GPU を用いた線型計算の高速化と Runge-Kutta 法への応用

Acceleration of Numerical Linear Computation by Using GPU and its Application to an Efficient Implementation of Runge-Kutta Methods

幸谷智紀*

Tomonori KOUYA*

Abstract: GPU (Graphics Processing Unit) is one of remarkable devices, which can extremely accelarate numerical linear computation. In addition to numerical computaion, GPU has been providing its widely applications to various area such as big data analysis day by day. The "CUDA" of NVIDIA Corporation is the most popular GPU environment in the current world. In this paper, we firstly summarize the CUDA GPU's achitecture and functions. Secondly, we report the results of benchmarking test for some existing numerical computation libraries accelarated by NVIDIA CUDA GPU, and finally evaluate the performance of our implementation of ODE(Ordinary Differential Equations) solver based on explicit and implicit Runge-Kutta formulas by using them. We used a high-priced Tesla C2070 in CUDA GPU family throughout this paper, but added some comments of the results on a low-priced GT640.

1. 初めに

科学技術計算の土台となるスーパーコンピューター（スパコン）は，近年，独立した PC やワークステーションとして用いられる小型デスクトップマシンを多数束ねたアーキテクチャの上に構築されるようになっている．小型デスクトップマシンは，スパコンとは直接縁のない多数の一般ユーザによってその動向が左右される．従って，今のスパコンの高性能化は，一般レベルの支持を集めた小型デスクトップマシンの高速化と直結しているといってよい．

小型デスクトップマシンの高性能化は CPU のマルチコア化と GPU の高機能化がドライブしている．動作周波数の単純な増加はリーク電流の増加を招き，消費電力の割には高速化に寄与しないという判断の元，CPU コアを複数搭載して並列性能を高める方向に，小型デスクトップマシンは進化した．その延長上に，グラフィックス性能を高めると同時に，CPU 並みの演算機能を備えた GPU の発展がある．ここ数年のスパコンのトレンドはこの GPU，特に NVIDIA 社の開発した CUDA アーキテクチャ[11] を備えた Tesla カードを多数搭載するという方向にある．ゲームやアニメーションに要求されるグラフィックス性能を高め，多数存在したグラフィックスカードの競合他社を跳ね除けてトップに立った NVIDIA 社は，今や高性能計算用カードメーカーとしても注目を集めている．

本稿ではまず，NVIDIA 社が開発した GPU を用いた高性能計算性能を，現在提供されている LAPACK/BLAS 由来の線型計算ライブラリを用いて計測し，その有効性を確認する．次に，この線型計算ライブラリの応用例として，陽的・陰的 Runge-Kutta 法に基づく常微分方程式ソルバーを実装し，線型常微分方程式を用いてその性能を評価する．最後に結論と今後の課題について述べる．

2. GPU の機能

ここでは，今回使用した CUDA 環境選択の背景と，そのアーキテクチャ，プログラミングモデルを簡単に紹介する．詳細は CUDA Toolkit[10] のドキュメントに詳しいので参照されたい．

2.1 GPU 登場の背景と CUDA

増大する一方の情報量を捌くため，ICT(Information and Communication Technology) を支えるハードウェアは常に情報確信と機能向上が求められている．特に処理速度の向上は至上命題であり，2000 年代に入るまで，小型デスクトップコンピューター (Personal Computer, PC) 用の CPU は動作周波数を引き上げることでその要求に答えてきた．しかし近年は，ランニングコストの引き下げ要求も強く，動作周波数の一方的な引き上げでは消費電力の増大に比して処理速度の向上が図れないという状況に陥っている．そのため，CPU は中核となる処理部分をコア (core) として分離し，複数搭載したマルチコア化することで，並列性能向上を目指すようになり，動作周波数の向上は頭打ち状態となっている．

一方で，3 次元コンピュータグラフィックス (3DCG) を多用する，ゲームをはじめとするアミューズメント方面からは，画面の高精度化と複雑化する 3DCG 表現の要求に答えるため，グラフィックス処理を高速に処理するための専用ハードウェア，即ちグラフィックスカードの高性能化の要求が強い．そのため，1990 年代は多様なグラフィックスカードを製造販売するメーカーが群雄割拠する状態となり，その中からグラフィックス処理を高速化するための中核チップ，即ち GPU(Graphics Processing Unit) が誕生した．最終的には，CPU にグラフィックス機能を搭載した Intel を除けば，GPU の主要メーカーは NVIDIA 社と AMD 社 (CPU メーカーと統合) の二強に集約され，現在に至っている．

このうち NVIDIA 社は，早くから GPU にグラフィックス以外の計算処理も行わせるためのハードウェアアーキテクチャ CUDA と開発ソフトウェア CUDA Toolkit の提供に乗り出した．2006 年 11 月に CUDA アーキテクチャの構想を発表，翌年 2007 年 6 月に CUDA Toolkit Version 1.0 を発表した．後者は CUDA C コンパイラ (後述)，PTX アセンブラ (GPU が直接処理するアセンブラ)，サンプルプログラム，ドキュメント類が同梱されたもので，CUFFT(Fast Fourier Transform) や後述する線型計算ライブラリ CUBLAS の Version 1.0 も合わせて提供されている．

現在発売されているコンシューマー向けの NVIDIA 社グラフィックスカードに用いられている GPU(GTX, Tesla, Quadro

---

2013 年 3 月 18 日 受理

*総合情報学部コンピュータシステム学科

シリーズ) には全てこの CUDA 機構が搭載されており，2012 年 10 月に公開された CUDA Toolkit 5.0 はこの全てのグラフィックスカードを用いて開発・計算が可能となっている．ハードウェア以外は無償公開されており，誰でも開発に参加できるため，世界規模で急速な開発競争が行われている．

なお，Intel CPU/MIC(Many Integrated Core) コプロセッサや，AMD 社の GPU である Radeon シリーズ向けの計算処理機能を提供する OpenCL 規格 [2] も CUDA 登場後に提唱され，NVIDIA 社の GPU でもこの OpenCL 規格に則った機能を用いてプログラミングは可能である．しかし現状，OpenCL 向けの，特に線型計算ライブラリとしては clMAGMA[8] しか存在していない状況であり，プログラミング環境も CUDA Toolkit に比べて見劣りする．今後，OpenCL 規格がどのメーカーのハードウェア上でも使用でき，開発環境が整ってくれば有力な選択肢になる可能性はあるが，今回はその使用を見送った．状況が変わればまた別途調査した上で開発環境として考慮したいと考えている．

2.2 CUDA アーキテクチャと用語

PC 用のグラフィックカードの大半は現在 PCIe バスに刺して使用するものが多い．GPU はこのカード上に搭載され，一般ユーザーからはもっぱらグラフィックス処理用として使用される．高性能計算用途に使用するには，GPU が処理可能な PTX コードを用いたプログラムが必要である．CUDA Toolkit がリリースされている現在は，同梱されている NVCC コンパイラ (CUDA C) を用いて C++言語とほぼ同等のプログラムが構築できる．しかし，性能を発揮するためには，Fig.1 に示すような複雑な CUDA アーキテクチャを理解した上で，性能を発揮するようチューニングを行う必要がある．

GPU を使用するプログラムは，CPU 側で動作する部分 (ホスト (Host) 側) と，GPU 側で動作する部分 (デバイス (Device) 側) に分かれる．デバイス側で動作する部分は，大量の CUDA コアを効率よく使用するため，スレッド (Thread) 単位で実行される．このスレッドはワープ (Warp) という単位でまとめて実行され，更にブロック (Block) という単位で同期させることができる．ブロックは更にグリッド (Grid) という単位で管理される．ユーザが指定できるのはブロック数と，ブロック当たりのスレッド数で，グリッドはブロック数に応じて自動的に決められる．

各スレッドは SPMD(Single Program, Multiple Data) の形式で実行される．この際，スレッドから直接操作できるデータは，ローカルメモリ (Local memory)，ブロックごとの共有メモリ (Shared memory)，全てのスレッドからアクセスできるグローバルメモリ (Global memory) で記憶してあるものだけである．これらはすべてグラフィックスカード及び GPU 内部のメモリで，ホスト側からは PCIe バスを通じて明示的にやり取りされる．

2.3 CUDA プログラムの例

CUDA プログラムの例として，直接法 (LU 分解+前進＆後退代入) を行うプログラムを下記に示す．メイン関数部分は通常の C/C++コンパイラを使用してもよい．ここでは今回実装した BNCuda ライブラリにある関数 (_bncuda 接頭詞付き関数) を用いている．

```
 1:#include <stdio.h>
 2:#include <cuda.h>
 3:#include <cuda_device_runtime_api.h>
 4:#include "bncuda.h"
 5:
 6:// メイン関数
 7:int main()
 8:{
 9:  DMatrix da, db, dx; // ホスト側変数
10:  DMatrix dev_da, dev_db, dev_dx; //デバイス側変数
11:
12:  // ホスト側で行列，ベクトルを確保
13:  da = init_dmatrix(dim, dim);
14:  db = init_dvector(dim);
15:  dx = init_dvector(dim);
16:
17:  // 連立一次方程式をセット
18:  get_dproblem(da, db, dans);
19:
20:  // 行列，ベクトルをデバイス側に確保&転送
21:  dev_da = _bncuda_init_set_dmatrix_rowmajor(da);
22:  dev_db = _bncuda_init_set_dvector(db);
23:  dev_dx = _bncuda_init_dvector(dim);
24:
25:  // LU 分解
26:  ret = _bncuda_DLU(dev_da);
27:
28:  // 前進＆後退代入
29:  ret = _bncuda_SolveDLS(dev_dx, dev_da, dev_db);
30:
31:  // 数値解をホスト側に転送
32:  _bncuda_get_dvector(dx, dev_dx);
33:
34:  // 画面表示
35:  print_dvector(dx);
36:
37:  // デバイス側変数の消去
38:  _bncuda_free_dmatrix(dev_da);
39:  _bncuda_free_dvector(dev_db);
40:  _bncuda_free_dvector(dev_dx);
41:
42:  // ホスト側変数の消去
43:  free_dmatrix(da);
44:  free_dvector(db);
45:  free_dvector(dx);
46:
47:  return 0;
48:}
```

このように CPU 上で通常の C/C++コンパイラだけで済むプログラムをホストプログラムと呼ぶ．後述する，CUDA 上で動作する線型計算ライブラリである CUBLAS や MAGMA はホストプログラムから呼び出すだけで CUDA GPU を使用できる．

しかし，直接 GPU の並列計算機能を生かそうとすると，ホストプログラムだけでは不可能な，グローバルメモリやシェアードメモリ内データの細かい操作が必要となる．そのために，カーネル (kernel) 関数を実装しなければならないケースも多数存在する．

カーネル関数ではデバイス側で行うマルチスレッドの処理を細かく記述できる．接頭詞`__global__`を付加するためグローバル関数とも呼ばれる．グローバル関数を含むプログラムソースは NVCC コンパイラ (CUDA C) を用いてコンパイルする必要がある．

例えば，LU 分解を倍精度で行うカーネル関数 (1 ブロック版`_bncu_DLU_pt`, 複数ブロック版`_bncu_DLU_pt2`) は次のよ

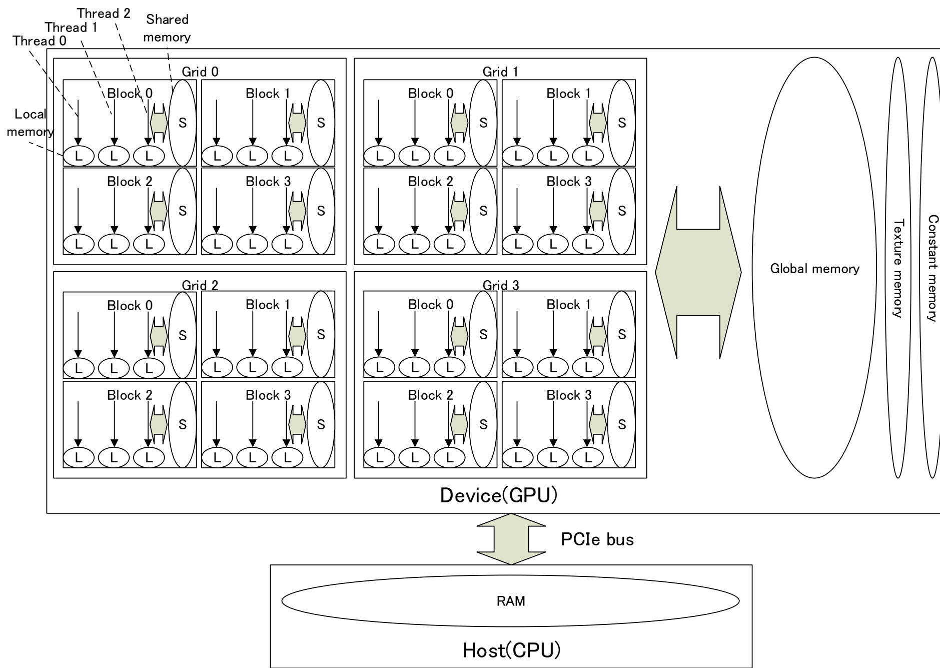

Fig. 1: CUDA GPU のアーキテクチャ

うに実装できる．

```
 1:// LU 分解のカーネル関数 (1 ブロック)
 2:__global__ void _bncu_DLU_pt(int *dev_ret, double *ma
t_element, int mat_row_dim, int mat_col_dim);
 3:{
 4:  int thread_index = threadIdx.x + blockIdx.x * block
Dim.x;
 5:  int stride = blockDim.x * gridDim.x;
 6:  int stride = blockDim.x;
 7:  __shared__ int i, j, k;
 8:  __shared__ double dtmp, dmaxii;
 9:
10:  for(i = 0; i < mat_row_dim; i++)
11:  {
12:    // LU 分解ループ (ここから)
13:    dmaxii = fabs(mat_element[i * mat_row_dim + i]);
14:    if(dmaxii == 0.0)
15:    {
16:      if(dev_ret != NULL)
17:        *dev_ret = -1;
18:      return;
19:    }
20:
21:    for(j = (i + 1 + thread_index); j < mat_row_dim;
j += stride)
22:      mat_element[j * mat_col_dim + i] /= mat_element[
i * mat_col_dim + i];
23:
24:    __syncthreads();
25:
26:    for(j = (i + 1 + thread_index); j < mat_row_dim;
j += stride)
27:    {
28:      for(k = (i + 1); k < mat_col_dim; k++)
29:        mat_element[j * mat_col_dim + k] -= mat_eleme
nt[j * mat_col_dim + i] * mat_element[i * mat_col_dim +
k];
30:    }
31:    __syncthreads();
32:    // LU 分解ループ (ここまで)
33:  }
34:
35:  if(dev_ret != NULL)
36:    *dev_ret = 0;
37:
38:  return;
39:}
40:
41:// LU 分解のカーネル関数 (複数ブロック)
42:__global__ void _bncu_DLU_pt2_in(int *dev_ret, double
 *mat_element, int mat_row_dim, int mat_col_dim, int sta
rt_i)
   (略)
```

__shared__接頭詞が付加された変数はブロック単位で用意されるシェアードメモリに置くことになる．関数の引数に与えられた変数は，実行時にホストプログラムから引き渡され，特に指定しない限りグローバルメモリに置かれる．従って，ベクトルや行列要素を格納したメモリを指定するポインタはあらかじめホストプログラムから CUDA API 関数を用いて確保しておく必要がある．

カーネル関数は

```
_bncu_DLU_pt2<<<ブロック数，スレッド数>>>(引数リスト)
```

という形でホストプログラムから呼び出せる．ブロック数，スレッド数は 3 次元整数配列を用いて渡すこともできるが，今回は使用していない．

これらのカーネル関数を呼び出して実行するホストプログラム側の関数は下記のようになる．

```
 1:#define MAX_NUM_THREADS 256
 2:
 3:// LU 分解のホスト関数
 4:int _bncuda_DLU(DMatrix mat)
 5:{
 6:    (略)
 7:  num_threads = (row_dim <= MAX_NUM_THREADS) ? row_di
m : MAX_NUM_THREADS;
 8:  num_blocks  = (int)ceil((double)row_dim / MAX_NUM_T
HREADS);
 9:
10:  // 1 ブロック使用の場合
11:  if(num_blocks == 1)
12:    _bncu_DLU_pt<<<1, num_threads>>>(dev_ret, mat->el
ement, (int)(mat->row_dim), (int)(mat->col_dim));
13:  // 複数ブロック使用の場合
14:  else
15:  {
16:    int i;
17:    _bncuda_set_i(dev_ret, 0);
18:
19:    for(i = 0; i < mat->row_dim; i++)
20:    {
21:      _bncu_DLU_pt2_in<<<num_blocks, num_threads>>>(d
ev_ret, mat->element, (int)(mat->row_dim), (int)(mat->co
l_dim), i);
22:    }
23:  }
24:
25:  ret = _bncuda_get_i(dev_ret);
26:
```

これらのカーネル関数を含むプログラムは NVCC でコンパイルする．その後は NVCC でも通常の C/C++コンパイラを用いてもリンクし実行ファイルを生成することができる．

3. 線型計算ライブラリ CUBLAS と MAGMA

今回我々は，CUDA GPU 上で動作する既存の線型計算ライブラリである CUBLAS 5.0[9] と MAGMA 1.3.0[8] を用いて陽的・陰的 Runge-Kutta 法の実装を行った．この二つは CUDA を土台にした LAPACK/BLAS 互換の機能を提供している高性能なライブラリで，特に MAGMA はホストプログラムからの利用を前提としたものである．ここではその機能の紹介をした後，BLAS 性能の評価と，それらを利用した連立一次方程式の解法を実装し，その性能評価を行う．

3.1 LAPACK 互換ライブラリの現状

現在のコンピューター上における基本的な線型計算は，単精度・倍精度浮動小数点数演算を利用する限り，LAPACK(Linear Algebra PACKage)[5] 互換の高性能なライブラリを用いることが普通になっている．特に ATLAS(Automatically Tuned Linear Algebra Software)[12] や Intel Math Kernel(MKL)[6] は，LAPACK の基盤となる基本線型計算を担当する BLAS(Basic Linear Algebra Subprograms)[13] の部分を高速化するため，SIMD 命令やキャッシュヒット率の最適化を行って最大の性能を導き出す工夫が随所になされている．多数のユーザーから使用されていることから信頼性も高い．

従って，GPU 上でも LAPACK/BLAS 互換の機能を同様の関数 API が使用できることが望ましい．そのために，NVIDIA 社からは，CUDA Toolkit と共に CUBLAS が配布されている．また，LAPACK 開発グループからは，CUBLAS と既存の LAPACK/BLAS を利用した，大規模計算にも高性能な線型計算が可能な MAGMA(Matrix Algebra on GPU and Multicore Architectures) が配布されている．これらのソフトウェア構造を Fig.2 に示す．

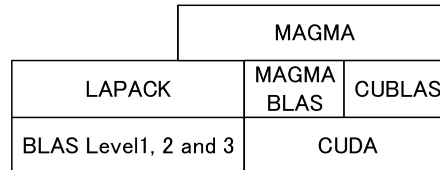

Fig. 2: LAPACK/BLAS, MAGMA, CUBLAS のソフトウェア構造

CUBLAS は一部の例外を除くと BLAS 互換の機能しか提供されておらず，ベクトル演算 (BLAS Level 1)，行列・ベクトル演算 (BLAS Level 2)，行列演算 (BLAS Level 3) しか実行できない．そのため，さらに上位の機能，たとえば連立一次方程式や固有値問題を扱うためには LAPACK の機能を備えつつある MAGMA をホストプログラムから利用する他ない．

以下では，CUBLAS と MAGMA を用いた基本的な線型計算の性能評価を行った結果について報告する．なお，第 5 節を除き，使用した計算機環境は下記の通りである．

H/W Intel Core i7-3930K (3.2GHz), 16GB RAM, NVIDIA Tesla C2070

S/W Scientific Linux 6.3 x86_64, Intel Compiler 13.0 + Intel Math Kernel, CUDA Toolkit 5.0, MAGMA 1.3.0

3.2 線型計算ライブラリの性能評価

まず，MAGMA の Testing ディレクトリの中にある CUBLAS と MAGMA BLAS の性能を比較するプログラムを用いた結果について報告する．今回 Runge-Kutta 法の実装に使用する線型計算のうち，計算時間の大部分を占めるのは BLAS Level 2 の行列・ベクトル積と，連立一次方程式の直接解法 (LU 分解+前進・後退代入) である．後者には BLAS Level 3 の行列積が使用されている．

そこで，単精度 (S, s), 倍精度 (D, d) の行列ベクトル積の性能を GFLOP/s(Giga FLOating-point operations Per second) で評価する testing_[s,d]gemv と，行列積の性能評価する testing_[s,d]gemm を実行した．その結果を Fig.3 に示す．

前述した CUDA アーキテクチャ(Fig.1) に示す通り，多数の軽い計算を大量に並列実行する特長を生かせる行列積の性能が最も高く，行列ベクトル積の最高性能が約 40GFLOP/s なのに比べ，約 600GFLOP/s の性能が叩き出せる．逆に，小さいサイズの計算は苦手で，PCIe バスを通じたデータ転送の時間が無視できない比率となって現れてくる．小規模な計算は CPU で計算を行った方が良い場合もあるだろう．

3.3 連立一次方程式の性能評価

次に，直接法を用いた性能評価結果を示す．ここではランダムに生成した実正方行列 (密行列) を用いて，MAGMA から提供されている部分ピボット選択付き LU 分解 (LAPACK の [S,D]GETRF 関数相当) と前進・後退代入 (同 [S,D]GETRS 関数相当) を組み合わせて数値解を求め，CPU との実行結果を

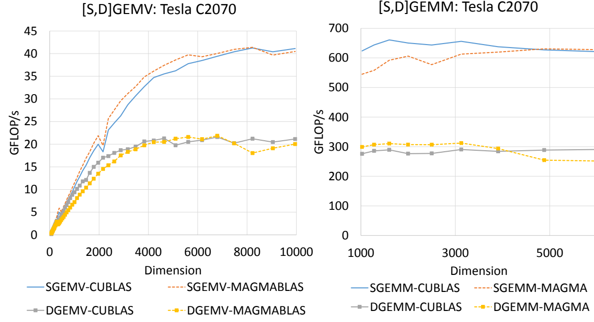

Fig. 3: CUBLAS, MAGMA BLAS の性能評価（左：行列・ベクトル積，右：行列積)

比較して計算ミスがないことを確認しながら，GFLOP/s 値を求めた．その結果を Fig.4 に示す．比較のため，CPU 上では最高性能を発揮する MKL の結果も掲載してある．

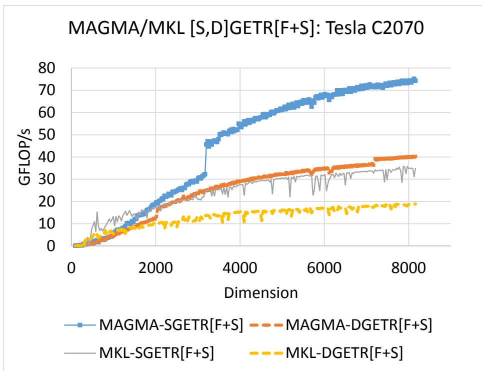

Fig. 4: LU 分解 (xGETRF)+前進・交代代入 (xGETRS) 性能評価

前述したように，1000 次元程度の小規模な連立一次方程式は，CPU 上で計算した方が良いケースが多数存在していることがわかる．大規模問題になればなるほど GPU 上での計算が有利で，単精度計算，倍精度計算どちらも CPU より 2〜3 倍高速になる．特徴的なのはある特定の次元数を超えると GPU 上の計算性能が急激にアップすることだが，これについては原因は不明である．

次に，CUBLAS を用いて実装した，行列・ベクトル積を多用する 4 つの倍精度積型 Krylov 部分空間法 (BiCG, CGS, BiCGSTAB, GPBiCG 法) の性能評価結果について述べる．連立一次方程式は直接法の問題と同様にランダムに倍精度密正方行列を生成して導出した．その結果を Table 1 に示す．収束に要した反復回数と計算時間 (カッコ内, 単位は秒) を掲載してある．下線は，収束に失敗したことを示している．

収束に成功したケースを見る限り，全ての解法は CPU, GPU どちらも同程度の反復回数で収束していることがわかる．また，2000 次元以上になると GPU の方が高速に計算できていることがわかる．逆に 1000 次元以下では同程度か GPU の方が若干遅いというケースが頻出する．これは密行列の例であるが，疎行列のように行列要素へのアクセス時間が多様になる場合は詳細なベンチマークが不可欠であろう．

4. 陽的・陰的 Runge-Kutta 法の実装と性能評価

以上述べてきた線型計算ライブラリの応用として，今回は陽的・陰的 Runge-Kutta 法の実装を行う．実装に当たっては，ベースとなっている BNCpack[4] の線型計算関数と互換になるよう，GPU 上で CUBLAS(BLAS Level1, 2, 3 相当の計算) と MAGMA(連立一次方程式の直接法) を用いて実行できるように書き換えを行ったホストプログラムである BNCuda ライブラリを作り，それを利用して倍精度計算のみに対応した実装を行った．しかし後述するように，ブロック行列生成の部分だけはホストプログラムだけでは実現できなかったため，この部分のみカーネル関数を実装して自前の並列化を行っている．ここでは簡単にそのアルゴリズムを述べ，実装した ODE ソルバーの性能評価結果を示す．

4.1 m 段 Runge-Kutta 法と陽的 Runge-Kutta 法

対象となる n 次元常微分方程式 (ODE) の初期値問題を (1) に示す．

$$\begin{cases} \dfrac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) \\ \\ \mathbf{y}(t_0) = \mathbf{y}_0 \in \mathbb{R}^n \end{cases} \tag{1}$$

この ODE の，$t_1 = t_0 + h_0, t_2 = t_1 + h_1, ..., t_{k+1} = t_k + h_k...$ における解 $\mathbf{y}(t_{k+1})$ の近似値を求める m 段 Runge-Kutta 法は次のように計算を行う．

まず下記の mn 次元非線型連立方程式 $(*)$ を解き，$\mathbf{k}_1, \mathbf{k}_2, ..., \mathbf{k}_m$ を求める．

$$(*) \begin{cases} \mathbf{k}_1 &= \mathbf{f}(t_k + c_1 h_k, \mathbf{y}_k + h_k \cdot \sum_{j=1}^{m} a_{1j}\mathbf{k}_j) \\ \mathbf{k}_2 &= \mathbf{f}(t_k + c_2 h_k, \mathbf{y}_k + h_k \cdot \sum_{j=1}^{m} a_{2j}\mathbf{k}_j) \\ & \vdots \\ \mathbf{k}_m &= \mathbf{f}(t_k + c_m h_k, \mathbf{y}_k + h_k \cdot \sum_{j=1}^{m} a_{mj}\mathbf{k}_j) \end{cases}$$

ここで，$c_1, ..., c_m, a_{11}, ..., a_{mm}, b_1, ..., b_m$ は Runge-Kutta 法を規定する定数である．以下，下記のように表形式で記述する．

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1m} \\ c_2 & a_{21} & a_{21} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ c_m & a_{m1} & a_{m2} & \cdots & a_{m,m} \\ \hline & b_1 & b_2 & \cdots & b_m \end{array} = \begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \end{array} \tag{2}$$

こうして求めた k_i $(i = 1, 2, ..., m)$ を用いて次の近似値 $\mathbf{y}_{k+1} \approx \mathbf{y}(t_{k+1})$ を

$$\mathbf{y}_{k+1} := \mathbf{y}_k + h_k \sum_{j=1}^{m} b_j \mathbf{k}_j \approx \mathbf{y}(t_{k+1})$$

として計算する．近似解と真の解との理論誤差が $O(h_k^s)$ $(k \geq 2)$ となる時，この公式を m 段 s 次公式と呼ぶ．

Runge-Kutta 法は係数 (2) の形式よって分類される．このうち A の対角成分が全てゼロとなる下三角行列になる公式を陽的 Runge-Kutta 法と呼ぶ．今回はそのうち 7 段 6 次公式となる下記の係数を用いる．

Table 1: 積型 Krylov 部分空間法の性能評価: 反復回数 (秒数)

	BiCG		CGS		BiCGSTAB		GPBiCG	
n	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
500	406(0.15)	372(0.09)	551(0.1)	488(0.15)	249(0.05)	293(0.13)	285(0.05)	268(0.13)
1000	858(1.08)	983(0.42)	1093(0.75)	1150(0.59)	470(0.32)	471(0.3)	517(0.36)	531(0.35)
2000	1566(17.19)	1524(1.52)	3441(14.2)	2615(2.85)	921(3.8)	894(1.08)	878(3.63)	961(1.2)
3000	4777(98.67)	9000(19.46)	3993(39.2)	9000(21.73)	1244(12.19)	1118(2.82)	1332(13.12)	1379(3.54)

7 段 6 次 : Butcher[3)]

$$
\begin{array}{c|ccccccc}
\frac{1}{3} & \frac{1}{3} & & & & & & \\
\frac{2}{3} & 0 & \frac{2}{3} & & & & & \\
\frac{1}{3} & \frac{1}{12} & \frac{1}{3} & -\frac{1}{12} & & & & \\
\frac{1}{2} & -\frac{1}{16} & \frac{9}{8} & -\frac{3}{16} & -\frac{3}{8} & & & \\
\frac{1}{2} & 0 & \frac{9}{8} & -\frac{3}{8} & -\frac{3}{4} & \frac{1}{2} & & \\
1 & \frac{9}{44} & -\frac{9}{11} & \frac{63}{44} & \frac{18}{11} & 0 & -\frac{16}{11} & \\
\hline
 & \frac{11}{120} & 0 & \frac{27}{40} & \frac{27}{40} & -\frac{4}{15} & -\frac{4}{15} & \frac{11}{120}
\end{array}
$$

陽的 Runge-Kutta 法の場合，(∗) は $\mathbf{k}_1$ から $\mathbf{k}_m$ まで順次求められる．そのためプログラムが簡単に構築できるというメリットがある．反面，段数に比して次数の制限があり，実用的な公式を作るのは手間がかかる．また，硬い (Stiff) 方程式と呼ばれる Lipschitz 定数が大きな ODE に対しては刻み幅 h_k を小さく取って計算する必要がある．

4.2 陰的 Runge-Kutta 法のアルゴリズム

陰的 Runge-Kutta 法は (2) の A の対角成分より上に非ゼロ成分がある係数を持つ．今回はフル陰的公式と呼ばれる，3 段 6 次の Gauss 型公式を用いる．

$$
\begin{array}{c|ccc}
\frac{5-\sqrt{15}}{10} & \frac{5}{36} & \frac{10-3\sqrt{15}}{45} & \frac{25-6\sqrt{15}}{180} \\
\frac{1}{2} & \frac{10+3\sqrt{15}}{72} & \frac{2}{9} & \frac{10-3\sqrt{15}}{72} \\
\frac{5+\sqrt{15}}{10} & \frac{25+6\sqrt{15}}{180} & \frac{10+3\sqrt{15}}{45} & \frac{5}{36} \\
\hline
 & \frac{5}{18} & \frac{8}{18} & \frac{5}{18}
\end{array}
$$

この場合，非線型連立方程式 (∗) を解く作業は簡単ではない．硬い方程式に対して実用的な時間で求めるためには，Newton 法が良いとされている．この時の反復式は下記のようになる．

初期値: $\mathbf{k}_1^{(0)}, ..., \mathbf{k}_m^{(0)}$

$$
\begin{bmatrix} \mathbf{k}_1^{(l+1)} \\ \mathbf{k}_2^{(l+1)} \\ \vdots \\ \mathbf{k}_m^{(l+1)} \end{bmatrix} := \begin{bmatrix} \mathbf{k}_1^{(l)} \\ \mathbf{k}_2^{(l)} \\ \vdots \\ \mathbf{k}_m^{(l)} \end{bmatrix}
-J^{-1}(\mathbf{k}_1^{(l)}, ..., \mathbf{k}_m^{(l)}) \begin{bmatrix} \mathbf{k}_1^{(l)} - \mathbf{f}(t_k + c_1 h_k, \mathbf{y}_k + h_k \sum_{j=1}^{m} a_{1j}\mathbf{k}_j^{(l)}) \\ \mathbf{k}_2^{(l)} - \mathbf{f}(t_k + c_2 h_k, \mathbf{y}_k + h_k \sum_{j=1}^{m} a_{2j}\mathbf{k}_j^{(l)}) \\ \vdots \\ \mathbf{k}_m^{(l)} - \mathbf{f}(t_k + c_m h_k, \mathbf{y}_k + h_k \sum_{j=1}^{m} a_{mj}\mathbf{k}_j^{(l)}) \end{bmatrix}
$$

ここで，$J(\mathbf{k}_1^{(l)}, \mathbf{k}_2^{(l)}, ..., \mathbf{k}_m^{(l)}) \in \mathbb{R}^{mn \times mn}$ は

$$
J(\mathbf{k}_1^{(l)}, \mathbf{k}_2^{(l)}, ..., \mathbf{k}_m^{(l)}) = \left[\begin{array}{c|c|c|c} I_n - J_{11} & -J_{12} & \cdots & -J_{1m} \\ \hline -J_{21} & I_n - J_{22} & \cdots & -J_{2m} \\ \hline \vdots & \vdots & & \vdots \\ \hline -J_{m1} & -J_{m2} & \cdots & I_n - J_{mm} \end{array}\right]
$$

である．但し，

$$
J_{pq} = h_k a_{pq} \frac{\partial}{\partial \mathbf{y}} \mathbf{f}(t_k + c_p h_k, \mathbf{y}_k + h_k \sum_{j=1}^{m} a_{pj}\mathbf{k}_j^{(l)}) \in \mathbb{R}^{n \times n}
$$

I_n : n 次元単位行列

である．

実際の計算においては，Newton 法の反復計算の負荷，特に連立一次方程式を解く手間を減らすため，J を下記のように固定した準 Newton 法を用いることが多い．

$$
J(\mathbf{k}_1^{(l)}, \mathbf{k}_2^{(l)}, ..., \mathbf{k}_m^{(l)}) = J(\mathbf{k}_1^{(0)}, \mathbf{k}_2^{(0)}, ..., \mathbf{k}_m^{(0)})
$$

前述したように，今回はなるべくカーネル関数は記述せず，既存の線型計算ライブラリを用いたホストプログラムとして作成するよう心がけたが，特にこの陰的 Runge-Kutta 法の実装においてはブロック行列 $J(\mathbf{k}_1^{(0)}, \mathbf{k}_2^{(0)}, ..., \mathbf{k}_m^{(0)})$ を構築しなければ，MAGMA の連立一次方程式ルーチンが使えないことになる．従って，この部分のみ，カーネル関数`_bncu_bigmat_dmatrix`を実装し，並列実行して計算時間を減らすことに成功した (Fig.5).

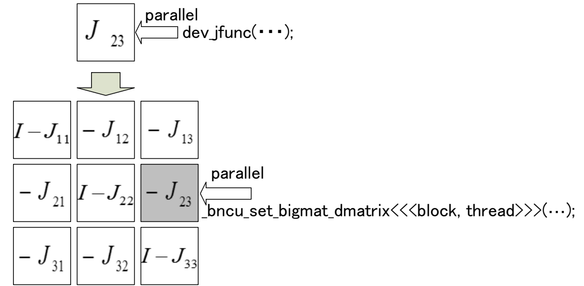

Fig. 5: 陰的 Runge-Kutta 法におけるブロック行列生成法

4.3 線型常微分方程式を用いた性能評価

一様乱数を用いて生成した正則行列 R と逆行列 R^{-1} を用いて

$$
\begin{cases} \dfrac{d\mathbf{y}}{dt} = -(R \ \mathrm{diag}(n, n-1, ..., 1) \ R^{-1}) \ \mathbf{y} \\ \mathbf{y}(0) = [1 \ ... \ 1]^T \end{cases}
$$

積分区間:[0, 1]

という定係数常微分方程式を作成．$n = 50, 100, 200, 500, 1000$ とした時の性能評価を行う．今回，陰的 Runge-Kutta 法の反復計算においては MAGMA の提供する倍精度直接法ルーチンを用いて計算を行っている．

まず，固定刻み幅 $h = 1/10$ とした時の結果を Fig.2 に示す．ERK76(陽的 7 段 6 次) と IRK36(陰的 3 段 6 次) を比較したものである．

Table 2: 7 段 6 次陽的 Runge-Kutta 法 (ERK76) と 3 段 6 次陰的 Runge-Kutta 法 (IRK36) の性能評価 (秒)

	ERK76		IRK36	
n	CPU	GPU	CPU	GPU
50	9.0×10^{-5}	0.0046	0.08	1.13
100	0.00026	0.0051	0.64	1.23
200	0.001	0.0062	4.96	2.3
500	0.0062	0.012	79.15	6.99
1000	0.025	0.019	633.37	15.91

すでに見てきた通り，並列効果が見込めない低次元問題に対しては GPU の効果は薄い．陽的解法の場合，$\mathbf{f}(t, \mathbf{y}) = A\mathbf{y}$ を呼び出す部分を除いてはベクトル演算のみから構成されており，1000 次元になってようやく GPU の方が高速になる．逆に陰的解法の場合は準 Newton 法の反復において必ず連立一次方程式を解く必要があり，100 次元以下では CPU の方が高速，200 次元以上では GPU の方が高速で，それ以上の次元数になると更にその差は大きくなる．

現在の実装では陰的解法の低速さが目立つ．実際，1000 次元の問題に対して GPU を用いた場合，同程度のノルム相対誤差になるように刻み幅（刻み数）を調節して計算時間を比較しみた結果を Table 3 に示す．

Table 3: 1000 次元問題における陽的解法と陰的解法の GPU 実装の比較

	Normwise Rel.Err.	#steps	Comp.Time(s)
ERK76	8.32×10^{-3}	352	0.68
IRK36	6.95×10^{-3}	17	28

かなり硬い問題になっているため，陽的解法の刻み数 (#steps) は陰的解法の約 20 倍になっているが，既に Table 2 に示した通り，陰的解法は陽的解法より 1 刻みあたりの計算時間が約 1000 倍遅くなっているために，トータルの計算時間の差は約 50 倍開いたままになっている．

今回の陰的解法の実装は，Hairer の RADAU5[1] や Jay の SPARK3[7] に示されているような行列のリダクションは行っていない．また，混合精度反復改良法による高速化 [14] もなされておらず，高速化の余地は多数残っていると思われる．

5. GT640 における性能評価

以上，高性能な計算用 GPU である Tesla C2070 を使用した性能評価結果を示してきたが，カード一枚で 20 万円以上 (2013 年 3 月現在の実勢価格) と非常に高価である．それに対して，汎用グラフィックスカード用 GPU を搭載したグラフィックスカードは 10 万円以内で買えるものが殆どである．ここではその中でも非常に安価な GT640(実勢価格 1 万円以内) を用いて，今まで示してきた結果と比較するための性能評価を行う．計算環境は下記の通りである．GPU のスペック以外は Tesla とほぼ同一の環境である

H/W Intel Core i7-3770 (3.4GHz), 32GB RAM, NVIDIA GT 640

S/W Scientific Linux 6.3 x86_64, Intel Compiler 13.0 + Intel Math Kernel, CUDA Toolkit 5.0, MAGMA 1.3.0

まず，線型計算の性能評価を行った結果を Fig.6 に示す．それぞれ行列ベクトル積，行列積の結果である．

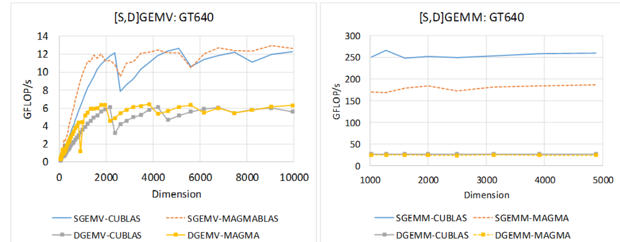

Fig. 6: GT640 の線型計算の性能評価（左：行列・ベクトル積，右：行列積)

全体的に，全ての計算性能は Tesla C2070 の半分程度になっていることが分かる．特徴的なのは，行列・ベクトル積においては性能の激しいブレ，行列積においては，特に単精度計算における CUBLAS と MAGMA の性能差の大きさである．行列積においては全般的に CUBLAS の性能が高い．

MAGMA の LU 分解，前進・後退代入のルーチンを用いた性能評価結果を Fig.7 に示す．

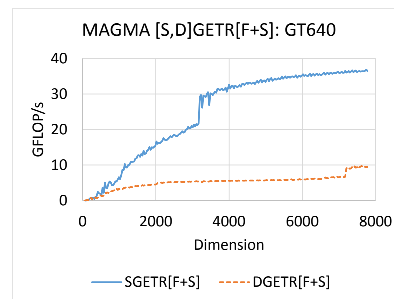

Fig. 7: GT640 の直接法の性能評価

行列ベクトル積，行列積の結果と比較して，直接法においては単精度と倍精度の落差が大きいことが分かる．

これらの結果から容易に推察できるのは，Runge-Kutta 法についても Tesla C2070 に比べて 2 倍以上の性能が落ちるということである．実際にベンチマークテストを行ってみると，Table 4 に示す通り，約 2 倍〜2.5 倍の性能差があることが分かる．

ベンチマーク結果には直接現れないが，メモリ保護機構のない GT640 ではグローバルメモリを不正アクセスするようなカーネル関数を記述すると途端に GPU がハングアップする．また，Tesla C2070 では現れない MAGMA の制限 (バグ？）により，規模の大きな行列の倍精度計算を行うとエラー表示を吐いてまともに実行できないことが度々あった．そのため開

Table 4: 陽的，陰的 Runge-Kutta 法の性能評価 (秒)(Tesla C2070 と GT640)

	ERK76-GPU		IRK36-GPU	
n	Tesla	GT640	Tesla	GT640
50	0.0046	0.0062	1.13	0.95
100	0.0051	0.0070	1.23	1.12
200	0.0062	0.012	2.30	2.35
500	0.012	0.021	6.99	9.48
1000	0.019	0.037	15.91	42.47

発にあたっては Tesla C2070 を主として使用し，比較用として GT640 を補助的に用いるようにした．このような実行性能以外のハードウェア環境の差も，GPU を用いた高性能計算を行う際には留意すべきである．

6. 結論と今後の課題

以上述べてきたように，大規模計算になればなるほど，CPU より GPU 上での計算が有利であることが今回の性能評価の結果，明確になった．しかし，小規模な問題に関しては GPU へのメモリ転送等がボトルネックになり，性能が発揮されないという問題も明らかになった．

今後は GPU 上での性能をより引き出すことを目的とした実装方法の探求を行っていきたい．特に陰的 Runge-Kutta 法に対しては，Tesla K20，GTX Titan 以上でサポートされる Dynamic Parallelization の機能が，特に Jacobi 行列から生成されるブロック行列の構築に対して有効に働くものと思われる．また，前述したような高速化のための仕組みの導入も必要であると思われる．これらの機能や高速化の手法を生かした実装を行っていく予定である．

謝辞

本研究を実施するに当たり，静岡理工科大学研究プロジェクト (B) の援助を受けた．使用した Tesla C2070 はこの援助によるものである．関係各位に厚く御礼申し上げる． また，秋田県立大学システム科学技術学部電子情報システム学科にてサバティカル滞在中，シミュレーション工学研究室から研究環境の提供を受けた．GT640 を組み込んだマシンはここで提供されたものである．サバティカル滞在に助力して頂いた秋田県立大学小澤一文教授，廣田千明准教授，中村真輔助教に感謝致します.

参考文献

1) E.Hairer. Radau5. http://www.unige.ch/~hairer/software.html.
2) Khronos Group. OpenCL. http://www.khronos.org/opencl/.
3) M. K. Jain. *Numerical Solution of Differential Equations*. Wiley Eastern Limited, second edition, 1987.
4) Tomonori Kouya. BNCpack. http://na-inet.jp/na/bnc/.
5) LAPACK. http://www.netlib.org/lapack/.
6) Intel Math Kernel Library. http://www.intel.com/software/products/mkl/.
7) L.O.Jay. Spark3. http://www.math.uiowa.edu/~ljay/SPARK3.html.
8) MAGMA. Matrix Algebra on GPU and Multicore Architectures. http://icl.cs.utk.edu/magma/.
9) NVIDIA. CUBLAS. https://developer.nvidia.com/cublas.
10) NVIDIA. CUDA toolkit. https://developer.nvidia.com/cuda-toolkit.
11) NVIDIA. What is CUDA. https://developer.nvidia.com/what-cuda.
12) ATLAS: Automatically Tuned Linear Algebra Software. http://math-atlas.sourceforge.net/.
13) BLAS: Basic Linear Algebra Subprograms. http://www.netlib.org/blas/.
14) 幸谷智紀. 倍精度と多倍長精度浮動小数点数を用いた反復改良法による連立一次方程式の高精度高速解法について. 日本応用数理学会論文誌, Vol. 19, No. 3, pp. 313–328, 2009-09-25.