = 4$, 8 でも同様に，$\varepsilon^{-(K-1)}$ 付近から精度が悪化し，ε^{-K} 以降では有効桁がなくなっていることが分かる．誤差解析の定理はこの結果を裏付けるものである．

また，K が大きくなるにつれ，Algorithm 3.5 の精度は Algorithm 2.5 よりも高くなる傾向が見られる．Algorithm 2.4 において内側ループ（i ループ）の計算途中に指数部の大きな数が現れると，それ以降同等の指数部を持つ数による打ち消しが起こるまで，激しい情報落ちが発生し実質的な加算は行われない．これに対して Algorithm 3.2 では外側ループ（d ループ）の初期の段階において激しい情報落ちは比較的少ない．条件数の大きなデータセットでは要素間の指数差がより大きくなるので，K の増加にともないアルゴリズムに精度差が見られるようになる．

5.2 計算時間

次に計算時間の比較を行う．$n = 512$ から 524,288 まで要素数を 2 倍しながら 11 の倍精度浮動小数点数データセットを用意し，Algorithm 3.5 と Algorithm 2.5 の計算時間を計測した．Algorithm 3.5 の GPU 実装の計算時間には，グローバルメモリとメインメモリ間のデータ送受信の時間が含まれる．内積計算 100 回の時間を測定し，その平均値を計算時間とした．結果を**図 6**（$K = 2$），**図 7**（$K = 4$），**図 8**（$K = 8$）に示す．横軸はベクトル長 n，縦軸は計算時間 (msec) とし，ともに対数目盛で表示している．

これより，ベクトル長が短かければ逐次アルゴリズムが高速だが，ベクトル長が長くなるほど，また K が大きいほど Algorithm 3.5 が優位となることが分かる．CPU マルチスレッド実装と GPU 実装の比較では $n = 131{,}072$ 以降で速度差が見られる．

今回の実験で，CPU で実行した逐次アルゴリズム（DotK）と並列アルゴリズム（DPDotK_omp）を比較して最大約 2.6 倍（$K = 8$，$n = 65{,}536$）の速度差があり，並列アルゴリズムの CPU マルチスレッド実装（DPDotK_omp）と GPU 実装（DPDotK）を比較して最大約 3.0 倍（$K = 8$，$n = 524{,}288$）の速度差があった．逐次アルゴリズム（DotK）と並列アルゴリズム GPU 実装（DPDotK）では最大約 4.4 倍（$K = 8$，$n = 524{,}288$）の速度差がある．

ベクトル長が大きい領域において CPU マルチスレッド実装の速度が低下し逐次アルゴリズムの速度に近づくのは，各スレッド（CPU コア）が扱うベクトルが大きくなりキャッシュ効率が低下するためだと考えられる．Core i7 は 1 コアごとに 256 KB の L2 キャッシュを搭載している．倍精度浮動小数点データの内積計算ではベクトル長 $n = 64{,}000$ 以降で各コアの扱うデータサイズがキャッシュサイズをオーバする．GeForce GTX 285 の（ピーク）メモリバンド幅は 159 GB/sec であるのに対し，Core i7 のメモリバンド幅は 25.6 GB/sec である．ベクトル長が大きい領域においてはこれが速度差となって現れるため GPU 実装が優位となる．

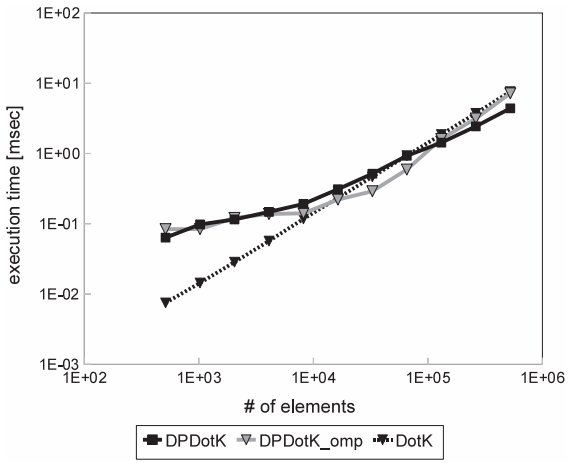

図 6　DotK と DPDotK の計算時間（$K = 2$）
Fig. 6　Execution time of DotK and DPDotK ($K = 2$).

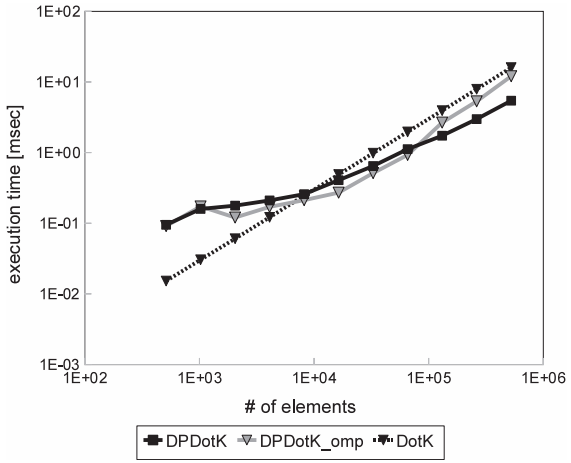

図 7　DotK と DPDotK の計算時間（$K = 4$）
Fig. 7　Execution time of DotK and DPDotK ($K = 4$).

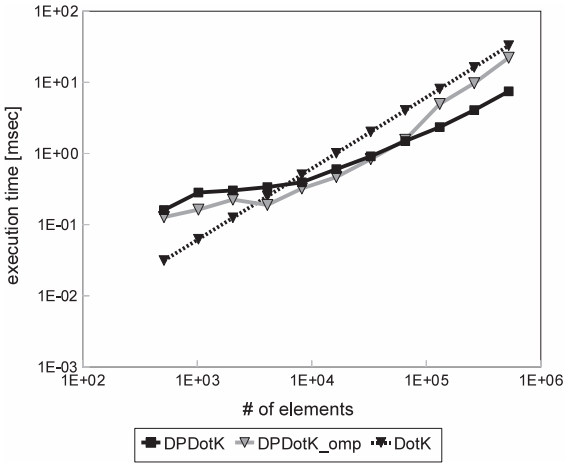

図 8　DotK と DPDotK の計算時間（$K = 8$）
Fig. 8　Execution time of DotK and DPDotK ($K = 8$).

6. おわりに

マルチコア，メニイコア向けのアプリケーションには細粒度の並列性を持つ新しいアルゴリズムの開発が求められる．本論文では無誤差変換に基づく高精度計算アルゴリズムの並列アルゴリズムを提案した．このアルゴリズムはマルチコアプロセッサや GPU のような高いスレッド並列性を持つ計算環境で有利である．

今回行った数値実験において提案手法は CPU で実行した逐次プログラムと比較して，大きなサイズのベクトルに対し，高い精度の計算結果において約 4 倍の速度向上が見られた．また，CPU マルチスレッド実装と GPU 実装で比較すると約 3 倍の速度向上が見られた．マルチコア CPU との比較における GPU の優位性はメモリバンド幅であると考えられる．

今後の課題として，総和や内積の高精度アルゴリズムを大規模行列の反復解法や行列分解などのアプリケーションに適用し，通常精度アプリケーションとの速度，精度の比較を行うことがあげられる．また，その際，グローバルメモリ–メインメモリ間のデータ転送時間を極力小さくするため，アプリケーションプログラムのフル GPU 化を意識した実装を行うことが重要である．

参 考 文 献

1) 青木尊之，額田　彰：はじめての CUDA プログラミング，工学社 (2009).
2) Dekker, T.J.: A floating-point technique for extending the available precision, *Numer. Math.*, Vol.18, pp.224–242 (1971).
3) GMP. http://gmplib.org/
4) Harris, M.: Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf
5) Hida, Y.: Library for Double-Double and Quad-Double Arithmetic. http://www.cs.berkeley.edu/~yozo/
6) Higham, N.J.: *Accuracy and Stability of Numerical Algorithms, 2nd edition*, SIAM (2002).
7) Knuth, D.E.: *The Art of Computer Programming, Vol.2: Seminumerical Algorithms*, Addison Wesley (1969).
8) NVIDIA: CUDA Programing Guide 2.2.1. http://www.nvidia.com/object/cuda_get.html
9) Ogita, T., Rump, S.M. and Oishi, S.: Accurate sum and dot product, *SIAM Journal on Scientific Computing*, Vol.26, No.2, pp.1955–1988 (2005).
10) 荻田武史：品質を落とさない数値計算法/無誤差変換と高精度計算，数学セミナー，Vol.11, pp.15–19 (2008).
11) OpenMP. http://openmp.org/
12) Yamanaka, N., Ogita, T., Rump, S.M. and Oishi, S.: A parallel algorithm for accurate dot product, *Parallel Computing*, Vol.34, pp.392–410 (2008).

(平成 21 年 9 月 28 日受付)
(平成 22 年 2 月 16 日採録)

鈴木　智博（正会員）

1991 年山梨大学大学院工学研究科修了．同年山梨大学工学部助手．2006 年山梨大学総合情報処理センター助教授．2009 年山梨大学大学院医学工学総合研究部准教授．博士（学術）．高性能計算，精度保証付き数値計算に興味を持つ．日本応用数理学会会員．