計算科学ユニット2012年度第１回研究交流会，2012年４月27日

異なる計算機環境に対応した
球面調和関数変換ライブラリの構築

— poor man's computing —

石岡圭一（京大院・理）

E-mail: ishioka@gfd-dennou.org

Introduction

球面調和関数変換(Spherical Harmonics Transform: SHT):

スペクトル法を用いている大気大循環モデルには必須な要素.

ダイナモ計算のような他の分野でも基本的なツール.

私の研究興味: 基礎的な地球流体力学. ここでも当然 SHT は必須.

球面のスペクトル法の適用例: 球面上の２次元乱流の時間発展

回転球面上の２次元渦度方程式

$$\frac{\partial \zeta}{\partial t} + \frac{\partial \psi}{\partial \lambda}\frac{\partial \zeta}{\partial \mu} - \frac{\partial \psi}{\partial \mu}\frac{\partial \zeta}{\partial \lambda} + 2\Omega\frac{\partial \psi}{\partial \lambda} = D[\zeta]$$

$\zeta \equiv \nabla^2\psi$: **渦度,** ψ: **流線関数,**

Ω: **球の自転角速度,**

λ: **経度,** $\mu \equiv \sin\theta$, θ: **緯度,** t: **時刻,**

$D[\zeta]$: **高階粘性項.**

球は単位時間あたり, $\Omega/(2\pi)$ **回転.**

以下に見せる数値計算の解像度: T682 (2048×1024**グリッド).**

今日の講演では

SHTにはFFTのような(厳密な)高速変換が無いので，球面モデルの水平解像度が非常に細かくなると，SHTの計算量が非常に多くなる.

従って，計算機環境に合わせた SHTの効率的な実装が必要.

講演者は，これまでISPACKという数値ライブラリを開発してきた.
ISPACKには，以下の計算機環境に合わせて設計された SHTライブラリが含まれている.

ベクトル計算機，　IA-32 CPU，　GPU(Nvidia Tesla)

今日の講演では，これらの SHT ライブラリの設計の概要について解説.

ISPACK は下記URLでフリーソフトとして公開(ライセンス: LGPL)

http://www.gfd-dennou.org/library/ispack/ .

ISPACK は，地球流体電脳倶楽部の「SPMODEL: 階層的地球流体スペクトルモデル集」(http://www.gfd-dennou.org/library/spmodel/ **)のコアライブラリとして利用されている.**

今日の話で出す例は全て GFlopsレベルの計算で，現在のスーパーな計算からは程遠い.

しかし，研究者が誰でも買えるような比較的安価な計算機環境を最大限利用できるようにすることは重要だと考えているので，あえて私のこれまでの開発経緯を参考までに紹介したい.

まずは球面のスペクトル法についてのおさらい

偏微分方程式中の従属変数を球面調和関数で展開.

$$f(\lambda,\mu,t)=\sum_{n=0}^{M}\sum_{m=-n}^{n}a_n^m(t)Y_n^m(\lambda,\mu)$$

ここに，λ: 経度，$\mu=\sin\phi$，ϕ: 緯度，t: 時刻，M: 切断波数.

$$Y_n^m(\lambda,\mu)=P_n^{|m|}(\mu)e^{im\lambda}.$$

$P_n^m(\mu)$は ルジャンドル陪関数.

$$P_n^m(\mu)\equiv\sqrt{(2n+1)\frac{(n-m)!}{(n+m)!}\frac{1}{2^n n!}}\times(1-\mu^2)^{m/2}\frac{d^{n+m}}{d\mu^{n+m}}(\mu^2-1)^n\qquad(0\le m\le n).$$

ルジャンドル陪関数による展開

球面スペクトル法には球面調和関数変換が必要．そのうち計算量の大部分を占めるのはルジャンドル陪関数に関する以下の変換:

逆変換

$$g_j^m = \sum_{n=m}^{M} s_n^m P_n^m(\mu_j) \quad (m = 0, \cdots, M;\ \ j = 1, \cdots, J)$$

正変換

$$s_n^m = \sum_{j=1}^{J} w_j g_j^m P_n^m(\mu_j) \quad (m = 0, \cdots, M;\ \ n = m, \cdots, M)$$

ここに，J: ガウス緯度の個数，μ_j: ガウス緯度，w_j: ガウス重み.

必要な計算量

$m \geq 1$に対しては実部・虚部を扱わねばならないことと，ルジャンドル陪関数の対称性から添字jに関する計算は半分の$J/2$でよいことを考慮すると，逆変換・正変換に必要な計算量はNはルジャンドル陪関数をすべて事前に計算しておくとしても(乗算と加算が必要なことを考慮して)

$$N = \frac{J}{2} \cdot (M+1)^2 \cdot 2 = J(M+1)^2.$$

さらに，正変換時の数値計算で誤差が出ないことを保証するためには
$J > \frac{3}{2}M$ としておく必要があるので，

$$N \sim \frac{3}{2}M^3.$$

Mが大のときは，この変換計算をいかに効率化するかが全体の計算スピードを決める．

ベクトル計算機での高速化

ベクトル計算機の能力を発揮させるためには，ベクトル長(最内側 DO-LOOP の長さ)を十分に長くとる必要がある.

最も素直にプログラミングすると，逆変換・正変換の計算はともに最内側DO-LOOPの DO変数は ガウス緯度の添字jにとるのが自然.

しかし，この方針だと効率面で2つ問題がある:

1. ベクトル長は $\frac{J}{2} = \frac{3}{4}M$ になるが，M が十分大でなければベクトル長が短くなってVPの能力が十分発揮されない. ($M = 170$でもベクトル長$= 128$)

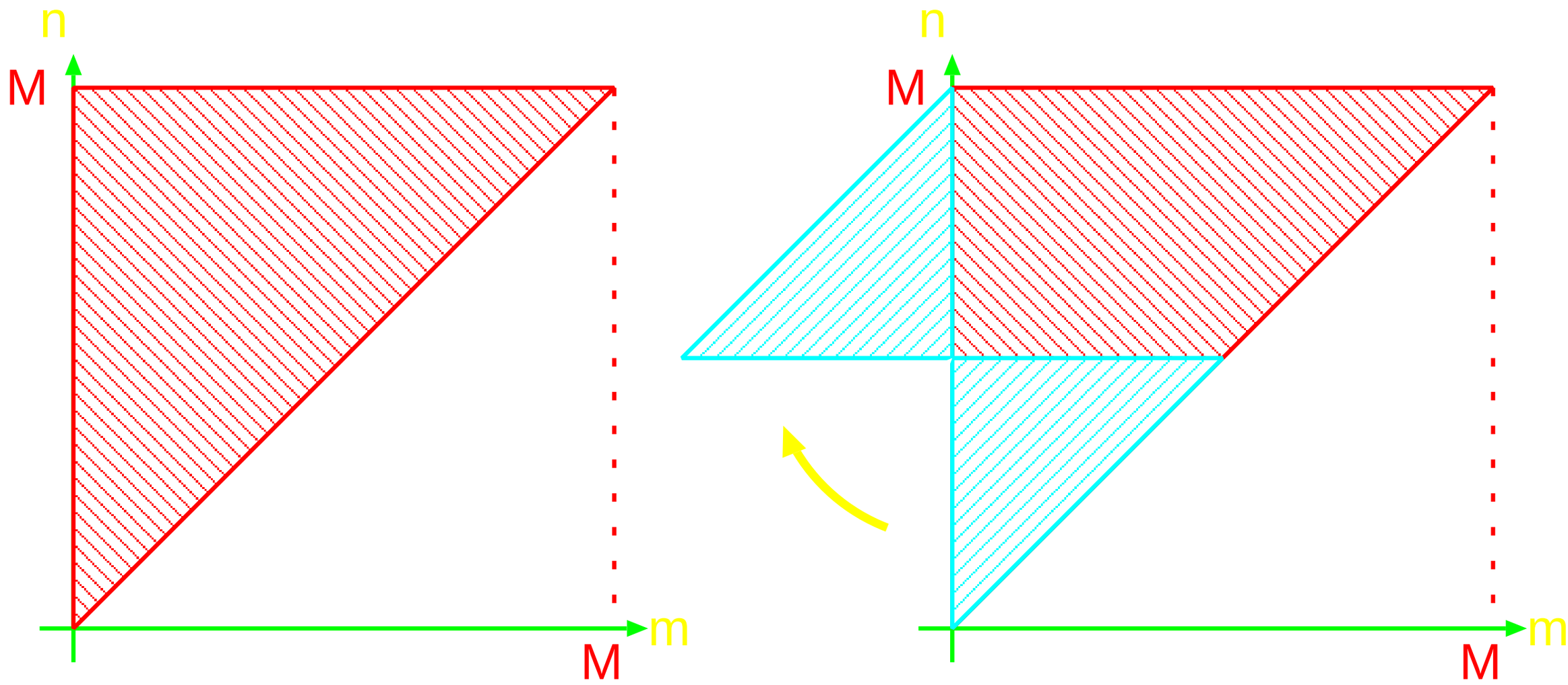
2. DO変数をjにとると，正変換の方の計算が内積型になり，逆変換の方の計算に比べて計算スピードが落ちる(約半分程度).

→ breakthrough: DO変数をmにとる.

そうすると，まず上記2の問題はクリアされる．また，ベクトル長はMとより長くなり，さらに複数種類の変数を同時に変換する場合には，ベクトル長は $M \times$(変数の種類) となり，一気にベクトル長が長くとれるようになる.

ただし，mをＤＯ変数にとるには工夫が必要(そのままのコーディングでは，各nに対するベクトル長はnと可変になってしまう).

図のように三角切断の一部を折り返すことによってmに関するＤＯ文のベクトル長を均一にMにする.



IA-32 CPUでの高速化

今日，IA-32 CPU はユビキタスである．ゆえに，IA-32 CPU用に最適化したコードを書くことは重要.

IA-32 CPUでは，(ベクトル計算機に比べて) メモリバンド幅が狭い．なので，キャッシュメモリを効率的に利用することが必須.

しかし，SHTは「行列$\times$ベクトル」型の演算なので，キャッシュが効きにくい.

$\rightarrow$ breakthrough: ルジャンドル陪関数を変換と同時に計算する.

ルジャンドル陪関数を計算する計算量は変換そのものの計算量と同じオーダー．なので，ルジャンドル陪関数の計算コストを下げられる工夫が可能ならやるべき．ルジャンドル陪関数は，以下の漸化式で計算される．

$$P_{n+1}^m(\mu) = (\mu P_n^m(\mu) - \epsilon_n^m P_{n-1}^m(\mu))/\epsilon_{n+1}^m$$

$\epsilon_n^m = \sqrt{(n^2-m^2)/(4n^2-1)}$． この漸化式を１回計算するには，$\epsilon_n^m$および$1/\epsilon_n^m$を事前に計算しておくとすれば，乗算３回と減算１回． これ以上減らしようが無いようにも見えるが，

$$P_n^m(\mu) = \alpha_n^m p_n^m(\mu)$$

とα_n^m，p_n^mを新に導入し，α_n^mをうまく選ぶことによって，

$$p_{n+1}^m(\mu) = \beta_n^m \mu p_n^m(\mu) + p_{n-1}^m(\mu)$$

とすることができる． こうすれば乗算２回と減算１回となり，乗算を１回削減できる．

さらに，Pentium4以降の x86系プロセッサで高速化を目指すにはSSE2 (Streaming SIMD Extension 2) 命令を最大限活用する必要がある．最近ではコンパイラもそれなりに賢くなってきてはいるが，SSE2命令を活用しようとするならば，アセンブリ言語でのプログラミングは避けて通れない．

ということで，ルジャンドル陪関数変換のコアの部分はアセンブリ言語で書いて最適化しておくべき．

ホットスポットの例(逆変換の一部分):

```
      SUBROUTINE LJLSWG(JH,S,R,Y,QA,QB,W)

      REAL*8 W(JH,2),Y(JH),QA(JH),QB(JH),S(2),R

      DO J=1,JH
        W(J,1)=W(J,1)+S(1)*QA(J) ! 展開係数から格子点値へ(実部)
        W(J,2)=W(J,2)+S(2)*QA(J) ! 展開係数から格子点値へ(虚部)
        QB(J)=QB(J)+R*Y(J)*QA(J) ! 漸化式
      END DO

      END
```

このような短いプログラムならアセンブリ言語で書き換えるのは容易.

SSE2命令を使って最適化した例

```
.globl ljlswg_
ljlswg_:
        movl    (%rdi),%edi
        movhpd  (%rsi),%xmm0
        movlpd  (%rsi),%xmm0
        movhpd 8(%rsi),%xmm6
        movlpd 8(%rsi),%xmm6
        movhpd  (%rdx),%xmm1
        movlpd  (%rdx),%xmm1
        movq   8(%rsp),%r10
        shlq  $3,%rdi
        movq %rcx,%rsi
        addq %rdi,%rsi
        movq %r10,%r11
        addq %rdi,%r11
.L0:    movapd (%rcx),%xmm4
        movapd  (%r8),%xmm2
        movapd  (%r9),%xmm3
        movapd (%r10),%xmm5
        movapd (%r11),%xmm8
        movapd %xmm0, %xmm7
        mulpd  %xmm1,%xmm4
        mulpd  %xmm2,%xmm4
        addpd  %xmm4,%xmm3
        mulpd  %xmm2,%xmm7
        addpd  %xmm7,%xmm5
        mulpd  %xmm6,%xmm2
        addpd  %xmm2,%xmm8
        movapd %xmm3,(%r9)
        movapd %xmm5,(%r10)
        movapd %xmm8,(%r11)
        addq $16,%rcx
        addq $16,%r8
        addq $16,%r9
        addq $16,%r10
        addq $16,%r11
        cmpq %rcx,%rsi
        jne .L0
        ret
```

ベンチマーク

私のLJPACKと，東大理・情報理工の須田礼仁氏の FLTSS:
http://olab.is.s.u-tokyo.ac.jp/~reiji/fltss/ との比較.
(FLTSSは高速多重極変換のアルゴリズムを用いている.
計算オーダーは，$O(M^2)$.)

以下の FLOPS の値は，公平な比較のためになお，実際の計算量(ルジャンドル陪関数の毎回計算を含む)ではなく，ルジャンドル陪関数をもし事前に計算してメモリに格納していたとする場合の変換そのものに絶対必要な計算量で算出している．すなわち，

$$J(M+1)^2/(\text{1回の変換計算に要した時間}).$$

ベンチマーク結果

テスト環境:

Intel Core2 Duo Xeon 3.0GHz (Woodcrest) 64bit mode

(L1: 32KB, L2: 4MB)

(理論性能: 12GFLOPS/コア)

Compiler: ifort -O3

解像度設定 T170(M＝170, J＝256)のとき

LJPACK(私の): 3.1 GFlops

FLTSS(須田氏の): 1.3 GFlops

解像度設定 T341(M＝341, J＝512)のとき

LJPACK(私の): 3.1 GFlops

FLTSS(須田氏の): 1.6 GFlops

AVX命令を用いた最適化

昨年の１月に発売が開始された Intel の Sandy Bridge アーキテクチャの CPUでは AVX (Advanced Vector eXtensions) 命令が新たに使えるようになった. (4月24日にさらに次世代の Ivy Bridge もリリースされたばかりだが，AVXが使えるという点は一緒).

AVX命令の主な特徴:

- 256bit 長の SIMD命令が使えるので，倍精度変数４個に対する処理が一括でできる. それにより，128bit長の SSE2命令に比べると理論的には倍速で処理ができる.
- 四則演算などに ３オペランド命令が使えるので，`C = A + B` のような形の計算がレジスタの余分なコピーなしで可能.

先程と同じ逆変換のホットスポットを考える.

```
      SUBROUTINE LJLSWG(JH,S,R,Y,QA,QB,W)

      REAL*8 W(JH,2),Y(JH),QA(JH),QB(JH),S(2),R

      DO J=1,JH
        W(J,1)=W(J,1)+S(1)*QA(J) ! 展開係数から格子点値へ(実部)
        W(J,2)=W(J,2)+S(2)*QA(J) ! 展開係数から格子点値へ(虚部)
        QB(J)=QB(J)+R*Y(J)*QA(J) ! 漸化式
      END DO

      END
```

AVX命令を使って最適化した例

```
.globl ljlswg_
ljlswg_:
  movl (%rdi), %edi
  vbroadcastsd  (%rsi),%ymm0
  vbroadcastsd 8(%rsi),%ymm5
  vbroadcastsd  (%rdx),%ymm1
  movq  8(%rsp),%r10
  shlq $3,%rdi
  xorq %rdx,%rdx
  subq %rdi,%rdx
  movq %r10,%r11
  addq %rdi,%r11
  subq %rdx,%r8
  subq %rdx,%r9
  subq %rdx,%r10
  subq %rdx,%r11
  subq %rdx,%rcx
.L0:
  vmulpd (%rcx,%rdx),%ymm1,%ymm4
  vmovapd (%r8,%rdx),%ymm2
  vmulpd %ymm2,%ymm4,%ymm4
  vmulpd %ymm0,%ymm2,%ymm3
  vmulpd %ymm5,%ymm2,%ymm2
  vaddpd  (%r9,%rdx),%ymm4,%ymm4
  vaddpd (%r10,%rdx),%ymm3,%ymm3
  vaddpd (%r11,%rdx),%ymm2,%ymm2
  vmovapd %ymm4,(%r9,%rdx)
  vmovapd %ymm3,(%r10,%rdx)
  vmovapd %ymm2,(%r11,%rdx)
  addq $32,%rdx
  jnz .L0
  ret
```

ベンチマーク結果

テスト環境:

CPU: Intel Core-i7 2620M(Sandy Bridge 2.7GHz,

ターボブースト時 3.4GHz, 理論性能: 27.2GFLOPS/コア)

OS: Debian squeeze 64bit,

Compiler: ifort 12.0.5, option xAVX

解像度設定: T170 (M=170, J=256),

逆変換と正変換の合計

SJPACK(with AVX): 6.7 GFlops
SJPACK(with SSE2): 5.1 GFlops

なぜAVXバージョンがSSE2バージョンの倍速にならないのか?

理由:

- **FFTなど，変換全体をAVX対応に書き換えたわけではない.**

- **また，AVX命令を使っても，load/store の部分がSSE2命令に比べて速くなるわけではない．SSE2版では乗算のところが律速になっているが，AVX版では乗算の部分が倍速になっている分，load/storeが律速になり，その関係で性能向上が十分得られていない.**

細かいことを言えば，Nehalem 世代にくらべて，Sandy Bridge 世代では，load ユニットが2倍になったが，store の方はそのまま.

GPGPUでの高速化

GPGPU = General Purpose GPU

GPU = Graphics Processing Unit

本来グラフィクスの処理に用いられていたグラフィクスボードを数値計算に利用できるようにしたもの.

GPU は CPU 程多機能ではないが，単純な計算を大量に処理するのには向いている． 昔は GPU上で数値計算しようとすると特殊な言語を使わなければならず敷居が高かったが，Nvidia 社が CUDA というC言語に類似した開発環境を数年前に公開して一気に敷居が下がり，GPUでの数値計算が広まった.

なぜ GPU計算?

メリット:

- **ピーク演算能力が高い．例えば，Nvidia Tesla C2050 なら倍精度 515GFlops．一方，CPUでは，同時期の Intel Xeon として，X5680 (Westmere-EP, 3.33GHz) を6×2＝12コア並列かけたとしても，160GFlops.**

現在は，Sandy Bridge EP 2687W(8コア 3.1GHz) で，AVXをフル活用して，かつ 16コア並列をかけたとすると，理論性能は，3.1×8×16＝396.8GFlops となり，かなりハイスペックになっている．ただし，Tesla の方も最新の M2090 なら理論性能は 665GFlops.

なぜ GPU計算?

メリット(つづき):

- メモリバンド幅が広い(ただし，このメモリは GPU のデバイス上のグローバルメモリで CPUに繋がっているメインメモリとは異なる).
C2050 では 144GB/s (M2090 では 177GB/s). CPU では，Xeon X5680 でも 32GB/s × 2 = 64GB/s. (Sandy Bridge EP は，51.2GB/s × 2 = 102.4GB/s なので，差は縮まってる.)

- GPUは価格が比較的安い. 例えば，Xeon X5680 を2つ搭載しようとすると，CPU価格だけで 30万円を超えるが，C2050 なら 20万円程度. また，最高スペックの GPU でなければ，数万円から手に入る.

- 電力消費が比較的少ない. C2050 は TDP 238W だが，Xeon X5680 × 2 だと TDP 260W.

なぜ GPU計算?

デメリット:

- GPU で計算を行うためには，必要なデータをメインメモリから転送して計算結果をまたメインメモリに戻す必要がある．その際はPCI-Express 2.0 x16 なバス経由となるので，8GB/s の帯域しかない．なので，メインメモリとの通信量が多いとそこが律速になってしまう．

- GPU上のグローバルメモリは上述のようにそれなりに広い帯域幅を持つが，大きさはメインメモリに比べると限られている． C2050 では 3GB，M2090 でも 6GB．

- ピーク性能は高いといっても，比較的非力なプロセッサの多数の集合体としての性能であるので，それらを活用するにはそれなりにプログラムに工夫が必要．CUDA 開発環境でそれなりには敷居が下がっているけど．

GPUの構造(Nvidia Tesla C2050 の例)

- 32個のSP(ストリーミングプロセッサ)の集合体が1個のSM(ストリーミングマルチプロセッサ)を構成し，それが 14個ある． なので，合計のSP 数は 448個.

- 各SM からは 3GBのグローバルメモリが共有で使える.

- 各SM内には 64kB のシェアードメモリがあり，これはそのSM 内の各SPで共有に使え，これは非常に高速アクセス可能．(ただし，64kBの領域の一部を L1キャッシュとして使うので，通常設定ではシェアードメモリは 48kB)．このシェアードメモリは各 SM で独立しているので，SMをまたがったデータのやりとりには使えない.

- というわけで，Xeon に例えるなら，32コアのCPUが14個挿さっていてそれらがメインメモリを共有しているようなイメージ.

CUDAプログラミングモデル

- GPU上の多数のSPを並列して実行させるために，マルチスレッドな処理を行う．

- SPがまとまってSMを形成しているという構造に対応して，CUDAでは複数のスレッドの集合体であるブロックという概念がある．各ブロックは1個のSMで実行される(複数のSMで実行が分割されることはない．逆に，1個のSMが同時に複数のブロックを実行することはもちろんある)．実行される全スレッド数は，

    1ブロックあたりのスレッド数 × ブロック数.

- 基本的にスレッド並列なので，OpenMP のプログラムを書いたことがあれば，並列化の概念を理解するのはそんなに難しくない．

球面調和関数変換のCUDAでの実装

- 計算の遅延などを隠蔽するために，各SMあたりのスレッド数はある程度大きくないといけない(できれば 128以上など)．また，14個あるSMを有効活用するには，ブロック数は少なくとも14以上でないといけない．

- というわけで，ルジャンドル陪関数変換において，各mの変換をブロックとし(なので，例えば T682 ならブロック数は 683)，ガウス緯度方向jを各ブロック内のスレッドとする(なので，T682 の普通の設定なら，ブロックあたりのスレッド数は 512)．

- 球面調和関数の計算に使う配列など，複数回の変換に共通して用いられるものはGPU側に最初にロードしておき，使い回す．これによってデータ転送量はある程度は減らせる．

※ 誰が実装してもまあそうなると思うけど..

実装例(sjpack-cuda/src/sjw.cu より)

- 逆変換の下請けルーチン(sjws2g_)より抜粋

```
cudaMemcpy(wsd,ws,sizews,cudaMemcpyHostToDevice);
  /* GPU へ展開係数の配列を転送 */

sjws2g_kernel<<<(*mm+1),jh>>>(pd,rd,wsd,gd,
                              *im,jh,*mm,*nm,*nn,*ipow);
  /* ルジャンドル陪関数変換を行う関数の呼出し.
     これは GPU で計算される. (*mm+1) の部分がブロック数,
     jh の部分がブロックあたりのスレッド数を与える. */

cudaMemcpy(g,gd,sizeg,cudaMemcpyDeviceToHost);
  /* GPU から変換結果の配列を転送 */
```

実装例(sjpack-cuda/src/sjw.cu より)

- 逆変換のカーネル部分(sjws2g_kernel)より抜粋

```
__global__ void sjws2g_kernel(double *p, double *r,
    double *ws, double *g,
    int im, int jh, int mm, int nm, int nn, int ipow)
{
    int m=blockIdx.x; /* ブロック番号を m に対応させる */
    int j=threadIdx.x;  /* スレッド番号を j に対応させる */
    if(m == 0)
      {
        /* 帯状成分の処理 */
      }
    else
      {
        /* 帯状成分以外の処理 */
      }
}
```

ベンチマーク結果

解像度設定: T682 (M=682, J=1024), 逆変換 + 正変換

テスト環境:

CPU: Intel Xeon X5670 (Westmere-EP, 2.93GHz)

GPU: Nvidia Tesla C2050,

Compiler: ifort 11.1 + nvcc release 3.1 V0.2.1221

CPU(1コア):	4.02 GFlops
GPU(データ転送コスト含む):	21.8 GFlops
CPU(OpenMP 並列化で12コア使った場合):	41.7 GFlops

ベンチマーク結果の評価

ということで，CPU 1コアとの比較ではGPU版が5倍速であるが，CPU 12コア並列との比較では GPU版が負ける．

ただし，FFTもGPU内で計算し，かつCPUとの通信を止めてGPU内で閉じた計算にすれば，GPU版はこの数字の3倍，すなわち 60GFlops 程度にはなり，それなら CPU 12コア並列を上回る．

なお，ルジャンドル陪関数自体の計算コストも含むトータルのFlops は上記の数字の 1.8倍程度なので，GPU内の実効性能は100GFlops は出ている．が，それでも理論性能の 1/5 程度．

GPUで今一つ性能が出ないのはなぜ?

1．正変換の部分が総和(reduction)型の演算になっているが，このような計算は並列計算では2分木を逆に辿るようにして計算するしかなく，遊んでしまう SPが出る分，効率が出ない.

2．そもそも，Fermiアーキテクチャには倍精度演算命令とload/store命令が同時発行できないという致命的弱点がある．なので，全くメモリアクセスをせずレジスタの中だけで計算が閉じているようなありえない状況でしか理論性能は出ない．行列積のようにレジスタをかなり使い回せるような場合を想定しても，理論性能の 2/3 が限界.

GPUで今一つ性能が出ないのはなぜ?

2(つづき). ということで, C2050 の理論性能は 343GFlops 程度と考えるべき. C2050 の理論性能を515GFlops として売ってるのは誇大広告だと思う. 私の知る限り, C2050上で最もパフォーマンスを出している実用的なプログラムは MAGMA BLAS の DGEMM で, それでも高々 300GFlops.

このあたり, 3月末に発表された Kepler アーキテクチャで改善されていることを期待したい.

GPUで今一つ性能が出ないのはなぜ?

3. さらに，理論性能は FMA = Fused Multiply-Add という積和がペアになった命令を使う場合を想定しているので，積和がペアになっていないような演算もある場合には性能はさらに下がる．これも考慮すると，理論性能の 1/3 程度，すなわち 172GFlops が出ていればいい方という感じ．そうなると，Westmere-EP で 12コア並列かけた場合と比べて大差ない.

※ で，実際，sjpack-cuda の逆変換の kernel 部分では 160GFlops 程度は出てしまっているので，プログラムいじってもこれ以上あまり性能向上が期待できる感じはしない.

まとめ:

私が開発している ISPACKという数値ライブラリに含まれるSHTライブラリのいろいろな計算機環境に合わせたデザインについて概説した.

- ルジャンドル陪関数を毎回変換と同時に計算するとキャッシュメモリを有効に使える.

- IA-32 CPU では，計算のホットスポットにおいて，SSE2命令やAVX命令を有効活用するためにアセンブリ言語でのコーディングが有効である.

- GPUを利用した変換プログラムも開発し，それなりのパフォーマンスは出ている. ただし，CPUとGPU間のデータ転送コストの問題や，またGPU自体の弱点など，まだ解決すべき問題は多い.