Mathematica

～統合的な研究開発環境の特徴とCUDAの活用～

my recommendations on research & development tools.

(正会員) 久下 哲郎†

キーワード: Mathematica, 数式処理, アルゴリズム記述, 画像処理, GPU, CUDA

1. ま え が き

Mathematicaは、1988年にWolfram Research社 (<http://www.wolfram.com/>) が発売した数式処理システムで、かれこれ24年の歴史を持っており、科学・技術分野での研究開発の統合的なツール・環境と言える。数式処理に留まらず、抽象的なアルゴリズム記述、任意桁精度の数値計算、数値データ処理や可視化などの機能を備える。さらに膨大な数の数学関数やアルゴリズムを網羅した科学技術の知識データベースでもある。

筆者は20年来、研究開発のツールとしてMathematicaを使用してきたので、その特徴の一端を紹介する。また、最新のVersion 8で新しく装備された特徴の一つであるGPU (Graphics Processing Unit) を駆動して、高速に並列演算を行う機能についても解説する。

2. 試行錯誤のための研究ツールとしての Mathematica

一般に、新しいアルゴリズムやデータ処理手順を開発する場合、机上(頭の中、紙の上など)で処理の手順を推敲し構築する。この段階は抽象的一般的な思考過程である。ある程度内容が固まったところで、C言語など機械(CPU)で計算実行可能な表現形態にアルゴリズムを記述し直し、実際に機械で実行させてみる。一般にこの手続きを「アルゴリズムの実装」と呼ぶ。計算実行により、その結果が思わしくない場合や、性能や機能で改良すべき点が見つかり、再び最初のフェイズに戻り、アルゴリズムを修正し、再び実装をし直すという作業を繰り返す。

この一連の手続きは、抽象的一般的にアルゴリズムを記述する過程と、実装のために現実的な制約環境(プログラム言語やマシン環境)の上でアルゴリズムを記述・再構成する過程を交互に繰り返している。Mathematicaは、もし、抽象的一般的で表現自由度が高い前者のスタイルのままで、現実世界でのアルゴリズムの実装まで行うことができれば、という研究開発者の夢を実現するためのツールと言える。

本稿では、アルゴリズムの抽象的一般的な記述から、具

体的なデータ処理手順とその実装までの一連の流れを例示し、Mathematicaの機能と特徴を紹介する。

Mathematicaのプログラムファイルは、拡張子".nb"を持ち"notebook"と呼ばれる。以下、幾つかのnotebookを示しながらMathematicaの機能を解説する。

3. 記号処理により方程式を解くSolve関数

Mathematicaでは、式を入力し(In [x]), シフト+リターンキーを押すことにより、その回答がOut [x] として表示される。まず、記号的数式処理の簡単な例として、2次方程式の一般解を求める(図1)。In [1] で定義したn次多項式f(x, n)について、n=2としてSolve関数で2次方程式を解く(In [3])。よく知られた一般解が実行結果Out [3] に得られる。次数がn=3, 4の場合も、In [4], In [5] により同様に一般解が得られる(紙面の都合で結果は省略)。

次に、より抽象度が高い数式処理の例として、方程式のすべての根が等しくなる条件を求める問題を考える。方程式f(x)=0について、この条件を命題(#)とすると

$$(\#) \quad f(x)=0 \wedge f(y)=0 \Rightarrow x=y$$

となる。具体例として4次方程式f(x)が(#)を満たす条件を求める。

In [8] は、関数Solveにより、命題(#)を満たすf(x)の係数{a, b, c, d}の条件を求める。実行結果がOut [9] に得られている。出力は二つの解のリストである。関数Solveによる解は置換えルール(シンボル→置換え表現)の形式となっている。二つの解は前者の一般解と後者の自明解(a=b=c=d=0)である。

次に、In [9] により、一般解の置換えルールを用いて、f(x)の表現を書換える。"% "はMathematicaの直前の出力結果(この場合はOut [2] のリスト)を指定する。"// First"はリストの最初の項目を取出す関数操作である。したがって("(%// First)"は上記の一般解を示す。"(表現) /. (ルール)"は、左辺の(表現)に右辺のルールを適用して置換える操作である。

以上により、In [9] はIn [7] で定義したf(x)のシンボル{b, c, d}を一般解のルールで置換える。この結果Out [9] を得る。さらに、In [10] でこの結果を関数Factorで因数分解して、結果Out [10] を得る。すなわち4次方程式f(x)=0が4重根を持つので、Out [9] が命題(#)を満たすことが判る。

† 財団法人NHKエンジニアリングサービス

"Mathematica" by Tetsuro Kuge (NHK Engineering Service, Tokyo)

■ n次方程式の一般解を求める

```
In[1]:= f(x_, n_) := Sum[a_k x^k, {k, 0, n}]
```

```
In[2]:= f(x, 2) == 0
```

```
Out[2]:= a_0 + x a_1 + x^2 a_2 == 0
```

```
In[3]:= Solve[f(x, 2) == 0, x]
```

```
Out[3]:= {{x -> (-a_1 - Sqrt[a_1^2 - 4 a_0 a_2]) / (2 a_2)}, {x -> (-a_1 + Sqrt[a_1^2 - 4 a_0 a_2]) / (2 a_2)}}
```

```
In[4]:= Solve[f(x, 3) == 0, x]
```

```
In[5]:= Solve[f(x, 4) == 0, x]
```

(* 五次以上の方程式に対して代数的な根の公式は存在しない *)

■ 四次方程式のすべての根が等しくなる条件を求める

```
In[6]:= f // ClearAll;
```

```
In[7]:= f(x_) := a x^3 + b x^2 + c x + d + x^4;
```

```
In[8]:= Solve[forall {x,y} (f(x) == 0 & f(y) == 0 => x == y), {a, b, c, d}]
```

```
Out[8]:= {{b -> (3 a^2) / 8, c -> (a^3) / 16, d -> (a^4) / 256}, {a -> 0, b -> 0, c -> 0, d -> 0}}
```

```
In[9]:= f(x) /. (% // First)
```

```
Out[9]:= (a^4) / 256 + (a^3 x) / 16 + (3 a^2 x^2) / 8 + a x^3 + x^4
```

```
In[10]:= % // Factor
```

```
Out[10]:= (1) / 256 (a + 4 x)^4
```

図1 記号処理により方程式を解くSolve関数

4. 関数の不定積分とグラフ表示

図2は、2変数関数をIn[1]で定義し、2次元不定積分する例(In[2])である。元の関数と積分関数(Out[2])を並べて3次元表示する(In[3])。連続関数が積分により不連続となる様子が容易に目視できる。Mathematicaはプログラミング言語であるが、このように慣用的な数学表記で記述することもできる。

5. 画像の双1次補間による拡大

Mathematicaによる画像の数値演算の例として、双1次補間で画像を拡大するアルゴリズムを実装する。さらに本アルゴリズムをCUDAによりGPU演算に実装して、演算速度が格段に向上することを示す(図3)。なお本稿では、画像を2次元行列で表し、座標{i, j}でi行j列目の画素を指定する。

双1次補間は、4個の正方座標点と4個の画素値が与えられたとき、水平と垂直方向の二重の線形補間により、内点における画素値を補間生成する。In[1]にこれを実行する関数bilinear2Dを定義する。4点座標

(#1) {{0, 0}, {0, 1}}, {{1, 0}, {1, 1}}
に対して、与えられた4画素値を

(#2) {{f00, f01}, {f10, f11}}

とすると、内点{i, j}における画素値を計算する。In[3]、Out[3]は、bilinear2Dによる画素値の3次元分布を表示している。In[1]のbilinear2Dはベクトルと行列の内積計算で定義しているので、これを展開した双1次線形式Out[4]で新たに定義しなおしたbilinear2DがIn[5]である。In[1]もIn[5]も同じ結果を与えるが、計算速度はIn[5]が早い。

次に、画像全体(2次元行列)を補間拡大する関数imageExpandBilinear2DをIn[7]で定義する。これはbilinear2Dを駆動して、入力画像aInを、垂直/水平の拡大係数{m, n}で拡大した画像を出力する。局所変数pは、4点正方座標(#1)に対して、{m, n}拡大のためのm*n個の座標集合を、外積関数Outerにより生成したものである。Out[9]に、{m, n} = {3, 4}の場合の座標集合の例を示す。局所変数aは、入力画像aInに右枠一列と下枠一行を拡張した行列である。"aIn // Dimensions"は行列aInの次元を示す。これを{ma, na}とすると、aInから4点正方座標の4画素Aを取出し、

```

In[1]:= f(x_, y_) := 
$$\frac{1}{(\sin(x) + 2)(2 - \sin(y))}$$

In[2]:= 
$$\int \int f(x, y) dy dx$$

Out[2]:= 
$$-\frac{4}{3} \operatorname{ArcTan}\left[\frac{1 + 2 \operatorname{Tan}\left[\frac{x}{2}\right]}{\sqrt{3}}\right] \operatorname{ArcTan}\left[\frac{1 - 2 \operatorname{Tan}\left[\frac{y}{2}\right]}{\sqrt{3}}\right]$$

In[3]:= Grid[{Plot3D[f(x, y), {x, 0, 2π}, {y, 0, 2π}], Plot3D[ $\int \int f(x, y) dy dx$  // Evaluate, {x, 0, 2π}, {y, 0, 2π}]}],
ItemSize → Medium, Frame → All]

```

図2 関数の不定積分とグラフ表示(数学表記の慣用形式)

■ bilinear2D

4点の座標 $\begin{pmatrix} \{0, 0\} & \{0, 1\} \\ \{1, 0\} & \{1, 1\} \end{pmatrix}$ の画素値 $\begin{pmatrix} f00 & f01 \\ f10 & f11 \end{pmatrix}$ が与えられたとき、
水平と垂直方向の二重の線形補間により、途中の位置座標 $\{i, j\}$ の画素値を補間計算する。

```

In[1]:= bilinear2D // ClearAll;
bilinear2D[{i_, j_},  $\begin{pmatrix} f00_ & f01_ \\ f10_ & f11_ \end{pmatrix}$ ] := (1 - i) .  $\begin{pmatrix} f00 & f01 \\ f10 & f11 \end{pmatrix}$  .  $\begin{pmatrix} 1 - j \\ j \end{pmatrix}$  // First // First;
In[3]:= Plot3D[bilinear2D[{i, j},  $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ ], {i, 0, 1},
{j, 0, 1}, AxesLabel → Automatic, BoxRatios → {1, 1, 0.7}]

```

■ ベクトルと行列の積計算を展開

```

In[4]:= Block[{i, j, f00, f01, f10, f11},
bilinear2D[{i, j},  $\begin{pmatrix} f00 & f01 \\ f10 & f11 \end{pmatrix}$ ] // Expand // FullSimplify]
Out[4]:= f00 (-1 + i) (-1 + j) + f01 j + i (f10 - (f01 + f10 - f11) j)

```

図3 双1次補間 (bilinear interpolation)

■ 展開式によりbilinear2Dを再定義

```
In[5]:= bilinear2D // ClearAll;
bilinear2D[{i_, j_}, {f00_ f01_
f10_ f11_}] :=
f00 (-1 + i) (-1 + j) + f01 j + i (f10 - (f01 + f10 - f11) j);
```

■ imageExpandBilinear2D

bilinear2D [{i_, j_}, {f00_ f01_ f10_ f11_}] を駆動して、画像全体 (2次元行列) を補間拡大する。

```
In[7]:= imageExpandBilinear2D // ClearAll;
imageExpandBilinear2D[aIn_, {m_, n_}] := Block[{
a = {aIn, aIn[[;; -1, -1 ;; -1]]} // ArrayFlatten,
p = Outer[{##} &, Range[0, 1 - 1/m, 1/m], Range[0, 1 - 1/n, 1/n]] // N,
Array[Block[{A = {a[[#1, #2]] a[[#1, #2 + 1]]
a[[#1 + 1, #2]] a[[#1 + 1, #2 + 1]]}
, Map[bilinear2D[#, A] &, p, {2}]
], aIn // Dimensions] // ArrayFlatten
];
```

■ 3x3拡大の場合の座標点

```
In[9]:= Map[List,
Block[{m = 3, n = 4}, Outer[{##} &, Range[0, 1 - 1/m, 1/m], Range[0, 1 - 1/n, 1/n]] // N,
{2}] // MatrixForm
Out[9]//MatrixForm=
( 0. 0. ) ( 0. 0.25 ) ( 0. 0.5 ) ( 0. 0.75 )
( 0.333333 0. ) ( 0.333333 0.25 ) ( 0.333333 0.5 ) ( 0.333333 0.75 )
( 0.666667 0. ) ( 0.666667 0.25 ) ( 0.666667 0.5 ) ( 0.666667 0.75 )
```

■ {3,3}拡大の実例 (CPU実行)

```
In[10]:= Block[{mono = // ImageData, monoEx, magFactor = {3, 3}},
(monoEx = imageExpandBilinear2D[mono, magFactor];) // AbsoluteTiming // Print;
Dimensions /@ {mono, monoEx} // Print;
Image[#, ImageSize -> {# // Dimensions // Reverse} / 2] & /@ {mono, monoEx} // Row
]
{2.989844, Null}
{{164, 223}, {492, 669}}
```

図3(続き) 双1次補間 (bilinear interpolation)

bilinear2Dを適用する動作を $m \times n$ 回繰り返すことで、次元 $\{m \times m, n \times n\}$ の拡大画像を生成する。ここで、関数Array, Mapは、与えられた関数を駆動してデータ集合(この例では2次元配列)を生成する。この二つの"関数の駆動関数"は、Mathematicaで多次元データ集合を取り扱う際の特徴なので、少し詳しく説明する。

Array関数は、一般に n 個の添字 $i_1, \dots, i_n$ を引数とする関数 $f[i_1, \dots, i_n]$ を駆動して、 n 次元の多次元配列のデータを生成する。C言語で記述すると、まず n 次元配列領域を確保しておき、繰返し構文forを n 重にネストしたループで関数 $f(i_1, \dots, i_n)$ の値を配列要素に代入することになる。以下に2次元の例を示す：

```
float A[m1][m2];
for(int i1=1; i1 <= m1; i1++)
    for(int i2=1; i2 <= m2; i2++)
        A[i1-1][i2-1] = f(i1, i2);
```

Mathematicaでは、Array関数を用いて次の1文で記述できる：

```
A = Array[f[#1, #2] &, {m1, m2}];
```

#1, #2は各々 $\{1, \dots, m1\}, \{1, \dots, m2\}$ の範囲を動き、関数値 $f[\#1, \#2]$ を要素とする長さ $m1 \times m2$ にネストしたリストの配列、すなわち、長さ $m2$ のリストが $m1$ 個並んだリスト(2次元配列)を生成する。このようにArray関数はデータそのものを生成するので、事前に領域Aを確保する必要はない。なお、Mathematicaでも、上記のC言語のネストしたfor構文と同様な繰返し処理を行うDo関数

`Do[(処理), {i, imin, imax}, {j, jmin, jmax}, ...]`も用意されているが、Array関数の方が圧倒的に処理速度が早い。

ここで注意しておくことは、C言語では配列要素のインデックスは0-originであるが、Mathematicaは1-originである。この例で配列Aの先頭要素は、C言語では $A[0][0]$ であるが、Mathematicaでは $A[1, 1]$ である。

Map関数は、多次元配列データ集合の各要素に関数を適

用して、出力として同じ次元のデータ集合を生成する：

`Map[(関数), (データ集合), levelspec]`
`levspec`は、データ集合のリストのネストレベル、すなわち多次元配列の次元を指定する。本稿では2次元配列を取り扱うので`levspec=2`である。

これらの関数の詳細は、Mathematicaのオンラインマニュアル(と内容の以下URL)を参照。

<http://reference.wolfram.com/Mathematica/ref/Array.ja.html>

<http://reference.wolfram.com/Mathematica/ref/Map.ja.html>

In[7]で定義したimageExpandBilinear2D[]で、Array関数とMap関数の使い方を見てみよう。入力画像の拡張配列a上のある一点 $a[\#1, \#2]$ を左上とする4点正方形座標の4画素A, {m, n}拡大のための $m \times n$ 個の相対座標集合pに対して、

```
Map[bilinear2D, #, A] &, p, {2}]
```

により、座標集合pの各座標点での補間値 $m \times n$ 個を求める。さらに、この処理全体をArray関数により、入力画像aの次元 $\{m, n\}$ で駆動することで、全体の補間画像を得ている。ただし、Array関数の出力は、次元 $\{m, n\}$ の小行列が次元 $\{m, n\}$ で並んだ入れ子の行列なので、これを次元 $\{m \times m, n \times n\}$ の単純な行列にフラット化する操作が"// ArrayFlatten"である。

Mathematicaの変数は、整数、実数などの特定の型宣言や、スカラー、ベクトル、行列などの構造宣言も必要ない。メモリーの許す限り任意桁長の数を表現できるなど、柔軟性が高い。これは変数の内部表現がリスト構造のためである(図4)。さらにimageExpandBilinear2Dの入力画像の行列は整数、実数、有理数、複素数のスカラー値でもベクトルでも構わない。Bilinear2Dにおける1次線形の積和演算がこれらの体(環)の違いによらず適用できるからである。

In[10], Out[10]にimageExpandBilinear2Dにより画

図4 Mathematica\08の内部データ表現

像を拡大率 {3, 3} で拡大して表示する例を示す。入力画像は他のアプリケーションからコピー・ペーストした画像オブジェクト(薔薇のモノクロ画像)を貼付けている。画像ファイルを開関数Importにより読み込むこともできる。"// ImageData"で左辺の画像オブジェクトから画素データを行列形式で取り出し、入力画像の局所変数monoに代入する。imageExpandBilinear2Dで拡大した画像がmonoExである。関数Dimensionsにより、入力画像と拡大画像の次元を調べると、各々 {164, 223}, {492, 669} となり、{3, 3} に拡大されていることが判る。"// AbsoluteTiming"は左辺を評価したときの経過時間(秒)を出力する。この例では約3秒である。

6. 双1次補間 (bilinear interpolation) の CUDA実装

前章では、Mathematicaの関数による画像の双1次補間拡大を示した。本章では、アルゴリズムの中心部分をCUDAによるGPUの並列高速演算に置き換えることで、演算の高速化を計る。CUDA (Compute Unified Device Architecture) は、NVIDIA社 (<http://www.nvidia.com>) のGPU (Graphics Processing Unit) を駆動するためのC言語ベースのソフトウェア統合開発環境で、NVIDIA社のサイト (<http://developer.nvidia.com/category/zone/cuda-zone>) から無償でダウンロードできる。コン

■ CUDAの初期化

```
In[1]:= Needs["CUDALink`"]
In[2]:= CUDAQ[]
Out[2]:= True
```

■ bilinear2DCUDA GPU上で双一次補間を高速実行する関数

```
In[3]:= "bilinear2D.cu" // FilePrint;

# define Loop(i,n) for(int i=0;i<(n);i++)
# define _ij2k(i,j,n)((i)*(n)+(j))
# define _k2ij(k,i,j,n) (i=(int)floorf((k)/(n)), j=(k)%(n))
//
__device__ float bilinear2D(float2 p, float f[2][2]){
// f00*(-1+i)*(-1+j) + f01*j + i*(f10 - (f01 + f10 - f11)*j)
return f[0][0]*(-1. + p.x)*(-1. + p.y) + f[0][1]*p.y
+ p.x*(f[1][0] - (f[0][1] + f[1][0] - f[1][1])*p.y);
}
//
__global__ void bilinear2D(float* X, float *A1, int2 *dimA1p, float2 *iyjxPos, int2* magp){
int idxA = threadIdx.x + blockIdx.x*blockDim.x; //linear index
int2 dimA1 = *dimA1p;
int2 mag = *magp;
int2 dimX = { (dimA1.x-1)*mag.x, (dimA1.y-1)*mag.y };
if( idxA < dimA1.x * dimA1.y ){
int i,j;
_k2ij(idxA, i, j, dimA1.y); // convert linear index k to {i,j}
if( i < (dimA1.x-1) && j < (dimA1.y-1) ){
float v[2][2] = { { A1[_ij2k(i, j, dimA1.y)], A1[_ij2k(i, j+1, dimA1.y)] },
{ A1[_ij2k(i+1, j, dimA1.y)], A1[_ij2k(i+1, j+1, dimA1.y)] } };
Loop(k, mag.y)
Loop(L, mag.y)
X[_ij2k(i*mag.x + k, j*mag.y + L, dimX.y)]
= bilinear2D(iyjxPos[_ij2k(k, L, mag.y)], v);
} // endif( i < (dimA1.x-1) && j < (dimA1.y-1) )
} // endif( idxA < dimA1.x * dimA1.y )
}

In[4]:= bilinear2DCUDA // ClearAll;
Block[{srcf = "bilinear2D.cu", nblk = 512, fname = "bilinear2D", cu},
cu = Import[srcf, "Text"];
bilinear2DCUDA = CUDAFunctionLoad[cu, fname, {
{"Float"}(* X *)
, {"Float"}(* A1 *)
, {"Integer32[2]"}(* A1dim[2] *)
, {"Float[2]"}(* iyjxPos[][2] *)
, {"Integer32[2]"}(* magp[2] *)
}
, nblk
];
];
```

図5 双1次補間 (bilinear interpolation) のCUDA実装

■ imageExpandBilinear2DbyCUDA

GPU上で双一次補間を高速実行する関数bilinear2DCUDAを駆動して、画像全体（2次元行列）を補間拡大

```
In[6]:= imageExpandBilinear2DbyCUDA // ClearAll;
imageExpandBilinear2DbyCUDA[a_, {m_, n_}] :=
Block[{FlattenLevelOne, lst2mat, a1, p, adim, result},
  lst2mat[r_, {mm_, nn_}] := Array[Take[r, {(#1 - 1) nn + 1, #1 nn}] &, mm];
  FlattenLevelOne = Flatten[#, 1] &;
  p = Outer[{##} &, Range[0, 1 -  $\frac{1}{m}$ ,  $\frac{1}{m}$ ], Range[0, 1 -  $\frac{1}{n}$ ,  $\frac{1}{n}$ ]] // N;
  a1 =  $\left( \begin{array}{cc} a & a[[;; -1, -1 ;; -1]] \\ a[[-1 ;; -1, ;; -1]] & a[[-1 ;; -1, -1 ;; -1]] \end{array} \right)$  // ArrayFlatten;
  adim = a // Dimensions;
  result = bilinear2DCUDA[
    ConstantArray[0., Times@@{adim{m, n}}],
    a1 // Flatten // N
    , a1 // Dimensions
    , p // FlattenLevelOne
    , {m, n}][[1]];
  result = lst2mat[result, adim{m, n}];
  result
];
```

■ {3,3}拡大の実例 (CUDA実行)

```
In[8]:= Block[{mono = // ImageData, monoEx, magFactor = {3, 3}},
  {monoEx = imageExpandBilinear2DbyCUDA[mono, magFactor];} // AbsoluteTiming // Print;
  Dimensions /@ {mono, monoEx} // Print;
  Image[#, ImageSize -> {# // Dimensions // Reverse} / 2] & /@ {mono, monoEx} // Row
]
{0.021137, Null}
{{164, 223}, {492, 669}}
```

Out[8]=

図5(続き) 双一次補間 (bilinear interpolation) のCUDA実装

パイラnvcc (NVIDIA CUDA Compiler), ライブラリー, ドライバ, サンプルコード (SDK) などで構成されている。最新のMathematica Version8は, CUDA環境とのAPIを装備しており, notebookからGPUの並列演算機能を直接利用できるようになった(図5)。

In[1]で, CUDAを利用するためのMathematicaのライブラリーを読み込む。In[2]のCUDAQ[]により, PC

に搭載されたGPUデバイスが, MathematicaでCUDA利用できるかどうかをチェックする。In[3]は, GPU上で実行するプログラム(kernel code)のソースファイル"bilinear2D.cu"を表示している。CUDA言語はこのように通常のC/C++言語とほとんど同一の文法で記述する。

__global__ void bilinear2D()は, 通常のC/C++言語環境(すなわちCPU側)から呼び出されるglobal関数である。

表1 kernel codeをコンパイルするCUDAFunctionLoad関数

CUDAFunctionLoad [src, fun, argtypes, blockdim]		Mathematica側のCUDA-API関数 bilinear2DCUDA
src	kernel codeのテキスト	Import["bilinear2D.cu": "Text"]
fun	global関数名	"bilinear2D"
argtypes	global関数の変数タイプ	{["Float"], ["Float"], ["Integer32[2]"], ["Float[2]"], ["Integer32[2]"]}
blockdim	GPUのblockの次元	512

一方、__device__ float _bilinear2D()は、device関数(kernel codeの中だけで通用する)と呼ばれ、bilinear2D()から呼び出される。_bilinear2D()は、先のMathematica関数bilinear2D[]を単純にCUDA形式に書換えたもので、bilinear2D()は、これを入力画像の1画素当り拡大比率の個数分(mag.x*mag.y)呼び出すことにより、画像の双1次補間拡大を実行する。

In [4] で定義したbilinear2DCUDAは、kernel codeのbilinear2D()をMathematicaから呼び出すためのCUDA-API関数である。関数CUDAFunctionLoad[]は、kernel codeのnvccによるコンパイルとMathematica環境への組込みを実行するもので、表1にその引数を示す。

GPUのプログラミングモデルは、SPMD (Single Program Multiple Data)であり、一種類のプログラムで多量のデータを扱う。本稿の画像処理では、1画素毎に「仮想的なCPU」を一つ割当て、すべてのCPUが同一のkernel codeを実行して画像全体を処理する。仮想的なCPUをCUDAで"thread"と呼ぶ。ハイエンドGPUでは、最大65,535×65,535×65,535個の3次元構造のthreadが使える。threadは、さらに3次元構造の"block"で分割される階層構造である。kernel codeは、実行過程で各threadに割当てられる固有のthread番号を用いて個々のデータにアクセスする。

bilinear2D()では、GPU内のメモリ("global memory"と呼ぶ)上に確保された画像配列A1のサイズdimA1をthread総数とし、idxA1に与えられるthread番号をA1の画素アドレス(補間拡大される出力画像の画素アドレス)としている。なおthreadとblockは3次元ではなく1次元の線形アドレスとしている。global関数bilinear2D()の引数float2, int2は、CUDAの"ベクトル型"変数で以下で定義される：

```
typedef {float x, y;} float2;
```

```
typedef {int x, y;} int2;
```

このモデルでは、kernel code(すなわち bilinear2D())の1回の同時並列実行で、瞬時にして画像全体の処理が終了することになる。ところが実際は、GPU内の物理的CPUは、thread総数より少ないため(ハイエンドGPUで500個程度)、物理CPU総数が現実同時に実行できるthread数の上限となる。上述のblockが同時実行するthreadの分割単位であり、

ここではそのサイズblockdimは512である。すなわち、最大512個の物理CPUが同時並列に動作する。

In [6] で、GPU上で双1次補間を高速実行するCUDA-API関数bilinear2DCUDAを駆動して、画像全体(2次元行列)を補間拡大するMathematica関数imageExpandBilinear2DbyCUDAを定義する。前章のimageExpandBilinear2Dの置き換えである。GPUのthreadを1次元としたため、行列(2次元配列)a1とpを"/Flatten"により1次元配列に変換して、bilinear2DCUDAに渡している。さらに、局所関数で定義したlst2matにより、戻値resultを2次元配列に折り戻している。

In [8] に、前章と同じ画像をGPUにより拡大率 {3, 3} で拡大して表示する例を示す。実行時間が0.02秒で、前章のCPU処理に比べて約300倍に処理速度が向上している。

7. む す び

本稿で、代数方程式の一般解や命題を証明する例や、関数の不定積分とそのグラフ表示など、Mathematicaの持つ記号的・非数値的な特徴を紹介した。画像の双1次補間による拡大を、一般的な体(環)を元とする2次元配列に対するアルゴリズムとしてMathematicaで簡潔に構成できることを示し、このアルゴリズムを、並列高速演算ハードウェアGPU上にCUDA言語で実装し、高速に演算実行する実例を示した。

このように、抽象的一般的なアルゴリズム構築から、機械(CPU)で計算実行可能な形態に実装するまでの一連の過程を、Mathematicaのフレームワークの中で実現できる。Mathematicaで開発したkernel codeは、C言語環境の通常のCUDA環境でもそのまま利用できるため、MathematicaはCUDAのkernel codeを開発・デバッグする環境としての利用価値が高い。紹介したMathematica Version8のnotebookとCUDAを実行したホスト環境は、MacOS X 10.6.8/CUDA 4.1とMacOS X 10.7.3/CUDA 4.2である。WindowsやLinuxなどの他環境でも同様に実行可能と考えられる。

本稿に関心を持つ読者が、Mathematicaを活用して研究成果の発展に活用されることを期待する。

(2012年6月27日受付)

くろだ 哲郎 1984年より、NHK放送技術研究所にて、HDTV用VTRの信号処理・画像符号化、立体HDTV用画像符号化の研究開発に従事。1995年より、NHK放送技術局、同報道局、同マルチメディア局にて、選挙報道・番組制作の各種情報処理システムの開発・運用、インターネットの放送応用の企画・開発に従事。2000年より、NHK放送技術研究所にて、HDTVのJPEG2000による番組制作用符号化の開発に従事。2008年より、(財)NHKエンジニアリングサービスにて、4K高精細映像のJPEG2000符号化、視覚モデルによる映像の高精細化(Super Resolution)の研究・開発に従事し、現在に至る。SC29/WG1国内委員、画像電子学会編集委員、1995年、映像情報メディア学会技術振興賞受賞、工学博士、正会員。