チュートリアル

ベクトルや行列に関する演算、連立一次方程式や固有値問題などの解をコンピュータ上で高速に得ることは、多くの数値計算の分野で大変重要なテーマとなっています。近年では、これらの線形代数演算をコンピュータのアーキテクチャに特化し、より高速に行うことのできるライブラリが開発されてきています。今回は線形代数演算ライブラリBLAS, LAPACKに関するチュートリアル（全2回シリーズ）の第2回目として、GPU上での利用を中心に（独）理化学研究所の中田真秀氏に解説していただきます。

BLAS, LAPACKチュートリアル パート2（GPU編）

中田 真秀

1 はじめに

前回に引き続き、今回は高速かつ安価なので注目されているGPU（Graphics Processing Unit）上でのBLAS, LAPACKについて解説する。

GPUはそもそもはグラフィックス・プロセッサで、汎用計算にも使えるGPUがNVIDIA社およびAMD社から出ている。AMD社のRADEON HDシリーズの方が演算速度は高速だが、ソフトウェア、ライブラリ、開発環境やユーザー数はNVIDIA社の製品向けのものの方が圧倒的に充実している。さらにNVIDIA社はグラフィックス向けではなく、汎用計算の利用を主眼においた製品も提供している。

このような背景から今回はハイパフォーマンスコンピューティング向けのGPU製品のNVIDIA社C2050上でBLAS, LAPACKを利用する方法を述べる。特にcuBLAS, MAGMAに焦点を当てた。CULAは紙面の都合で、取り上げなかった。

前回パート1としてBLAS, LAPACKについての簡単な使い方とプログラミングを示した。ホームディレクトリの位置や改行などは前回のチュートリアルに示したものを用いるので、見逃した方は、計算工学のホームページからpdfで誰でも閲覧可能なのでこれをご覧頂きたい[1]。

筆者紹介

なかた まほ
理化学研究所 情報基盤センター 協力研究員。博士（工学）。専門は量子化学（縮約密度行列の変分法）、最適化（半正定値計画）、ハイパフォーマンスコンピューティング（線形代数）。BLAS, LAPACKの高精度版の構築と応用に興味を持っている。
http://accc.riken.jp/maho/

2 GPU, GPGPUとは？

近年、CPUの速度はどんどん頭打ちになってきた。複雑な処理を高速に処理するため、回路は大変複雑になり、コストや物理的な限界に近づいてきたからだ。そんな中で、もともとはパソコン向けのグラフィックス・プロセッサとして、ゲーム、CADやCAEなどで利用されてきた、Graphics Processing Unit（GPU）が脚光を浴びてきた。一般用途に使うこと（General-purpose computing on graphics processing units; GPGPU）も可能なこと、さらに安価な割にCPUと比べて数倍から10数倍高速に処理を行うことができるからだ。限られた単純な処理しか行えないがその代わり高速になっている。

コンピュータの浮動小数点演算の最大速度はFlops単位（FLoting Point per Second; 一秒間に何回浮動小数点演算をできるか）で測られる。大まかな目安として最大理論性能値を比較しよう。最新のIntel Core i7 2600Kは100GFlops程度、AMD社のRADEON HD6990は1.37TFlops, NVIDIA社 のC2050は515GFlopsと、GPUはCPUより5倍から10倍速い。

3 コンピュータの基本原理と高速な計算

何も考えずに単純にGPUを用いただけではプログラムは高速にはならない。むしろ遅くなることさえある。GPUでBLAS, LAPACKを使い、高速化の恩恵を受けるにはコンピュータの一般的な仕組みを大まかに理解し、高速にするにはどうすればよいか、をある程度意識しなければならない。この考えは様々な局面に通じる。例えばアルゴリズムが定まったプログラムを高速化するときどうしたらいいかというのも、基本的には同じ考えである。

(1) コンピュータの基本原理

コンピュータで高速な計算を行うにはどうすればよ

いだろうか。CPUだけが高速であれば良いのであろうか。そうではない。まずは基本原理から見よう。現在のコンピュータは、メモリからデータを読み出し、レジスタ経由で論理演算装置で処理するというフォン・ノイマン型のものがほぼすべてである。この構成は、演算論理装置、制御装置、メモリ、入出力と、これらを接続する情報経路（バス）となっている（図1）。

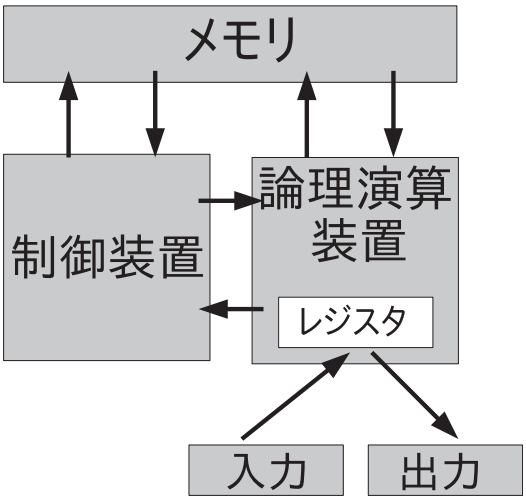

図1　フォン・ノイマン型コンピュータの概念図。いくつかの基本となる部分から成る。（Wikipediaより改変）

(2) どこがボトルネックかを知ろう

計算のボトルネックはどこかというのを考え、そこをなるべく高速化すればプログラムはもっとも効率的に高速化される。普通は論理演算装置（CPU）が一番高速で、次にメモリ、一番低速なのは外部との入出力である。計算の種類やアルゴリズムにより違いはあるが、CPUの能力、メモリバンド幅、入出力のいずれか、または全てがボトルネックになる。必ずしもCPUの能力だけがそうなるのではない。例えばIntel Core i7 920だと約50GFlopsの性能をもつが、これは約400Gbytes/秒のバンド幅をもつ。一方、メモリはPC3-8500で最大約17Gbyes/秒なので、25倍もの開きがある。これを踏まえると、データは何らかの形で入出力なしで再利用できる形にしないと、CPUはフル回転させられない。

BLASでこの話をすると、Level 1, 2はベクトル-ベクトル積、行列-ベクトル積で、再利用がほとんどできないため、メモリバンド幅がボトルネックとなる。例えばPC3-8500では約2〜4GFlops出たら、それからの性能向上はほとんど望めない。メモリからデータ転送は目一杯されているが、この時CPUはかなり遊んでいる。Level 3は行列-行列演算でデータを再利用しやすいのでCPUの処理速度がボトルネックとなる。そのため、CPUの理論性能値に近い性能が出る。従ってIntel Core i7 920だと約50GFlops出る。

(3) 入出力からCPUで計算させるまで

実際にCPUで計算させるには、コンピュータの外からデータを入力し、レジスタまで持ってくる必要がある。その概略を図2に示した。デバイスはスピードの高速な順に容量が少なく、低速になればなるほど容量は多くなる。大体一桁速くなると一桁容量が小さくなると考えてよいだろう。単にメモリから読み出すと思うかもしれないが、実際に計算するまでには、ハードディスクから長い道のりを経て始めてレジスタに載って計算されるのである。一般には、データの再利用といっても様々な処理をすることが多いので、いつどこにデータを保持、転送するというスケジュールを考えてプログラムすることが大変重要となる。

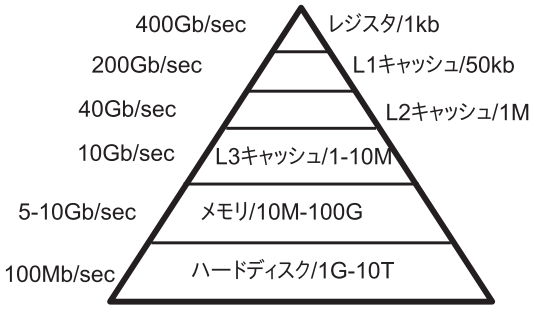

図2　ハードディスクからレジスタまでの非常に大まかなデータ転送スピード（左側）とデバイスと（中側）と容量（右側）

4　GPU（NVIDIA C2050）のアーキテクチャと長所と短所

NVIDIA C2050についてアーキテクチャの長所と短所を、簡単に述べる。ただし他のGPUもメーカーを問わず大体、同じような特徴、長所、短所を持っている。

(1) NVIDIA C2050（GPU）の長所：演算が高速：多くのプロセッサを搭載

NVIDIA社製のGPUのアーキテクチャの大体の図を図3に示した。ストリーミングマルチプロセッサが多数あり、その中に、最小単位であるストリーミングプロセッサ（SP）が入っている。SPの数を多くして高速にするのが特徴である。NVIDIA C2050は1.15GHzで動く448個もSPを持っているため最大

$$448 \times 1.15 = 515\text{GFlops}$$

ものパフォーマンスを持つ。Level 3 BLASは処理能力が高いほど高速になるなので、SPが多いのは有利となる。

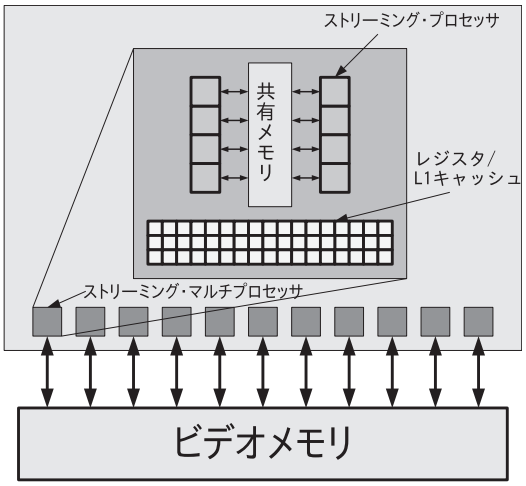

図3 NVIDIA社製のGPUのアーキテクチャ概略：多数のストリーミングプロセッサ、ビデオメモリのバンド幅が高いのが特徴。

(2) GPU (C2050) の長所: メモリが高速であること

NVIDIA C2050はメモリに大変高速なGDDR5を搭載している。これは144GBytes/秒という広いバンド幅を持つ。現在よく使われるPC3-8500では最大17Gbytes/秒であることを考えると8.5倍もの幅がある。従ってGPUを用いるとLevel 1, 2 BLAS、高速フーリエ変換なども高速になる。逆にLevel 3 BLASは恩恵をそんなには受けない。

(3) GPU (C2050) の短所: PCIeバスが低速であること

GPUはCPUとは別で、もう一台のコンピュータである。したがってメモリや情報は共有されない。これらを結ぶのはPCIeバスであり、これは最大転送速度8GB/secとGPUに搭載されたメモリと比べると20倍くらい遅い（図4）。

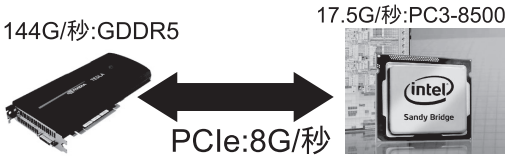

図4 CPU-GPUの転送はPCIeを介して行うが、PCIeバスの転送速度が遅い（PCIe 8GB/s, GPUメモリ144GB/s, CPUメモリ17.5GB/s）

(4) C2050 (GPU) の短所: プログラミングが複雑

GPUは高速なメモリを積んでいるが、その引換にメモリアクセス制限がある。ある一定の大きさの単位での（コアレスな）メモリアクセスをしないと遅くなることがある。さらに、アクティブブロック数の確保などCUDA向けの特殊なプログラミングが要求されることである。またGPUの型番が変わればチューニングのパラメータなどが大きく変わることがある。

5 NVIDIA C2050でのBLAS, LAPACK実習

NVIDIA C2050でのBLAS, LAPACKについて実習してゆく。ここではcuBLASとMAGMAを取り上げた。実習に必要なのはNVIDIA社C2050および、CUDAToolkit 3.2, MAGMA, GotoBLAS2、またはIntel MKL, OSはx86-64（64bit）のLinuxを用いていただきたい。

GPUでBLAS、LAPACKを動かすとうまくゆけばCPUで動かすより5倍から10倍高速化できるが、GotoBLAS2やIntel MKLの場合とは違い、何も考えずそのままライブラリの置き換えのみとすると、逆に低速になる場合がある。高速化にはプログラムの書き換えも必要となる。

GPUのみの速度、CPUの転送まで含んだ実効的な速度では大きな差があるため、ベンチマーク等では二つともまたは、前者のみ（しばしばkernelと呼ばれる; もちろんこちらのほうが一般的に高速）のパフォーマンスとして提示されることがある。ここには注意しなければならない。

(1) cuBLAS実習

cuBLAS[2]はNVIDIAのCUDAでのBLASの実装である。倍精度、複素数にも完全に対応し、ほとんどの関数が比較的高性能である。FORTRAN/C/C++から呼び出せるようになっている。ほぼ同じようなプログラムにはなるのだが、残念ながらコードの変更は必要となる。次節のMAGMAはその高速な実装や、機能拡張などを行っており、いくつかの関数はcuBLASに取り入れられているので性能がまったく同じになる場合がある。ソースは公開されていない。

GPUの使った場合の注意点は先にもあげたとおり二つある。

まずはデータ転送をするPCIeバスが演算速度に比べ圧倒的に低速であること。そのため、GPUによる線形代数演算の加速には、なるべく一括でデータを転送し、GPU上で演算を完結させ、結果を回収するという方法を取り、なるべくCPU-GPU間のデータのやりとりを抑えることが必要になる。

もう一つは高速なメモリアクセスと引換にメモリのアクセスパターンに制限があるということである。制限に合わない場合もアクセスは可能だが、大変遅くなるのである。ベンチマークでは高速になるよう特定の倍数のみ提示することも多く、実用では思ったよりパフォーマンス向上につながらないことがある。

以下に注意点をまとめた。

- 大きなサイズの行列を計算すること。小さいとPCIeバスの転送速度がボトルネックとなり、パフォーマンスが落ちる。
- 行列、ベクトルを一括で転送し、計算結果を最後に回収すること。PCIeバスは低速なのでPCIe-CPUの転送の回数を減らすこと。
- BLAS Level 1, 2も加速される。これはメモリバンド幅が広いことに起因する。ただしCPU-GPU転送を行なわないこと。

- プログラムは変更される。cuBLAS特有のGPUの制御文がいくつか入る（図5）。
- ルーチンを呼び出すには、“cublasDgemm”のように“cublas”をつけ、BLASのルーチン名の最初の文字を大文字にする。
- 参照呼び出しではなく、値呼び出しになっている変数がある（C/C++に自然なように変更されている）。
- 配列は1から始まる（FORTRANと同じ; 前回のチュートリアル参照）。
- 行列は一次配列でcolumn-majorでもつ（FORTRANと同じ; 前回のチュートリアル参照）。
- GPU上で行列、ベクトルを確保するには、ldaなどの行列のサイズを32の倍数に揃える。MAGMAの“testing_dgemm.cpp”等を参照のこと。

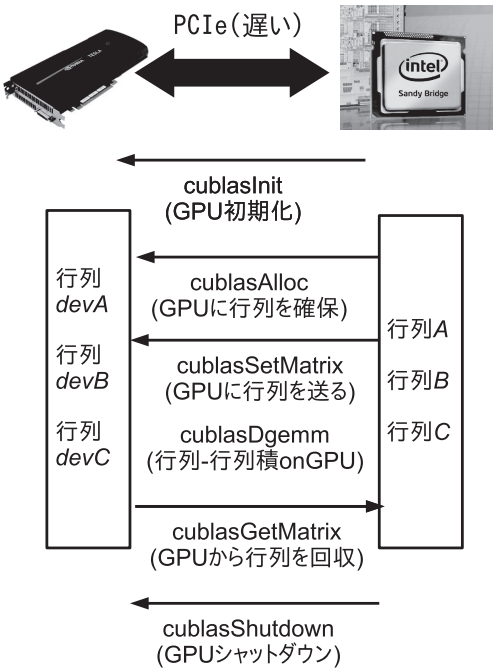

図5　cuBLAS特有のGPUの制御の図

ここで、前回も用いた行列-行列積であるdgemmのcuBLAS版をC++でもう一度サンプルとしてとりあげる。

図6を入力し、

```
$ nvcc -o dgemm_demo dgemm_demo.cpp -lpthread -lcublas \
  -lcudart -L/usr/local/cuda/lib64 -L/usr/lib64
```

とする。エラーが起きなければ

```
$ ./dgemm_demo
# dgemm demo...
  A =[ [ 1.00e+00, 8.00e+00, 3.00e+00];\
       [ 2.00e+00, 1.00e+01, 8.00e+00];\
       [ 9.00e+00, -5.00e+00, -1.00e+00] ]
  B =[ [ 9.00e+00, 8.00e+00, 3.00e+00];\
       [ 3.00e+00, 1.10e+01, 2.30e+00];\
       [ -8.00e+00, 6.00e+00, 1.00e+00] ]
  C =[ [ 3.00e+00, 3.00e+00, 1.20e+00];\
       [ 8.00e+00, 4.00e+00, 8.00e+00];\
       [ 6.00e+00, 1.00e+00, -2.00e+00] ]
alpha = 3.000e+00
beta  = -2.000e+00
  ans=[ [ 2.10e+01, 3.36e+02, 7.08e+01];\
        [ -6.40e+01, 5.14e+02, 9.50e+01];\
        [ 2.10e+02, 3.10e+01, 4.75e+01] ]
#you can check by Matlab by:
alpha * A * B + beta * C =
```

となるはずである。他の関数も同様に利用することができる。

```
// dgemm CUDA test public domain
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cublas.h"

//Matlab/Octave format
void printmat(int N, int M, double *A, int LDA) {
  double mtmp;
  printf("[ ");
  for (int i = 0; i < N; i++) {
    printf("[ ");
    for (int j = 0; j < M; j++) {
      mtmp = A[i + j * LDA];
      printf("%5.2e", mtmp);
      if (j < M - 1) printf(", ");
    } if (i < N - 1) printf("]; ");
    else printf("] ");
  } printf("]");
}
int main()
{
  int n = 3; double alpha, beta;
  cublasStatus statA, statB, statC;
  double *devA, *devB, *devC;
  double *A = new double[n*n];
  double *B = new double[n*n];
  double *C = new double[n*n];
  cublasInit();
  statA = cublasAlloc (n*n, sizeof(*A), (void**)&devA);
  statB = cublasAlloc (n*n, sizeof(*B), (void**)&devB);
  statC = cublasAlloc (n*n, sizeof(*C), (void**)&devC);

  A[0+0*n]=1; A[0+1*n]= 8; A[0+2*n]= 3;
  A[1+0*n]=2; A[1+1*n]=10; A[1+2*n]= 8;
  A[2+0*n]=9; A[2+1*n]=-5; A[2+2*n]=-1;

  B[0+0*n]= 9; B[0+1*n]= 8; B[0+2*n]=3;
  B[1+0*n]= 3; B[1+1*n]=11; B[1+2*n]=2.3;
  B[2+0*n]=-8; B[2+1*n]= 6; B[2+2*n]=1;

  C[0+0*n]=3; C[0+1*n]=3; C[0+2*n]=1.2;
  C[1+0*n]=8; C[1+1*n]=4; C[1+2*n]=8;
  C[2+0*n]=6; C[2+1*n]=1; C[2+2*n]=-2;
```

```
  statA = cublasSetMatrix (n, n, sizeof(*A), A, n, devA, n);
  statB = cublasSetMatrix (n, n, sizeof(*B), B, n, devB, n);
  statC = cublasSetMatrix (n, n, sizeof(*C), C, n, devC, n);

  printf("# dgemm demo...\n");
  printf("A =");printmat(n,n,A,n);printf("\n");
  printf("B =");printmat(n,n,B,n);printf("\n");
  printf("C =");printmat(n,n,C,n);printf("\n");
  alpha = 3.0; beta = -2.0;

  cublasDgemm('n', 'n', n, n, n, alpha,
      devA, n, devB, n, beta, devC, n);

  statA = cublasGetMatrix (n, n, sizeof(*A),
          devA, n, A, n);
  statB = cublasGetMatrix (n, n, sizeof(*B),
          devB, n, B, n);
  statC = cublasGetMatrix (n, n, sizeof(*C),
          devC, n, C, n);

  printf("alpha = %5.3e\n", alpha);
  printf("beta  = %5.3e\n", beta);
  printf("ans="); printmat(n,n,C,n);
  printf("\n");
  printf("#you can check by Matlab by:\n");
  printf("alpha * A * B + beta * C =\n");

  cublasFree (devA);
  cublasFree (devB);
  cublasFree (devC);
  cublasShutdown();
  delete[]C; delete[]B; delete[]A;
}
```

図6 C++でのcuBLASを用いたdgemmのサンプル。行列-行列積を求める。ファイル名は“dgemm_demo.cpp”とすること。

(2) MAGMA実習

MAGMA[3]はStanimire Tomovらのグループで作成されている、NVIDIA GPU向け（CUDA）の線形代数ライブラリである。CPUも同時に使うハイブリッド型のライブラリで、そのためGPU単体より高速となる（ただし現在はCPUを使っていないルーチンも多くある）。

開発は現在精力的に進められており、最新版は2011/4/6に公開された1.0.0RC5である（RC5とは5番目のリリース候補）。cuBLASと同じようにBLAS, LAPACKに準ずるような形でAPIが定められている。LAPACKのルーチンはLU分解、Cholesky分解、QR分解、連立一次方程式、固有値問題特異値分解などよく使われる32のルーチンが実装されている。BLASのルーチンも高速化されたルーチンが提供されている。ただし上にも述べたようにdgemmなどcuBLASに取り込まれてしまっているものもある。さらに、3条項BSDライセンスでソースコードが配布されているため無料で入手でき、誰でも自分のプログラムに組み込んだり、試せるようになっている。

まず、MAGMAのサイト[3]からダウンロードして、次のようにビルドする。

```
$ cd /home/maho
$ tar xvfz magma_1.0.0-rc5.tar.gz
...
$ cd magma_1.0.0-rc5/
$ less README（よく読むこと）
$ emacs make.inc.goto（GotoBLAS2の場合，適切に編集すること）
$ cp make.inc.goto make.inc（GotoBLAS2の場合のみ）
$ emacs make.inc.mkl（Intel MKLの場合，適切に編集すること）
$ cp make.inc.mkl make.inc（Intel MKLの場合のみ）
$ make
...
  testing_cgehrd.o testing_zhetrd.o testing_cgeqrs\
_gpu.o testing_dsytrd.o testing_cgebrd.o testing_\
zgehrd.o testing_zpotrf_gpu.o testing_dsposv_gpu.o\
testing_zgesv_gpu.o
make[1]: Leaving directory '/home/maho/magma_1.0.0\
-rc5/testing'
$
```

だいたい上のようになっていればビルド成功である。

次に、MAGMAでどの程度パフォーマンスが出るかをみよう。MAGMA 1.0.0RC5のdgemmのC2050での正方行列のベンチマーク結果である（図7）。GPUのみの計算では2000次元付近から300GFlops程度出る。ただしCPU-GPU転送を含むとパフォーマンスが落ちる。CPU-GPU転送を含め50GFlopsを越えるのは行列のサイズが400次元をこえたところ、100GFlopsを越えるのは700次元付近からとなる。CPUのBLASを置き換える目的での利用を考えた場合に参考としていただきたい。どちらの場合も行列のサイズが少し変わると性能が10%程度落ち込む場合がある（特にGPUのみの場合、1000次元あたりで顕著）。これはメモリアクセスがあるパターンを満たさない場合は性能が落ちるためである。ここには示さないがcuBLAS3.2は一つ前のMAGMA dgemmなのでまだ少し性能が不安定であった。紙面の都合上、コードを示せないのが残念である。

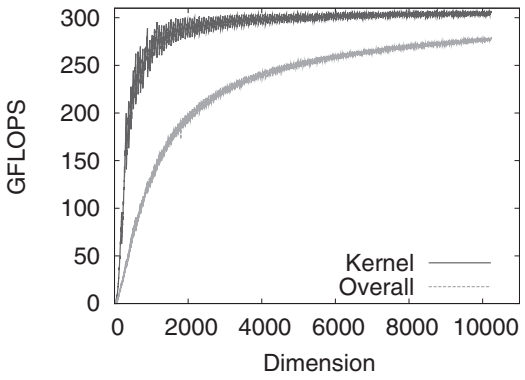

図7 MAGMA1.0.0RC5のdgemmのC2050での正方行列のベンチマーク。KernelはGPUのみのパフォーマンス、OverallはCPU-GPU転送も含んだパフォーマンス値。2000次元より大きいと約300GFlops出る。CPU-GPU転送を含む場合、50GFlops, 100GFlopsを越えるのは400次元、700次元付近からとなる。

さて、最後にLAPACKルーチン“dgetrf”と同じことをMAGMAで行ってみよう。これはLU分解であり、行列Aを下三角行列Lと上三角行列Uの積に分解する。

$$A = LU$$

LU分解は行うと連立一次方程式が簡単に解けたり、他にも様々な用途があるため、有用なルーチンである。例えば行列を

$$A = \begin{pmatrix} 1 & 8 & 3 \\ 2 & 10 & 8 \\ 9 & -5 & -1 \end{pmatrix}$$

としたときのLU分解は

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2222 & 1 & 0 \\ 0.1111 & 0.77 & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} 9 & -5 & -1 \\ 0 & 11.1111 & 8.2222 \\ 0 & 0 & -3.22 \end{pmatrix}$$

MAGMAにはGPU版、CPU-GPUハイブリッド版（つまりCPUもLU分解に使う）の二つが用意されている。今回はCPU-GPUハイブリッド版を使ってみる。図8を入力し、次のようにコンパイルする。ただしIntel MKLの場合は[4]やバージョンの違いも考慮し適切に書き換えることが必要となる。

```
//LU factorization MAGMA public domain
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cuda.h>
#include <cublas.h>
#include <cuda_runtime_api.h>
#include "magma.h"
#include "magma_lapack.h"
#include "testings.h"

//Matlab/Octave format
void printmat(int N, int M, double *A, int LDA) {
  double mtmp;
  printf("[ ");
  for (int i = 0; i < N; i++) {
    printf("[ ");
    for (int j = 0; j < M; j++) {
      mtmp = A[i + j * LDA];
      printf("%5.2e", mtmp);
      if (j < M - 1) printf(", ");
    } if (i < N - 1) printf("]; ");
    else printf("] ");
  } printf("]");
}

int main()
{
  int M=3, N=3, lda, min_mn, nb;
  magma_int_t *ipiv, info;
  double *A;

  min_mn = min(M,N);
  nb     = magma_get_dgetrf_nb(min_mn);
  lda = N;

  TESTING_CUDA_INIT();
  TESTING_MALLOC(ipiv, magma_int_t, min_mn);
  TESTING_HOSTALLOC( A, double, M*N);

  A[0+0*lda]=1; A[0+1*lda]= 8; A[0+2*lda]= 3;
  A[1+0*lda]=2; A[1+1*lda]=10; A[1+2*lda]= 8;
  A[2+0*lda]=9; A[2+1*lda]=-5; A[2+2*lda]=-1;

  printf("A =");printmat(M,N,A,lda);printf("\n");
  magma_dgetrf( M, N, A, lda, ipiv, &info);
  printf("LU =");printmat(M,N,A,lda);printf("\n");

  TESTING_FREE( ipiv );
  TESTING_HOSTFREE( A );
  TESTING_CUDA_FINALIZE();
}
```

図8 C++でのMAGMAを用いたdgetrfのサンプル。LU分解を求める。ファイル名は“testing_dgetrf.cpp”とすること。

```
GotoBLAS2のとき
$ nvcc -o testing_dgetrf testing_dgetrf.cpp -I/home/
maho/\
magma_1.0.0-rc5/testing/ -I/home/maho/
magma_1.0.0-rc5/\
include/ -L/usr/local/cuda/lib64 -L/usr/lib64 -L/home/
maho/\
magma_1.0.0-rc5/lib/ -L/home/maho/GotoBLAS2 -lcuda
-lmagma\
-lmagmablas -lmagma -lcublas -lgoto2
Intel MKLのとき
$ nvcc -o testing_dgetrf testing_dgetrf.cpp -I/home/
maho/\
magma_1.0.0-rc5/testing/ -I/home/maho/magma_1.0.0-rc5/
include/\
 -L/usr/local/cuda/lib64 -L/usr/lib64 -L/home/maho/\
magma_1.0.0-rc5/lib/ -L/opt/intel/Compiler/11.1/072/
mkl/lib/\
em64t/ -lcuda -lmagma -lmagmablas -lmagma -lcublas \
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core \
/opt/intel/Compiler/11.1/072/lib/intel64/libiomp5.a\
 -lpthread
```

これが成功したら実行しよう。

```
$ ./testing_dgetrf
device 0: Tesla C2050, 1147.0 MHz clock, 2687.2 MB
memory
A =[ [ 1.00e+00, 8.00e+00, 3.00e+00]; \
   [ 2.00e+00, 1.00e+01, 8.00e+00]; \
   [ 9.00e+00, -5.00e+00, -1.00e+00] ]
LU =[ [ 9.00e+00, -5.00e+00, -1.00e+00]; \
    [ 2.22e-01, 1.11e+01, 8.22e+00];\
    [ 1.11e-01, 7.70e-01, -3.22e+00] ]
```

となっていればよい。行列のアウトプット*LU*の上三角は*U*行列、下三角に対角要素1を代入すると*L*になっている。MAGMA特有の違いについてみてみよう。まず、TESTING... などはCUDAを呼び出すマクロである。次に、dgetrfの呼び出し方であるが、次のようにほぼLAPACKと準拠している。

```
magma_dgetrf( M, N, A, lda, ipiv, &info);
```

最後に、ワークスペースの問い合わせであるがここはLAPACKと大きく違うところである。

```
magma_get_dgetrf_nb(min_mn);
```

ここは注意すべきだろう。ただしLAPACKのそれよりは自然に見える。

他にも様々なルーチンがあるので、ぜひMAGMAを使ってみていただきたい。

6 終わりに

GPUのNVIDIA C2050でBLAS, LAPACKを使うというチュートリアルを非常に駆け足で執筆した。cuBLAS, MAGMAを主に取り上げた。GPU上でのBLAS, LAPACKは、性能も、プログラミングもまだまだ未成熟だという印象をもたれたと思う。性能には癖があるし、プログラムの書き換えも必要と、一手間かかる。しかし、うまくゆけばあなたのプログラムは大変高速化されるだろう。

今後は様々な速度で動くマルチコアプロセッサが、様々な場所にあり、それらを用いて効率的に動かすというハイブリッドかつヘテロジニアスな環境になるだろう。それにあわせてMAGMAのようなアプローチでBLAS, LAPACKを使うようになると考えられる。

謝辞

日頃から多くの指導、助言、励ましを頂いた、藤澤克樹先生、後藤和茂氏、姫野龍太郎先生、今村俊幸先生、高雄保嘉氏、安井雄一郎君に深く感謝する。

■参考文献

[1] http://www.jsces.org/Issue/Journal/pdf/nakata-0411.pdf
[2] http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf
[3] http://icl.cs.utk.edu/magma
[4] http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor/