From: Tsukasa NAKANO
To: Satoshi Okumura, "NAKASHIMA, Yoshito"
Cc: Kentaro UESUGI
Date: Thu, 01 Jul 2010 20:18:45 +0900
Subject: ECS-programs

おくむらさま、
なかしまさま、

GSJ/AIST のなかのです。先に報告した奥村くんの画像に対する処理で用いた最新版の ECS 用のプログラム群の説明をします。実は、Linux 計算機で走る通常 CPU 用のものと CUDA (Compute Unified Device Architecture == NVIDIA GPU) で走るものの両方を含む、ぼくが書いた ECS 用のプログラム群には以下の３つのバージョンのものがあります。

ecs_old (http://www-bl20.spring8.or.jp/~sp8ct/tmp/ecs_old.taz)

通常 CPU のシングルスレッドで ECS を実行するプログラム（および ECS の helper programs）は２年以上前に、それらのマルチスレッド版と CUDA 版は１年程前に書いた。以前に奥村くんに紹介したのはこれらの（ecs_old のシングルスレッド版の）プログラム。なお、こちらではシングルスレッド版のプログラムを使って複数の ECS の runs を１年近い期間に渡って実行した。

ecs_new (http://www-bl20.spring8.or.jp/~sp8ct/tmp/ecs_new.taz)

ecs_old のプログラムでは一定の電位の値（0 or 1）を与える画像表面上の位置が解くべき ECS の問題の本来の境界条件とは微妙に異なっていた(詳細は「仕様書」の PDF "ecs_*/doc/ecs.pdf" をご覧下さい)。それを ecs_new では修正した。ただし、きちんと調べたところ、この境界条件の違いに伴う値 Q0 (ECS で得た画像表面から流入・流出する電流の正規化した総量）の違いは１% 以下だった。それから、どうでも良いことかもしれませんが、ecs_new ではシングルスレッド版とマルチスレッド版のコードを ecs_old のものから整理した。

ecs (http://www-bl20.spring8.or.jp/~sp8ct/tmp/ecs.taz)

ecs_old と ecs_new の ECS 用のプログラムでは画像表面での境界条件が固定されていた。つまり、ecs_old では２・３次元画像のどちらに対しても x 方向に、ecs_new では２・３次元画像のそれぞれに対して y もしくは z 方向に単位の電位差を加えるようになっていた。そのため、これら以外の方向に電位差を加える ECS を行う場合は（プログラム si_rar などを使って）事前に画像を直角回転しておく必要があった。これ

に対して、最新版の ecs には ECS の境界条件として適用が可能な複数の電位差の方向のそれぞれに応じたプログラム群が用意されている。それから、これまでの CUDA 版の３次元 ECS 用プログラム群は x 方向の画素数が 1024 以下の画像しか取り扱えなかった。その制限を排した、任意の画素数の画像（ただし、GPU の「グローバルメモリ」に載る画像）を処理できる CUDA 版の３次元 ECS 用プログラム群も ecs に加えた。

追記（2010/8/14）：従来からある CUDA 版３次元 ECS 用プログラム（具体的には、後述する ecs_3d_[a,b]_[x,y,z]）で処理可能な画像が 1024 以下の x 方向の画素数のものだけなのは、これらを現在使用している NVIDIA Tesla C870 や C1060 などの（compute capability 1.x の）GPU で実行する場合の制約でした。今後導入する（compute capability 2.x の）Tesla C2050 や C2070 などでこれらのプログラムを実行すれば、x 方向の画素数が 2048 以下の画像を処理できます。以下の E-mails は未修正なので、このような処理可能な画像の x 方向の画素数に関する記述を適当に読み替えて下さい。

これらのうちの最新版の ecs のプログラム群を使って奥村くんの画像の処理を行いました。以下ではこのバージョンの ECS 用プログラム群の説明をします。

(1) プログラムのインストール

インストールに先立ち、ECS 用のプログラム群のソースファイルの類をまとめた以下の gzip 圧縮した TAR 形式書庫ファイルをダウンロードしておいて下さい。

http://www-bl20.spring8.or.jp/~sp8ct/tmp/ecs.taz

(1-1) Windows 用のプログラムのインストール

NVIDIA が無償提供している開発環境をインストールすれば Windows 上で CUDA 版の ECS 用プログラム群を扱うことが可能です。しかし、こちらにはその環境がないので、ここでは単に MinGW（Minimalist GNU for Windows）の GNU-C compiler でコンパイルした通常の 32 bit CPU 用の Windows の実行ファイル群をインストールする方法だけを書きます。それは以下のようにします。

[1] 適当な書庫ファイル解凍・展開用のプログラムを使って前記の書庫ファイル ecs.taz を Windows の HDD の適当な場所に解凍・展開する。

[2] Windows 用の実行ファイル "*.exe" はすべて展開されたディレクトリ ecs¥exe の中に入っている。これらをコマンドプロンプト（DOS 窓）で実行するために DOS 窓を開き（す

べてのプログラム→アクセサリ)、そこで以下のように入力して実行パスの登録を行う。

```
path　ドライブ名:¥何とかかんとか¥ecs¥exe;%path%
```

これで任意のディレクトリから ECS 用のプログラム群を起動できるが、DOS 窓を閉じるとその設定は無効になる。恒久的に設定したい場合は ...。もしくは、ecs¥exe の中のファイルすべてを実行パスに登録済の適当なディレクトリにコピーした方が良いかもしれない。

なお、これらの ecs¥exe の中の実行ファイル群はディレクトリ ecs¥src の中にあるファイル Makefile_ptw32 を使って以下のようにしてコンパイルしました。

```
make　-f　Makefile_ptw32　install.exe　clean
```

また、マルチスレッド版のプログラム群には「pthreads-win32」のライブラリを使いました。その利用法などは以下のページをご覧下さい（と言っても、ぼくはこの MinGW 用のライブラリのファイル名を Linux 用と同じになるようにインストールして、Makefile_ptw32 ではその名前のファイルを引用していますが)。

http://sourceware.org/pthreads-win32/

(1-2) Linux 計算機上でのインストール

CUDA 版プログラム群のインストールは後回しにします。それら以外の ECS 用のプログラム群のインストールは、gcc などのプログラム開発用ツール群がインストールしてあるなら、以下のように端末入力するだけです。

[1] 前記の書庫ファイルを解凍・展開する。

```
tar　xzf　ecs.taz
```

[2] 展開されたソースファイルから実行ファイルをコンパイルし、それらを展開された空のディレクトリ ecs/bin にコピーする。

```
cd　ecs/src　&&　make　install　clean　&&　cd ../..
```

ただし、この処理に使う ecs/src/Makefile ではコンパイルを gcc で行う設定になっている

（必要なら適当に書き換えて下さい)。

[3] ディレクトリ ecs/bin を実行パスに登録しておく。

login shell として sh もしくは bash を使っている場合

```
cd  ecs/bin  &&  PATH=`pwd`:$PATH  &&  cd ../..
```

csh もしくは tcsh を使っている場合

```
cd  ecs/bin  &&  set  path=(`pwd`  $path)  &&  cd ../..
```

もしくは、ecs/bin の中のファイルすべてを実行パスに登録済みの適当なディレクトリ（例えば、/usr/local/bin など）にコピーしても良い。

(1-3) Linux 計算機上での CUDA 版のプログラム群のインストール

CUDA 版のプログラム群のインストールには NVIDIA が無償提供している以下の３種類のソフトウェアが必要です。

GPU のデバイスドライバ（kernel driver）
: Linux のシステム（kernel）に組み込まれる。

CUDA toolkit
: 普通はディレクトリ /usr/local/cuda の下に格納されている。

CUDA Software Developer's Kit (SDK)
: 普通はユーザのホームディレクトリの下の NVIDIA_CUDA_SDK もしくは NVIDIA_GPU_Computing_SDK という名のディレクトリに格納されている。

これらのうち kernel driver の格納場所を気にする必要はないですが、残りのものがどこに格納されているかをあらかじめ知っておかねばなりません。前記の書庫ファイル ecs.taz に入っている CUDA 版プログラムのインストールに使うファイル ecs/cuda/Makefile には以下が埋め込まれています（ただし、下記の「${HOME}/」 はユーザのホームディレクトリを表しています)。

CUDA toolkit がインストールされている場所
: CUDA　　=/usr/local/cuda

CUDA SDK がインストールされている場所
: SDK　　=${HOME}/NVIDIA_CUDA_SDK

これらはSPring-8にあるNVIDIA Tesla C870とC1060の２台を搭載したLinux（CentOS4）計算機 sp8ct.spring8.or.jp に合わせた設定で、最新版の NVIDIA のソフトウェアをインストールした計算機では変更が必要なはずです。

sp8ct.spring8.or.jp の CUDA(toolkit と SDK)は ver. 2.0 です。ぼくのところの ver. 2.3 では SDK を NVIDIA_GPU_Computing_SDK という名のディレクトリにインストールするようになっていました。なお、NVIDA が無償提供している最新版の CUDA は ver. 3.? です。

と言うことで、ファイル ecs/cuda/Makefile を修正する必要かもしれませんが、CUDA 版プログラム群のとりあえずのインストール法を以下に書いておきます。

[1] 前記の書庫ファイルを解凍・展開する（展開・解凍済なら不要）。

```
tar  xzf  ecs.taz
```

[2] 展開されたソースファイルから実行ファイルをコンパイルし、それらを通常 CPU 版の場合と同じディレクトリ ecs/bin にコピーする。

```
cd  ecs/cuda  &&  make  install  clean  &&  cd ../..
```

[3] ディレクトリ ecs/bin を実行パスに登録する（登録済なら不要）。

以上のようにしてインストールした後、CUDA 版プログラムの起動の前にそれを実行する GPU を選択しないといけないかもしれません。

上杉くんの sp8ct.spring8.or.jp やぼくの計算機ではそうなっているのですが、これらにはCUDAプログラムが「とりあえず」走るNVIDIAの普通のグラフィックボード(Quadro NVS29?)が刺さっています。Tesla C1060でなくこちらを使うとメモリ不足でエラーが出ますが ...。

自分の計算機に装着されている CUDA プログラムが走る GPU を調べるには前記の SDK のプログラム deviceQuery を使います。以下は SDK がディレクトリ ${HOME}/NVIDIA_CUDA_SDK に格納されている場合のその起動法です。

```
${HOME}/NVIDIA_CUDA_SDK/bin/linux/release/deviceQuery
```

これで表示される "Tesla C1060" の「Devce 番号」を確認し、それが０でない場合には CUDA 用

の ECS プログラムの起動前に以下を入力して下さい。

login shell として sh もしくは bash を使っている場合
export CUDA_GPU=番号

csh もしくは tcsh を使っている場合
setenv CUDA_GPU 番号

すみません。長くなったので、残りは明日以降にします。以下のことを書こうと思っています。

(2) プログラムの起動法
(3) プログラムの動作テスト

From: Tsukasa NAKANO
To: Satoshi Okumura, "NAKASHIMA, Yoshito"
Cc: Kentaro UESUGI
Date: Thu, 08 Jul 2010 16:53:56 +0900
Subject: ECS-programs-2

おくむらさま、
なかしまさま、

GSJ/AIST のなかのです。ECS 用プログラムに関する先日の E-mail の続きです。

すみません。うっかりしていました。先の E-mai ではぼくが書いた ECS 用プログラム群のインストールの話から始めましたが、本当はその前にぼくが考えている ECS のきちんとした説明が必要でした。それは前記の書庫ファイル ecs.taz を解凍・展開して得られる PDF ファイル ecs/doc/ecs.pdf に記した通りですので、目を通しておいて下さい。

(2) プログラムの起動法

前記の書庫ファイル ecs.taz を解凍・展開して得られるディレクトリ ecs¥exe には 44 +1 個の Windows の実行ファイルが入っています。また、先の E-mail の説明通りにすれば、ディレクトリ ecs/bin に通常 CPU と CUDA GPU 用の合計 44 + 16 個の Linux の実行ファイルをインストールできるはずです。ここではこれらのプログラムのそれぞれの起動法などを説明します。

これらのうち ecs¥exe¥stop_watch.exe は後述する Windows 上での動作テストを行うバッチファイルが ECS の SOR 演算の処理時間を計測する際に使うプログラムですが、ここではその詳しい説明は省略します。

(2-1) ECS 用プログラムの概略

インストールした実行ファイルの名前はそれぞれの機能などを表わしています。具体的には、以下の５種類の機能を持つプログラムがあります（Windows の実行ファイルでは名前の最後に ".exe" が付きますが、ここではそれを省略しました)。

ecs_DIM_DIR_VER
: ECS の SOR 演算を指定した回数（Relaxation Iteration Count；RIC）だけ繰り返す（これが ECS 用の「メインプログラム」です)。

ilf_DIM_DIR
: SOR の初期値とする線形電位場（Initial Linear Field）を計算する。

cgm_DIM
: 本番の ECS の前処理として行う、縮小画像を使った ECS 用に電流の流路を表す画像（ mask 画像）を粗視化する（Coarse-Graining Mask)。ただし、この処理では元の流路のつながりが保持される（逆に言うと、流路のつながりを保つため、これで縮小した画像上の流路は「太る」; それゆえ、これを一般的な画像縮小プログラムとして使ってはいけない)。

rsd_DIM
: 縮小画像の ECS で得た電位場の画像（data 画像）から元々のサイズの画像（もしくは、先のものよりも「大きい」縮小画像）の ECS に使用するための初期電位場をリサンプリングする（Re-Sampling Data)。

afv_DIM_DIR
: ECS で得た電位場画像からその各画素における電流（正確には電位勾配）のベクトルの振幅もしくは絶対値（Amplitude of Flow Vector；AFV）を計算する（AFV の計算法については前記の書庫ファイル ecs.taz を解凍・展開して得られる PDF ファイル ecs/doc/ecs.pdf を見て下さい)。

ただし､上の実行ファイル名の文字列に含まれるプログラムの機能を表す小文字記号の後の､"_" で区切られた大文字記号のそれぞれの意味は以下の通りです。

DIM（実際の記号は "2d" もしくは "3d"）
: 取り扱う画像の「次元」を表す。

DIR（"x"、"y" もしくは "z"）

ECS の境界条件として画像の表面に与える電位の場の「方向」を表す。つまり、DIR = "x" なら x 軸に垂直な画像表面に電位０と１を与える。ただし、DIR = "z" は DIM = "3d"（３次元画像）の場合にのみ有効。

VER（実際の記号については後述）

ECS の SOR 演算で画素ごとに保持する電位の値の精度（ビット数）や SOR で使う画像のスキャン法などを表す記号（プログラムの「バージョン」）。これについては後でやや詳しく説明する。

これらの ECS 用プログラム群は TIFF 形式の画像ファイル(DIM = "2d" の場合)、もしくは TIFF 形式のスライス画像ファイルからなる３次元画像（DIM = "3d"）を入出力します。ただし、３次元画像の場合はスライス画像ファイルだけが格納されている、もしくはそれらを格納するための（事前に作成しておいた）ディレクトリを指定する必要があります。以下の３種類の画像を取り扱います。

mask 画像

ECS で電流の流路とする画素を指定する画像。０でない画素値の画素を流路と見なす。つまり、２値画像でなくてもかまわないが、それらの流路の画素は境界条件として電位０や１を与える画像の壁面につながっていなければならない（さもないとその流路の画素の電位が不定になる）。

data 画像

０～１の電位の値を正規化した画素値を格納する画像（電位場画像）。画素数（の構成）が同じ mask 画像と対にして使用する。ただし、その mask 画像上の画素値０の画素（流路ではない画素）に対応する画素における電位の値は不定である（実際には画素値０にしてあるが）。

AFV 画像

プログラム afv_DIM_DIR で計算した AFV（電流値）を格納する画像。

(2-2) SOR 演算用プログラム ecs_DIM_DIR_VER のバージョン（VER）

ぼくのプログラムでは画像上の流路の画素を伝った電流を支配する電位に関する Laplace 方程式（正確には、それを差分化した連立方程式）を SOR（Sucessive Over Relaxation；逐次過緩和）法で解いています。この方法（正確には、SOR 法を含む緩和法）では解くべき式を変形した「ある画素の値 = その周囲の画素の値の関数」に与えられた初期値を代入してそれぞれの画素の新しい値を計算します。その後、この新しい値を初期値として同じ処理を行い、計算すべきすべての画素の電位の値（電位場解）が収束するまでそれを繰り返すわけです。なお、SOR 法では

「新しい値 =(1-α) × その画素の以前の値 + α×その周囲の画素の値から得た値」とします。ただし、αは加速係数（Over Relaxation Factor ; １～２の値）で、ぼくが書いたプログラムでは α = 1.5 を採用しています。

問題はこの「新しい値」の計算に使うその画素の「以前の値」や「周囲の画素の値」の取り扱いです。画像（の上の流路の各画素）をスキャンしながら SOR 演算を行うわけですから、これには以下の両極端の方法が考えられます。

Jacobi 法
　それぞれの画像のスキャンの前に画像上の流路の画素の値すべてを保存しておき、それらを使って各画素の新しい値を計算する。
Gauss-Seidel 法
　同じ回の画像スキャンでそれまでに得た周囲の画素の新しい値と以前のままの値を区別せずに使って、それぞれの画素の新しい値を計算する。

これらには一長一短があります。まず、これらのどちらを使った場合も最終的な解の収束は保障されていますが、Gauss-Seidel 法に比べて Jacobi 法では解の収束が遅いと言われています。一方、これは微妙な話なので詳しく書きませんが、新しい値を並列計算する場合には Gauss-Seidel 法を使えません。そこで、プログラム ecs_DIM_DIR_VER の通常 CPU 用マルチスレッド版や CUDA GPU 版では Jacobi 法と Gauss-Seidel 法の中間に位置する「チェッカーボードスキャン」という手法で SOR 演算を行うようにしました。前記の書庫ファイル ecs.taz を解凍・展開して得られる PDF ファイル ecs/doc/sor.pdf にこの手法の具体的なことを記しておきましたので、ご覧下さい。なお、このようなチェッカーボードスキャンに対して普通の（Gauss-Seidel 法に近い）SOR 法で使っている画像のスキャン法を「単純ラスタスキャン」と呼びます。

以上のような（電位場解の収束の速さに関わる）SOR 演算の際の画像のスキャン法を変えた複数の版のプログラムを通常 CPU 用に用意しました。これに対して CUDA GPU 用にはチェッカーボードスキャンの版しかありません。

　チェッカーボードスキャンによる SOR はどちらかと言うと Jacobi 法に近い手法なので、単一コアだけの通常 CPU で SOR 演算を行う場合、それよりも単純ラスタースキャンを行う版のプログラムを使う方が電位場解の収束までに要する総処理時間を短縮できるはずです。

CUDA GPU 用にそのハードウェアの本質的な弱点である超多数の「コア」によるメモリ（正確には、global memory）のアクセスの競合を緩和する「結合メモリアクセス（coalesced memory access)」を実現したプログラムを用意しました。これに対して、GPU のコアが個々バラバラに

行う（効率の悪い）メモリアクセスの方法を「単純メモリアクセス」と呼びます。それを行う版の CUDA プログラムも比較のために用意しておきました。

それから、これもメモリアクセスの遅さに対応するための工夫ですが、CUDA GPU 用のプログラムでは各画素が流路であるか否かの１ビットの情報とそこでの電位のデータを４バイト（32 ビット）に「パック」してメモリ上に格納しています。

C 言語などと同様に CUDA でも４バイト(32 ビット)長の単精度浮動小数点数と８バイト(64 ビット）長の倍精度浮動小数点数を使えます。しかし、値域が０～１の電位のデータを格納するとそれらの「指数部」のメモリ領域が無駄になります。そこで、詳しい話は省略しますが、CUDA 版のプログラムでは電位の値を 29（DIM = "2d" の２次元画像の場合）もしくは 28 ビット（DIM = "3d"）長の整数に変換してメモリに格納するようにしました。

つまり、CUDA プログラムでは画像上の画素ごとに４バイトのメモリを使います。NVIDIA Tesla C1060 のメモリ（正確には、global memory）の容量は４ GB ですから、それで取り扱える画像の最大の画素数は 1024^3 になります。なお、通常 CPU 用のプログラムではこのような画素データのパックを行っていません。各画素が流路か否かの情報に１バイトのメモリを使っています。そして、メモリアクセスの速度と（SOR 演算の）計算精度の trade-off を知るため、電位の値を単精度浮動小数点数（値０～１を保持する「仮数部」は 23 ビット長）に格納する版と倍精度浮動小数点数（52 ビット長）に格納する版を用意しました。

３次元画像を処理する CUDA GPU 用プログラムには更なる版があります。前述のように CUDA プログラムではチェッカーボードスキャンによる SOR 演算の並列処理を行いますが、その際に x、y、z 方向の画素数がいずれも元の画像の値の半分の個数の「サブ画像」を処理の単位としています。ところが、ハードウェア上の制約により、x 方向の画素数（Hx）が 512 よりも多いサブ画像の各画素に GPU のコアを割り当てるような CUDA のコードを書くのが困難でした。そこで、Hx ≦ 512 のサブ画像（つまり、x 方向の画素数が 1024 以下の元の３次元画像）だけに対応するコード（正確には、前述の単純・結合メモリアクセスのそれぞれを行う２種類のコード）をまず書きました。その後、これらに加えて、z 方向の画素の SOR 演算を（並列ではなく）逐次計算することにより x 方向の画素数が 1024 以上の画像を取り扱えるようにした版のコードも書いてみました。こちらの版にも単純・結合メモリアクセスのそれぞれを行う２種類があります。

このようなプログラム ecs_DIM_DIR_VER の多数の版の一覧は以下の通りです。

通常 CPU 用のプログラムのバージョン（VER）
　以下は２・３次元画像用の両方がある。

VER	SOR scan	precision
0F	simple raster scan (SRS)	23 bits
0D	SRS	52 bits
1F	checker-board scan (CBS)	23 bits
1D	CBS	52 bits

以下は２次元画像用 (DIM = "2d")

VER	SOR scan	precision
2F	CBS by 2 threads	23 bits
2D	CBS by 2 threads	52 bits

以下は３次元画像用 (DIM = "3d")

VER	SOR scan	precision
4F	CBS by 4 threads	23 bits
4D	CBS by 4 threads	52 bits

CUDA GPU 用のプログラムのバージョン (VER)

以下は２次元画像用 (DIM = "2d")

VER	memory access	precision
a	simple memory access (SMA)	29 bits
b	coalesced memory access (CMA)	29 bits

以下は３次元画像用 (DIM = "3d")

VER	memory access	precision	remarks
a	SMA	28 bits	pixels along x ≦1024
b	CMA	28 bits	pixels along x ≦1024
c	SMA	28 bits	
d	CMA	28 bits	

注：これらはすべてチェッカーボードスキャンで SOR を並列処理する。

これらのうちのどの版を使えば電位場解が高速に収束するかは一概に言えません。例えば CUDA 版を使えば SOR の繰り返しを高速に実行できますが、通常 CPU 用の倍精度浮動小数点数を使ったものに比べると計算精度がやや低いので、電位場解が収束するまでには余分な繰り返しの回数が必要になるはずです。また、通常 CPU 用の単純ラスタスキャン版とチェッカーボードスキャン版に関しても同様で、さらに、後者のマルチスレッド版を実行する場合には使用する計算機に搭載されている CPU のコアの数が関係することは言うまでもありません。と言うわけで、「本番」の ECS の前に後述する動作テストを行ってみて、自分の計算機環境でどの版の ecs_DIM_DIR_VER を使えば良いかを検討して下さい。

(2-3) 個々のプログラムの起動法

ECS 用のプログラムはいずれも、以下で説明するパラメータを実行ファイル名の後に付加して端末(Windows ならコマンドプロンプト)から起動します。前述のように、これらが処理する mask、data および AFV 画像に関しては２次元画像なら TIFF 形式画像ファイル（の名前)、３次元画像なら TIFF 形式スライス画像用のディレクトリ（の名前）をパラメータとして起動時に指定します。ただし、３次元画像を出力するプログラムはいずれも、それ用の新しいディレクトリを作らないことに注意して下さい。それから、これらのプログラムでは起動後の端末からの入力はありませんが、プログラム ecs_* と afv_* ではテキストデータを端末（正確には、標準出力）に出力します。それらが必要なら適当なファイルに自分で「リダイレクト」して下さい。

プログラム ecs_*

起動法

```
ecs_DIM_DIR_VER  mask  old_data  RIC  new_data
```

説明

パラメータ mask で指定された mask 画像が指す流路を使った ECS の SOR 演算を old_data で指定された data 画像によって与えた電位場を初期値として RIC で指定された回数だけ繰り返し、結果の電位場画像を new_data に格納する。その後、SOR 演算で得た電位場に関する以下の５個の数値をタブコードで区切りでまとめた１行を端末に出力する。

[1] 画像の相対する壁面から流入・流出する電流の正規化した総量（Q1 と Q2）の平均値、Q0 = (Q1 + Q2) / 2

[2] それらの電流の総量の食い違い、dQ = |Q1 - Q2| / Q0

[3] old_data と new_data で異なる画素値の画素の個数

[4] それらの画素値の違いの平均値

[5] それらの画素値の違いの標準偏差

ilf_*

起動法（以下のいずれか）

ilf_DIM_DIR mask BPS data

ilf_2d_DIR Nx Ny BPS data

ilf_3d_DIR Nx Ny Nz BPS data

説明

パラメータ mask で指定された mask 画像の値と同じ画素数、もしくは Nx、Ny および Nz で指定された x、y および z 方向の画素数（の構成）の画像に壁面で境界条件として与えた電位０と１を単純に線形補間した電位場のデータを入れる。それを BPS で指定されたビット数の画素値の画像としてパラメータ data で指定された data 画像に格納する。

注

ぼくの ECS 用のプログラム群の中で data 画像の画素値のビット数を指定できるのは ilf_* だけです。

cgm_*

起動法（以下のいずれか）

cgm_2d old_mask Bx By new_mask

cgm_3d old_mask Bx By Bz new_mask

説明

mask 画像 old_mask の上の隣接した Bx×By もしくは Bx×By×Bz 個の画素のブロックを１画素に「縮小」した新しい mask 画像を作成し、new_mask に格納する。ただし、この縮小の処理では０でない画素値を持つ画素がひとつでもあればそのブロックを画素値１の画素に変換する。

rsd_*

起動法（以下のいずれか）

rsd_DIM old_data mask new_data

rsd_2d old_data Nx Ny new_data

rsd_3d old_data Nx Ny Nz new_data

説明

パラメータ mask で指定された mask 画像の値と同じ画素数、もしくは Nx、Ny および Nz で指定された x、y および z 方向の画素数（の構成）の新しい画像の各画素の画素値を old_data で指定された既存の電位場画像からリサンプリングし、それを new_data に格納する。

afv_*

起動法（以下のいずれか）

afv_DIM_DIR mask data BPS AFV

afv_DIM_DIR mask data largest BPS AFV

説明

パラメータ mask と data で指定された mask 画像と data 画像を使用してそれらの各画素における AFV（電流値）を計算する。それらの値を largest で指定された値、もしくはその指定が省略された場合には画像に出現した電流の最大値で正規化した後、BPS で指定されたビット数の画素値に変換してパラメータ AFV で指定された AFV 画像に格納する。なお、largest の指定の有無に関わらず、このプログラムは画像に出現した電流の最大値を端末（正確には、標準出力）に書き出す。

非常に長い E-mail になりました。今日のところはここまで。

From: Tsukasa NAKANO
To: Satoshi Okumura, "NAKASHIMA, Yoshito"
Cc: Kentaro UESUGI
Date: Fri, 09 Jul 2010 14:03:40 +0900
Subject: ECS-programs-3

おくむらさま、
なかしまさま、

GSJ/AIST のなかのです。ECS 用プログラム群に関する E-mail の続きです。

(3) プログラムの動作テスト

前述のように、多数のバージョンがある SOR 演算用プログラムのうちのどれで本番の ECS を行うべきかは、使用する計算機上でのそれらの動作テストの結果から判断すれば良いです。先の書庫ファイル ecs.taz を解凍・展開したディレクトリ ecs/test には以下の２×（2+3）個のテスト用のスクリプト（Linux 用の C-shell scripts と Windows のバッチファイル）が入っています。

sxct20_[x,y].csh（２個の Linux 用の C-shell scripts）

sxct20_[x,y].bat（２個の Windows のバッチファイル）

150×150 画素の２値画像 ecs/test/sxct20_m.tif を mask 画像として、その [x,y] 軸

と垂直な画像の壁面（辺）に境界条件として電位０と１を与えた ECS の SOR 演算を 10 万回繰り返す。

bead_4_[x,y,z].csh（３個の C-shell scripts）
bead_4_[x,y,z].bat（３個の バッチファイル）
ディレクトリ ecs/test/bead_4_m_1 に格納されている 128×128×128 画素の３次元２値画像を mask 画像として、その [x,y,z] 軸と垂直な画像の壁面に電位０と１を与えた ECS の SOR 演算を４万回繰り返す。

ただし、これらに関して以下のことにご注意下さい。

[1] これらの画像には画素値０が白、１が黒となる表示属性が付いています。
[2] これらのスクリプトはディレクトリ ecs/bin や ecs¥exe に ECS 用のプログラム群がインストールされていることを想定しています。そこにあった実行ファイルを別の場所に移動した場合は ...。
[3] CUDA GPU 用のプログラムは「Device 番号 =０」の GPU で走るようになっています。そうでない場合は前述の設定を事前に行って下さい。

上記の C-shell scripts はいずれも、ディレクトリ ecs/test に移動（cd）した後に「csh　スクリプトのファイル名」と入力すれば起動します。また、Windows のバッチファイルの起動法も同様で、コマンドプロンプト（DOS 窓）を開いて ecs¥test に移動した後、バッチファイルの名前を入力すれば OK です。

Windows のバッチファイルはファイルブラウザ（Explorer）でファイル名をクリックすることでも起動できます。しかし、その場合は処理の終了直後に DOS 窓が閉まってしまうので、そこに表示されているテキスト情報（後述）をじっくりと眺めることはできません。

テスト用のスクリプトはいずれもプログラム ilf_* をまず起動して、それぞれの mask 画像に適した初期値の線形電位場の data 画像を作ります。その後、ディレクトリ ecs/bin(C-shell script の場合）や ecs¥exe（バッチファイル）にインストールされているプログラム ecs_* のそれぞれ（SOR 演算用のすべての版のプログラムのそれぞれ）を順に起動して SOR 演算を行います。最後に、プログラム afv_* でそれぞれの結果の電位場に対応する AFV 画像を作ります。

指定した繰り返し回数が十分大きいので、テスト用のスクリプトで得られるどの電位場解もほぼ完全に収束しているようです。それゆえ、これらの電位場画像の観察によって SOR 演算用のプログラム群の個々の性能を評価するのは困難です。そこで、今回のテスト用スクリプト群では作成

した画像を即座に消去するようにしてあります。スクリプト群が端末に出力する以下のテキスト情報をもとにして SOR 演算用のプログラム群の性能を評価して下さい。

C-shell scripts の端末出力

SOR 演算用プログラムのバージョンごとに以下の５個の値がタブコード区切りで１行にまとめて出力される。

[1] バージョン名（VER）

[2] その版で得た電位場の画像の相対する壁面から流入・流出する電流の正規化した総量（Q1 と Q2）の平均値、Q0=(Q1+Q2)/2

[3] それらの電流の総量の食い違い、dQ=|Q1-Q2|/Q0

[4] その版の SOR 演算用プログラムの実行時間（秒）

[5] 得られた電位場画像から計算した AFV（電流）の最大値

バッチファイルの出力

SOR 演算用プログラムのバージョンごとに以下の４行が出力される。

１行目：バージョン名

２行目：前述のプログラム ecs_* が端末出力した（５個の値が含まれる）行

３行目：その版の SOR 演算用プログラムの実行時間（秒）

４行目：得られた電位場画像から計算した AFV（電流）の最大値

こちらの複数の計算機でこれらのテスト用のスクリプトを実行してみました。テキストファイル ecs/doc/ecs.txt にその結果を書き込んでおきましたので、ご覧ください。注目すべき点は以下の通りです。

[1] ECS の処理速度は用いたハードウェア（CPU や GPU）はもちろんですが、ソフトウェア環境（OS や C-compiler）にも依存します。

[2] ECS の SOR 演算の計算精度は前述の Q0 と dQ で評価できます。ただ、どの版のプログラムで得た Q0 も概ね同じ値（== 電位場解がほぼ完全に収束している）なので、ここでは dQ が重要です。メモリ上での電位の値の保持に使っているビット数と dQ の関係に注目して下さい。

[3] CUDA GPU 版プログラムの SOR 演算に使うビット数を変えたテストは、ここで紹介したものとは別のスクリプトで行いました。前述のように、ここでインストールした CUDA プログラム群で採用しているビット数は 29（２次元画像用）と 28（３次元）です。

まだ他にもありそうですが、今日のところはここまで。以上で予告していた ECS の話は終わりですが、これだけの情報では「本番」の ECS の実行に不十分だと思われるので、次の E-mail でそ

れに役立つ話をするつもりです。

From: Tsukasa NAKANO
To: Satoshi Okumura, "NAKASHIMA, Yoshito"
Cc: Kentaro UESUGI
Date: Wed, 14 Jul 2010 15:23:42 +0900
Subject: ECS-programs-4
Attached: install.bin.csh, run.ecs_m.csh, run.ecs_d.csh

おくむらさま、
なかしまさま、

GSJ/AIST のなかのです。ECS 用プログラム群に関する最後の E-mail です。

(4) 「本番」の ECS に役立つ tips

前述の動作テスト用のスクリプトでは元の mask 画像をそのまま使って ECS を行うようになっています。つまり、mask 画像との画素数の一致だけを考慮した単純な線形電位場を ECS の SOR 演算の初期電位場として使っていますが、それでは電位場解の収束が遅いので、本番の ECS では最終的な電位場解により近いものを初期値として使いたいです。それには、プログラム cgm_* で元のものを粗視化した（画素数を減らした）mask 画像を作成し、それを使って予備的な ECS を行い（その際に指定する初期電位場については後述）、その結果の（画素数の少ない）data 画像からプログラム rsd_* でリサンプリングして得たものを使えば良いです。この手法に関しては以前にも説明したので、ここではこれまでにぼくが実際に使って来たものと同等な SOR 演算用の C-shell scripts を紹介します。また、このような SOR 演算に加えて、本番の ECS ではその「前処理」や「後処理」も必要です。使用するプログラム群のインストール法を含めたこれらの処理の tips も以下で紹介します。

(4-1) 最新の３次元画像処理用プログラム群のインストール

前にも言ったと思いますが、ぼくは旧来の３次元画像処理用のプログラム群「slice*」（http://www-bl20.spring8.or.jp/slice/）の改良をやめました。

> slice* を廃した最大の理由は、それらの随所に埋め込んだ 32 ビット CPU を想定したコードの修正が難しかったことです（何せ、これを書き始めたのは 15 年以上前ですから ...）。また、TIFF 画像ファイルの I/O にメンテナンスが不十分なオープンソースのライブラリ

「libtiff」を使っていることも気になっていました。他にも色々あって、...。

それに代わるプログラム群「si_*」のコードを２年くらい前から書き始め、slice* の主なものに相当するプログラムは一応完成済です。si_* では C 言語の標準ライブラリ関数だけを使っているので、UNIX（Linux と MacOSX を含む）と Windows の（64 ビット CPU の）計算機上で実行できます。問題は si_* 用のマニュアル群で、これは例によって関係者への E-mails につらつらと書きましたが、まだ si_* のプログラムすべてをカバーしていません。これに関しては今後の課題にしたいと思います。ここでは、使用する si_* のプログラムのそれぞれについての必要最小限の説明だけを書くことにします。

前置きが長くなりましたが、このような si_* のプログラムのうちの主要なもののインストール法は簡単で、以下のようにするだけです。

Windows 用の実行ファイルのインストール

なかしまくん、

ここで紹介している Windows 用の実行ファイルは以前に紹介した所内共有ディスク geopub のディレクトリ tsukasa/nsc_100226 に置いてあるものと現時点では同一ですが、今後は geopub のものを更新しません。

[1] FTP サイト http://www-bl20.spring8.or.jp/~sp8ct/tmp/ に置いてある以下の zip 形式書庫ファイルを WEB ブラウザ（IE や FireFox）でダウンロード・保存し、ファイルブラウザ（Explorer）で開く（または、これらそれぞれを WEB ブラウザから直接開いても良い）。

mcl.zip、pvr.zip、mask.zip、trim.zip、osp.zip、rar.zip、cm.zip、
affine.zip、2cc.zip、t2ps.zip

[2] これらの中の実行ファイル"*.exe"のすべてを実行パスに登録済みの適当なディレクトリにコピーする。なお、これらの実行ファイルはこちらの（32 ビット CPU 計算機上の）MinGW でコンパイルした 32 ビット CPU 対応なので、（圧縮していない）画素値の容量が２GB 以上の３次元画像を処理できない場合もある。

Linux 計算機でのインストール

注

以下のインストールの処理はインターネットに接続している（Linux）計算機でのみ実

行可能。

[1] この E-mail に添付した C-shell scriipt "install.bin.csh"を適当な場所にコピーし、以下のように入力してそれを実行する（コピーした場所に移動して実行する場合は下記のディレクトリの指定は不要)。

csh　コピーしたディレクトリ/install.bin.csh

[2] これによりソースコード群の自動ダウンロードとコンパイルの処理が正常に行われたなら、カレントディレクトリに作成されたディレクトリ bin に Windows 用のものと同等な実行ファイルがインストールされているはず(32 ビット CPU 用になるか 64 ビット対応かは処理系に依存)。その後、これらのファイルすべてを実行パスに登録済みのディレクトリにコピーするか、ディレクトリ bin を実行パスに登録すればよい。

(4-2) ECS の前処理と後処理

ECS の SOR 演算の前に以下の処理を行うことが必須だと思われます。

３次元画像の画素値ヒストグラムの調査

ディレクトリ byte の中に３次元画像のスライス画像ファイルだけが入っている場合、以下の入力でその画素値ヒストグラムを取得できます。

```
si_pvr  byte  -  > byte.txt
```

これにより、テキストファイル byte.txt の各行に画素値とそれを持つ画素の個数がタブコード区切りで書き込まれます。このファイルは普通のエディタや MS-Excel で取り扱えます。また、UNIX の標準コマンド tr でそれを CSV 形式に変換すれば、OpenOffice-Calc（Linux で走る MS-Excel 互換ソフトウェア）でも処理可能です。

```
tr  ¥¥t  ,  < byte.txt  > byte.csv
```

画像のトリミング

３次元画像上の対角点の座標値が（x1,y1,z1）と（x2,y2,z2）の直方体領域を切り出したい時は以下のようにします（以下では、ディレクトリ byte に入っている画像から切り出した直方体領域のスライス画像ファイルを新たに作成したディレクトリ trim に格納しています)。

```
mkdir  trim
si_trim  byte  -  x1  y1  z1  x2  y2  z2  trim
```

流路の画素のクラスタラベリング

ディレクトリbyteに格納されている３次元画像上の画素値０～70 を持つ画素群を「空隙」と見なしてクラスタラベリングする場合、以下のようにします。

```
mkdir  mcl
si_mcl  byte  -  0-70  mcl  > mcl.txt
```

ただし、ここではディレクトリ mcl をまず作成し、そこに結果のクラスタ画像を格納しています。そして、テキストファイル mcl.txt の１行目には画像上に出現したクラスタの総数 C が、２～C+１行目の各行には以下の７個の整数値がタブコード区切りで書き込まれます。

[1] そのクラスタに属する画素の個数（クラスタサイズ）
[2-4] それがちょうどおさまる直方体の x、y、z 座標値の最小値
[5-7] それがちょうどおさまる直方体の x、y、z 座標値の最大値

mcl.txt に書き込まれたデータから２種類の情報を引き出せます。まず、２行目は「背景（空隙以外の部分）」の画素群の、３行目以降はサイズの降順で並んでいる個々の空隙のクラスタのデータなので、これらそれぞれのクラスタサイズを N2（背景）、N3（最大の空隙クラスタ）、N4、... として、

```
porosity = 1 - N2 / (N2 + N3 + N4 + ...)
connectivity = N3 / (N3 + N4 + ...)
```

となります。UNIX の標準コマンド awk を使えば、これらの値を以下のようにして計算できます（MS-Excel でも可能でしょうけれども ...）。

```
awk 'NR>=2 { N+=$1; if (NR==2) N2=$1; if (NR==3) N3=$1 } ¥
      END { printf "%f¥t%f¥n",1-N2/N,N3/(N-N2) }'  mcl.txt
```

また、mcl.txt を使って ECS の流路として適当なクラスタを識別できます。先にも書いたように、ECS の流路は境界条件として電位０と１を与える画像の壁につながっていな

ければなりません。mcl.txt の３行目以降に記されている空隙のクラスタがちょうどおさまる直方体の対角点の座標値を見れば、この条件を満たすクラスタの識別は容易です。なお、このようなクラスタの複数個を ECS の流路としてもかまいません。

クラスタ画像から mask 画像を作る

前記のプログラム si_mcl で作ったディレクトリ mcl に格納したクラスタ画像の各画素には mcl.txt の２行目以降の順番に応じたクラスタ番号（０～）が画素値として入っています。ただし、現在の設定では 16 ビット以下の画素値なので、mcl.txt の 65537 行目以降に相当するクラスタ番号（画素値）はすべて 65535 です。例えばクラスタ番号が１（最大のクラスタ）と３の画素群を ECS の流路としたければ、以下のようにしてクラスタ画像の画素値２と４～65535 のそれぞれを画素値０に置換した新しい mask 画像を作れば良いです(以下の例では事前に作成しておいたディレクトリ mask に mask 画像を格納しています)。

Linux の場合（行末の "¥" は継続行があることを意味します）

```
( echo  2  0  ;  echo  4  65535  0 )  |  ¥
si_pvr  mcl  -  mask > /dev/null
```

Windows の場合（行末の "^" は継続行があることを意味します）

```
( echo  2  0  &&  echo  4  65535  0 )  |  ^
si_pvr  mcl  -  mask  > nul
```

ECS の前処理はこんなものだと思います。次に、SOR 演算で得た電位場画像や AFV 画像に対する後処理ですが、これには大物として鳥瞰図の作成があります。しかし、その説明は長くなるのでまたの機会にします。ここではこれらの画像に電位や AFV（電流）の値に応じた色を付ける方法だけを示します。

画素値に応じた疑似カラー画像の作成

前記の書庫ファイル ecs.taz を解凍・展開して得られるディレクトリ ecs/etc にぼくがこれまでの ECS 処理で使って来た以下の４個の疑似カラーのデータファイル（テキストファイル）が入っています。

cm_rgb_b.txt（８ビット画素値の画像用）

cm_rgb_w.txt（16 ビット画素値の画像用）

画素値が０から最大値(255 もしくは 65535)で増加するにつれて青、シアン、緑、黄、赤、マジェンタの順に変化する色の情報が書き込まれている。具体

的には、各行に以下の４個の値がタブコード区切りで書き込まれている。

[1] 画素値もしくは画素値の範囲（see cm_rgb_w.txt）
[2-4] 色の R、G、B 成分の強度を表す０～255 の整数値

cm_inv_b.txt（８ビット画素値の画像用）
cm_inv_w.txt（16 ビット画素値の画像用）
上記の cm_rgb_*.txt で使っている色の順番を逆にしたデータ。

以下のようにすれば、これらの色データを（例えば）ディレクトリ afv に格納した 16 ビット画素値の AFV 画像に埋め込むことができます。

```
si_cm_rgb  afv  -  ecs/etc/cm_rgb_w.txt  afv
```

ただし、このような 16 ビット画素値の疑似カラー画像は通常の画像 viewer では処理できません（８ビット画素値のものなら可能ですが）。それを可能にしたいならフルカラー画像に変換しておく必要があります。

```
mkdir  afv_rgb
si_rgb  afv  -  afv_rgb
```

電位場画像のマスク処理

ECS では電流は流路以外の部分を流れないので、その画素に電流値０に相当する画素値０が入っている AFV 画像の色付けは上記の通りで問題なしです。しかし、電位場画像の流路以外の画素の電位は不定なので、それらは電位０のものとは別の色で表現すべきです。流路を示す mask 画像と 16 ビット画素値の data 画像を使って、流路の画素に対してはその電位の値に応じた前述の cm_rgb_*.txt の色を、また、それ以外の画素に対しては白を割り当てる手法は以下の通りです。ただし、"#"で始まる行はコメントで、直前の行の処理内容（ややトリッキーなものもあります）の説明を記しました。

```
mkdir  rgb
# 作業用および最終的な画像を入れるディレクトリ rgb を作成。
echo  0  65535  0  255  |  si_pvr  data  -  rgb  > /dev/null
# data 画像の画素値０～65535 を０～255 に置換（線形変換）。
si_mask  mask  -  1-  -256  rgb  -  rgb
# mask 画像上の画素値１以上の流路の画素には上で変換した data
```

```
# 画像の画素値（０～255）を、０の画素には画素値 256 を入れた。
( cat  ecs/etc/cm_rgb_b.txt ; ¥
  echo  256-  255  255  255 ) |  si_cm_rgb  rgb  -  -  rgb
# 画素値０～255 の画素には cm_rgb_b.txt の色の、画素値 256
# 以上の流路以外の画素には白（R=G=B=255）の情報を付加した。
# なお、Windows ではコマンド cat ではなく type を使えば良い。
# また、継続行などの書式も違うが、... (see previous example)。
si_rgb  rgb  -  rgb
# こうして得た疑似カラー画像は８ビットより大きい画素値なので、
# 後で取り扱いやすいようにフルカラー画像に変換した。
```

(4-3) 本番の ECS の SOR 演算に使える C-shell scripts

前述のように、ECS の SOR 演算では粗視化した mask 画像を使って行った予備的な ECS の結果の電位場をリサンプリングした data 画像を初期値として使うと解の収束が速くなります。ぼくがこれまでに行った ECS ではもう少し凝っていて、この粗視化・リサンプリングの処理を複数の「レベル」で行っています。つまり、元のもの（これをレベル０の画像と呼びます）の x、y、z 方向のそれぞれの画素数を 1/2、1/4、... にした mask 画像を用意し、まず、それらの最大レベルのもの（画素数が最も少ないもの）を使って（初期値に単純な線形電位場を与えた）SOR 演算を行います。その結果の電位場解からリサンプリングした data 画像を初期値として１段階下のレベルの計算を行い、...、を繰り返して元の（レベル０の）画像の SOR 演算まで実行するわけです。

この E-mail に添付した C-shell script "run.ecs_m.csh"は

```
csh  run.ecs_m.csh
```

と入力して起動します。これは ECS 用の新しいディレクトリ ecs を作った後、その下のディレクトリ ecs/m_[0-3] のそれぞれに４レベル分の粗視化した mask 画像を格納します。ただし、レベル０の mask 画像（ecs/m_0）はディレクトリ mcl に入っているクラスタ画像上の最大の空隙クラスタを流路の画素としたものです。その作成法は前述の通りで、run.ecs_m.csh の起動時にはクラスタ画像が入ったディレクトリ mcl がカレントディレクトリにないといけません。

この後、C-shell script "run.ecs_d.csh" を使って４レベルの SOR 演算の処理のそれぞれを順に行います。その起動法は以下の通りです。

最初に実行する場合、

```
csh  run.ecs_d.csh  > ecs.txt
```

中断した SOR 演算を再開する場合

```
csh  run.ecs_d.csh  >> ecs.txt
```

このように run.ecs_d.csh の実行を途中で止めてもかまいません。どのレベルの SOR 演算を次に行うべきかをこのスクリプトは自分で判断します。これが実行する具体的な処理内容は以下の通りです。

[1] まず（粗視化した）data 画像用のディレクトリ ecs/d_[0-3] の有無を調べて、次にどのレベルの画像の処理を行うかを ecs/d_[2,3] がない場合はレベル３、ecs/d_1 がなければ２、ecs/d_0 がなければ１という風に決める(ecs/d_0 があってもレベル０の処理を行う)。

[2] 最大レベル用のディレクトリ ecs/d_3 がない場合はそれを作成した後、ecs/m_3 に入っている mask 画像に応じた画素数の単純な線形電位場の data 画像をプログラム ilf_3d_z（後述）で合成し、そこに格納する。

[3] プログラム ecs_3d_z_b（CUDA 版；後述）を使ってそれぞれのレベルに応じた繰り返し回数（後述）の SOR 演算の「トライアル」を実行する。

[4] トライアルの前後の data 画像の画素値に違いがない「完全収束」した場合や、SOR 演算で得られた画像から流入･流出する電流の正規化した総量 dQ が指定したしきい値(後述)以下になった場合にはそのレベルの電位場解は収束したものと見なす。そうでない場合はレベルを保ったまま[3]の処理に飛ぶ。

[5] レベル０の電位場解が収束したなら、それですべての処理は終了である。そうでない場合は現在よりも１段階下のレベルで使う data 画像用のディレクトリを作成し、そこに現在のレベルの電位場解からリサンプリングして合成した初期電位場の画像を格納する。その後、１段階だけレベルを引き下げて[3]の処理に飛ぶ。

以上のようなスクリプト run.ecs_d.csh（および run.ecs_m.csh）が行う処理内容の一部はファイル（テキストファイル）上の設定の書き換えで変更できます。

使用するプログラム名

処理[2]と[3]に使う ECS 用の２個のプログラムの名前からわかるように、run.ecs_d.csh では z 軸に垂直な画像の壁面に電位０と１を与えた ECS を行います。ま

た、SOR 演算では CUDA 用の「最速版」のプログラムを使うようになっています。スクリプトに以下のように設定したプログラム名を書き換えれば、これらを変更できます。

```
set  ecs_3d = ecs_3d_z_b
set  ilf_3d = ilf_3d_z
```

注

run.ecs_d.csh は環境変数 CUDA_GPU を設定しません。その起動前に使用している計算機環境に応じた設定を行って下さい。

粗視化・リサンプリングの処理のレベルの総数

run.ecs_[m,d].csh ではレベルの総数を４にしてあります。これら両方のスクリプトの２行目に記した以下の設定を書き換えることによりそれを変更できます（run.ecs_[m,d].csh の両方に同じ値を設定して下さい）。

```
@  levels  =  4
```

SOR 演算の繰り返し回数

処理[3]で実行するそれぞれのトライアルの SOR 演算の繰り返し回数 RIC はレベルに応じた「RIC = 200000 / ２のレベル乗」にしています。つまり、レベル０に対する RIC は 20 万です。run.ecs_d.csh でこの値を以下のように設定していますので、必要なら書き換えて下さい。

```
@  ric0  =  200000
```

電位場解の収束を判定する際のしきい値

run.ecs_d.csh では各トライアルの電位場解の収束の判定に使う dQ のしきい値を以下のように（レベルによらず）0.01（１%）にしています。

```
set  limit = 0.01
```

電位場（data）画像のビット数

プログラム ilf_3d_* の起動パラメータとして 16 を指定していますが、これを変えても SOR 演算の処理速度は変わらないので、電位場解の完全収束を期さない限り今のままの値で問題ないでしょう。

さて、run.ecs_d.csh はそれぞれの SOR 演算のトライアル後にその概要を示す行を端末（正確には、標準出力）に書き出します。先の起動例ではそれをテキストファイル ecs.txt にリダイレクトしました。その各行にはタブコード区切りで以下の６もしくは８個の数値が並んでいます。

[1] 処理のレベル（３～０）
[2] SOR 演算の繰り返し回数 RIC
[3] 画像に流入・流出する電流の正規化した総量 Q0
[4] 画像に流入・流出する電流の総量の食い違い dQ
[5] SOR 演算の前後で異なる画素値になった data 画像上の画素の総数 N
[6] それらの画素値の違いの平均値（N≠０の場合のみ）
[7] それらの画素値の違いの標準偏差（N≠０の場合のみ）
[8] そのトライアルの SOR 演算の処理時間（秒）

(4-4) ECS を含む画像解析全般に関する重要なこと

これはぼくが重要だと思っていることですが、ECS に限らず（研究に使う）画像解析では実行したすべての処理の記録を残しておくべきです。特に、実際に入力した行を書き並べたスクリプトの形にしておけば、次回の処理に再利用できます。

とりあえずこれで ECS 用プログラム群に関する説明は終わりです。非常に長い E-mail になってすみません。

---

install.bin.csh

```
#!/bin/csh
mkdir bin
#
set awk=('{ if ((l=length)>0 && substr($0,l,1)=="¥¥")' ¥
             'printf "%s",substr($0,1,l-1);' ¥
           'else' ¥
             'print }')
#
foreach src (mcl pvr mask trim osp ovoid rar cm affine 2cc t2ps)
  wget -nv http://www-bl20.spring8.or.jp/¥~sp8ct/tmp/${src}.taz
  if ($status) exit 1
```

```
#
  tar xzf ${src}.taz
  rm -f ${src}.taz
  cd $src
  make
  mv `awk "$awk" Makefile | grep ¥^ALL | cut -d= -f2-` ../bin
  cd ..
  rm -rf $src
end
```

---

run.ecs_m.csh

```
#!/bin/csh
@ levels = 4
#
mkdir ecs ecs/m_0
echo 2 65535 0 | si_pvr mcl - ecs/m_0 >/dev/null
#
@ level = 1
while ($level < $levels)
  mkdir ecs/m_$level
  @ b = ( 1 << $level )
  cgm_3d ecs/m_0 $b $b $b ecs/m_$level
  @ level += 1
end
```

---

run.ecs_d.csh

```
#!/bin/csh
@ levels = 4
#
@ ric0 = 200000
set limit=0.01
#
```

```
set ecs_3d=ecs_3d_z_b
set ilf_3d=ilf_3d_z
#
@ level = 0
while ($level < $levels && !(-e ecs/d_$level))
  @ level += 1
end
if ($level == $levels) then
  @ level = $levels - 1
  mkdir ecs/d_$level
  $ilf_3d ecs/m_$level 16 ecs/d_$level
endif
#
while (1)
  set d=ecs/d_$level
#
  @ ric = ( $ric0 >> $level )
  while (1)
    set result=(`/usr/bin/time -f %e $ecs_3d ecs/m_$level $d $ric $d |& cat`)
    echo $level $ric $result | tr ' ' '¥t'
    if ($#result == 4 || `echo $limit $result | awk '{ print ($1>$3) }'`) break
  end
  if ($level == 0) break
#
  @ level -= 1
  mkdir ecs/d_$level
  rsd_3d $d ecs/m_$level ecs/d_$level
end
```

From: Tsukasa NAKANO
To: Satoshi Okumura, "NAKASHIMA, Yoshito"
Date: Mon, 13 Sep 2010 11:05:47 +0900
Subject: new-version-of-ECS-programs-for-CUDA

おくむらさま、
なかしまさま、

GSJ/AIST のなかのです。２および３次元 ECS（正確には、ECS の SOR 演算）用の CUDA プログラムの改造版を書きました。以下の３種類（計８個）です。

ecs_2d_bt_[x,y]
　　ecs_2d_b_[x,y]（従来の２次元 ECS 用プログラムの最速版） の改造版
ecs_3d_bt_[x,y,z]
　　ecs_3d_b_[x,y,z] （従来の３次元 ECS 用の最速版）の改造版
ecs_3d_dt_[x,y,z]
　　ecs_3d_d_[x,y,z]（x 方向の画素数 > 1024 に対応の最速版）の改造版

これらはいずれも CUDA GPU の「texture memory」という機能を使っています。

On Sat, 28 Aug 2010 17:41:20 +0900
Tsukasa NAKANO wrote for Kentaro UESUGI:
> 　　texture memory は CUDA GPU の主要なターゲットである CG で領域の
> 　　塗りつぶし（texture mapping）のために導入されたものらしいです。
> 　　GPU の祖先の VRAM（Video RAM）で使われていた CPU からのアクセス
> 　　とは独立に動作する画面表示用の DMA（Direct Memory Access）と類似
> 　　の機能と考えればよい？

新版のソースコードは従来のものと同じ以下の書庫ファイルに入れてあります。

http://www-bl20.spring8.or.jp/~sp8ct/tmp/ecs.taz

また、これを解凍・展開したディレクトリ ecs/test/ 中の C-shell scripts "sxct20_[x,y].csh" と "bead_4_[x,y,z].csh" も新版のプログラムをテストするように修正してあります。そして、これらで得た新版のプログラムの処理速度をテキストファイル ecs/doc/ecs.txt に書き込んでおきました。

ecs.txt からわかるように、２次元 ECS 用の新版(ver. bt)は従来の ver. b よりも低速です。これは CUDA の texture memory の設定に時間がかかるためだと思われます。一方、３次元用の ver. bt は従来の ver. b よりも 20% 程度、ver. dt は ver. d より５% 程度高速に動作するようです。

と言うことで、３次元 ECS に新版の ecs_3d_[b,d]t_[x,y,z] をお試しください。とり急ぎ、