GPGPUと宇宙シミュレーション

村主　崇行*

Use of GPGPUs in Simulational Astrophysics

Takayuki Muranushi*

Key words: Astrophysics, Hydrodynamics, Accelerator

1. GPGPUとは

私は宇宙物理学についても，シミュレーションについてもまだまだ駆け出しの身ながら，この記事の執筆依頼をいただいた．開きなおってろくに勉強もせず知りえた範囲内のことを書くので，どうかこれからの成長を温かく見守っていただければ幸いである．

ほんの数年前までは存在すらしなかったGPGPU（General Purpose computing on Graphic Processing Unit）という単語を，いま多くのシミュレーション学会員が耳にしているのではないだろうか．GPU（Graphic Processing Unit）とは元来コンピューターの画面を描画するための専用ハードウェアである．その並列演算性能に着目し，GPUで汎用的な計算を行おうという技術がGPGPUである．

むろん，並列計算というアイデアの歴史はずっと古く，また何種類もの並列計算機が作られてきた．配列変数の各要素におなじ演算命令を適用するデータ並列型（Single Instruction Multiple Data, SIMD），複数の演算器が1つのメモリを同時に読み書きする共有メモリ型（Multiple Instruction Multiple Data, MIMD），独立した演算器とメモリを備える計算ノードを通信路で連結した分散メモリ型，あるいはそれらの組み合わせなどである．データ並列を基調にしたベクトル型計算機が全盛であったころは，FORTRANできちんとループを書けば性能が出た，良い時代だったという話は聞く．しかし，市販のパソコン向けのCPUの性能が目覚ましい向上を遂げるにつれ，ベクトル計算機は次第に価格性能比で劣勢になっていった．

ほぼ全てのパソコンと，9割がたのスパコンのCPUがIntel x86互換になった2000年代後半になって登場した新機軸がGPGPUである．このころ，2年ごとに倍という指数関数的ペースで増加してきたCPU1コアあたりの周波数の向上が止まったこと（フリーランチの終焉）がはっきりし，単一CPU内でもSIMD拡張やマルチコアといった並列化技術が多用されるようになった．GPUメーカーは，将来にわたって指数関数的性能向上を継続させるとしているが，それはもはや周波数の増加によるものではない．

むしろ現行のGPUの周波数は，CPUの1/3程度である．GPUの演算力の源は1チップに500個から1500個程度もの演算ユニットを搭載し，数万のスレッドをさばくことのできる莫大な並列性である．パソコンの部品として量産されているため安価であったこともGPUの爆発的流行の原因として無視できない．実際，本稿執筆時点では単精度1TFlopsあたり1万円と，市販のCPUと比べても価格あたりのピーク性能比は10倍良い．ハードウェアの費用は開発投資の占める割合が多く，いくつ売れたかで価格が決まる．汎用的な演算器を多数搭載して並列性能を，というアイデアは特別なものではなく，そのようなハードウェアを目指した学術的研究はいくつもあった．だが画像処理専用チップでしかなかったGPUをここまで進化させたのは，より高品質，より斬新なグラフィックをもとめるゲーマーたちの欲望であった．

天河一号（Tianhe-I）という，GPUを主力演算器として採用した中国のスパコンが首位を獲得した2010年11月のTOP500ランキングは，Roadrunnerという，プレイ

* 京都大学次世代研究者育成センター
The Hakubi Center, Kyoto University

ステーション3のCPUであるCellを採用したスパコンが初めて1PFlopsを突破した2009年6月に引き続き，アクセラレータを搭載したスパコンの台頭を印象付けた．国内でも東京工業大学のGPUスパコンTSUBAME2.0が共同利用に開かれている．GPUが組み込まれた数値計算向けのパソコンも多数発売されている．ハードウェアの準備は整った．

使用する立場からすればプログラミングの容易さも重要だ．NVIDIA社はCUDAというプログラミング言語(**図1**)を開発し，GPGPUの歴史の初期から無料で頒布していた．CUDAは基本的に，配列変数を添え字で操作するという，昔ながらの記法だ．内部では何要素かづつSIMD(Single Instruction Multiple Data)実行されている・・・などの詳細はひとまず気にしなくてもプログラムが書ける．並列計算の記述法も，各スレッドが自分のインデックスを取得し，それをもとに配列の異なる要素にアクセスするという，MPIの使い手にはお馴染みのものだ．

またCUDAでは，広範囲のプログラマにとっての共通言語であるC++の機能を備えていたため，メモリの確保と解放，転送，データの管理など，プログラミングの根本部分を便利にするライブラリを作ることができる．私がGPGPUを学んでいく過程でも，公開されていくライブラリに，また自分でライブラリを実装できることにずいぶんと助けられたものだ．NVIDIAのライバル企業ATIも同時期にGPGPUを展開していた(そして演算性能ではATIの方が上だ)が，プログラミング言語から掲示板まで含めた開発環境の整備で遅れをとり，GPGPUといえばCUDAが代名詞となるまでの差がついた．それほどプログラミング環境は重要なのだ．最近では，OpenCLという，各種GPUやCell，CPUを統合的にプログラムできる言語も普及が始まっている．

どうしてもFortranでプログラムしたいという人にも，PGI社などから，Fortranプログラムにプラグマを挿入することでGPU向けのプログラムを生成できるコンパイラが発売されている．試してみられるといいだろう．

かつて全盛をきわめたベクトル計算機が衰退したように，GPUもいつか，何かに取って代わられ，滅ぶ運命にある．ただし演算回路の周波数はもう大幅には上がらないとされている以上，新しい計算機も，1チップに膨大な並列性を含むアクセラレータを備えている

```
module my_data
  integer,parameter :: N = 500
end module my_data

subroutine calculate (xs, ys)
  use my_data
  real, dimension(N) :: xs, ys
  do i=1, N
     ys(i) = sqrt(xs(i))
  end do
end subroutine calculate

program main
  use my_data
  real, dimension(N) :: xs, ys
  do i=1, N
     xs(i) = i-1
  end do
  call calculate(xs, ys)
  do i=1, N
     print *, "sqrt ", xs(i), " is ", ys(i)
  end do
end program main
```

```
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

const int N = 500;

void calculate (vector<float> &px, vector<float> &py) {
  for (int i = 0; i < N; i++) {
    py[i] = sqrt(px[i]);
  }
}

int main () {
  vector<float> xs(N), ys(N);
  for (int i = 0; i < N; i++) {
    xs[i] = i;
  }
  calculate(xs, ys);
  for (int i = 0; i < N; i++) {
    cout<<"sqrt "<<xs[i]<<" is "<<ys[i]<<endl;
  }
}
```

```
#include <iostream>
#include <thrust/device_vector.h>
using namespace std;

const int N = 500;

__global__
void calculate (float *px, float *py) {
  int i = threadIdx.x;
  py[i] = sqrt(px[i]);
}

int main () {
  thrust::device_vector<float> xs(N), ys(N);
  for (int i=0; i<N; i++) {
    xs[i] = i;
  }
  calculate<<<1, N>>>
    (thrust::raw_pointer_cast(&*xs.begin()),
     thrust::raw_pointer_cast(&*ys.begin()));
  for (int i = 0; i < N; i++) {
    cout<<"sqrt "<<xs[i]<<" is "<<ys[i]<<endl;
  }
}
```

図1　500個の整数の平方根を計算するサンプルプログラム．それぞれFortran (左上), C++ (右上), CUDA (左下)

はずだ．今は「アクセラレータ」に対する主人の地位にあるCPUの方が無くなっているかもしれない．そのようなハードウェアの使い方を学ぶ機会としてGPGPUに取り組めば，世代交代のときにも何がしかを持ちこせるだろう．

神戸に建設中の次世代スパコン「京」とて，アクセラレータこそ採用していないもののSIMD命令から10万のノード，ファイルシステムまで，多段にして膨大の並列性を備えている．将来のスパコンを動かすのはより驚くべき体験になるに違いない．それを味わうために，さまざまな並列計算を使いこなせるようにしておきたい．

2. 天文・宇宙物理分野におけるGPGPUの利用

今や，GPUが利用されているシミュレーションの分野は非常に多岐にわたっている．その広がりを一望したければ，NVIDIA社のサイト“CUDA ZONE”を訪れるとよい．ここでは私の専門である天文・宇宙物理分野にかぎって，GPU計算の利用例を紹介させていただく．

天文観測家たちは初期からGPGPUを利用し始めたグループの1つだ．電波干渉計では，複数のアンテナで受信した信号を位置空間の画像に変換するために離散フーリエ変換を行う必要がある．Harris[1]らはこの計算をGPU化した．いくつかのアルゴリズムを比較し，1.8Ghzのデュアルコア Opteron265プロセッサを使った場合に比べ，NVIDIA社のGeForce 8800 GTSを使った場合，10倍から100倍の高速化を達成したとしている．Wayth[2]らは同様の計算をTESLA 1060など，より新型のGPUでも行い速度を比較している．

またThompson[3]らは，宇宙に分布する天体の重力により光が曲げられる重力レンズ効果を計算するためのレイトレーシングをGPU上で実装し，Intel Q6600 CPUの4コアをOpenMPで利用した場合にくらべ，NVIDIA 8800GTを2枚使用した場合で60倍，NVIDIA S1070を1枚使用した場合で190倍の高速化を達成したと報告した．

JonssonとPrimackの仕事[4]は，銀河がどの波長の光をどのくらい出すかというスペクトルモデルの計算にGPUを使用したというものだ．モデル自体はさまざまな効果を含む複雑なものなのだが，その中でも宇宙塵の熱収支という一番計算コストがかかる部分だけをGPU化すれば済んだというケースである．

多数の質点のニュートン重力による相互作用の計算（N体計算）は，より多くの粒子を扱い新しい物理を引き出すべく，高性能計算の競争が激しい分野だ．ここでもいちはやくGPUの利用が見られた．Belleman[5]らはIntel Xeon，NVIDIA 8800 GTX，それからN体計算専用計算機GRAPE-6Afを使った場合のN体計算の速度を比較し，GPUではCPUのおよそ100倍，専用計算機GRAPE-6Afに匹敵する速度が得られたとしている．

これまで紹介してきた研究はおもにNVIDIA社のGPUとプログラミング言語CUDAを使用したものだったが，市販されている並列計算ハードウェアの選択肢はそれにとどまらない．Pang[6]らは，磁気流体方程式のシミュレータをIntel社のx86のほか，プレイステーション3のCPUであるCell，NVIDIA社およびATI社のGPUに対して，FORTRAN, C, Cell, CUDA, OpenCLを使用して実装し，その性能を比較した．XEON E5506を1コア使用した場合に比較して，8コア使用した場合は6.7倍，Cellを使用した場合10.2倍，NVIDIA社のGTX 260を使用した場合105.7倍，ATI社のHD5870を使用した場合68.5倍高速化したとしている．

かように様々な計算がGPGPUを利用して100倍だの1000倍だの速くなったとGPU陣営が宣伝するので，CPU陣営から物言いがついた[7]．どんなものでも100倍速くなるというのは最適化されていないCPUコードとGPUコードを比較した誇大広告で，CPU側のコードもきちんと最適化すれば，性能差は10倍もないだろう，というのである．ピーク性能に近い速度が出るまで最適化するとなればCPUの方が比較的容易で，CPUとGPUの演算ピーク性能は10倍程度しか違わないからだ．私としては健全な競争によりハードウェアが速くなってくれれば嬉しい．ただ，なんでも100倍という誇大広告にも一片の真実はあると思う．私のような素人が書いたコードなどピーク性能の1%も出ないことはよくある．ピーク性能に近い値というのはそこから数倍，時に1.1倍の改善を積み重ねて初めて到達しうるものだ．一方，そんなプログラムなら，GPU計算に適したアルゴリズムを見抜き，CUDA配列変数の操作を使って書き直すだけで，百倍千倍速くなることはありうる．一手でこれだけ稼げる手軽な最適化は他にあまりない．むろん，多くの人が，GPUを価格あたり，電力あたりの性能を極限まで追求する手段として血の滲むような努力をしておられる．だが私は皆さんに，GPUを最適化のための最終兵器と敬遠するよりも，楽して速くするための最初の一歩として試してみられては，とお勧めしたい．

一台のパソコンにおさまる数枚のGPUにとどまら

ず，何百というGPUを並列に利用した宇宙物理学的シミュレーションも行われている．たとえば長崎大学のHamadaらのグループは同大学先端計算研究センターのGPUスパコンDEGIMAを使用して様々なGPUアプリケーションを開発しており，N体計算において2009年に42TFlops[8]，2010年に190TFlops[9]を達成した．私が本稿で紹介する星間物質のシミュレーションも，そのDEGIMAや東京工業大学TSUBAMEにおいてGPU並列計算を実現したものだ．

宇宙物理学への応用を考えた場合，より複雑な基礎方程式やその組み合わせのシミュレーションが必要だ．それらについても研究が進んでいる．先にあげた磁気流体方程式もその1つだ．Schive[10]らのグループはGPGPUの歴史としてはかなり初期に，GPUクラスタを用いて流体力学と重力を組み合わせた方程式を，状況におうじて解像度を細かくする適合格子上で解くという複雑なことをやっていた．Riemannソルバに基づく一種の粒子法で磁気流体方程式のソルバを実装した，などという変わり種もある[11]．そして最近，GPUで実装されたADM形式にもとづく一般相対論的磁気流体方程式のソルバも現れた[12]．シミュレーション宇宙物理学のさまざまな方程式で，GPU計算の準備が整ってきている．

3. GPGPUを利用した星間乱流の高解像度シミュレーション

ここで，私自身がGPGPUを使って行った研究，水素原子ガスの相転移エネルギーにもとづく熱的不安定に駆動される星間乱流の三次元流体シミュレーションを紹介したい．この問題は星間物質の観測の理解との直接対応はむろん，熱的不安定によって無数に生じた低温高密度の塊がやがて恒星，惑星系形成の初期条件に繋がるという意味でも重要だ．

シミュレートすべき式は，以下のように流体力学の基礎方程式，Navier-Stokes方程式に加熱・冷却項を加えたものである．

$$\frac{\partial \rho}{\partial t}+\nabla\cdot(\rho\mathbf{u})=0 \tag{1}$$

$$\frac{\partial \rho\mathbf{u}}{\partial t}+\nabla\cdot(\rho\mathbf{u}\bar{\mathbf{u}})+\nabla P=0 \tag{2}$$

$$\frac{\partial E}{\partial t}+\nabla\cdot((E+P)\mathbf{u})=\Gamma(\rho,T)-\Lambda(\rho,T) \tag{3}$$

ここで，加熱項Γと冷却項Λは，星間環境で考慮すべき様々な加熱・冷却過程[13]をもとに，Inoue and Inutsuka[14]が提案した次のフィッティング式をもちいる．

$$\Gamma(\rho,T)=6.802\times10^{-2}\rho \tag{4}$$

$$\Lambda(\rho,T)=\rho^2\left[3.871\times10^{5}\exp\left(\frac{-19.25}{T+0.1626}\right)+4.25\times10^{-2}\sqrt{T}\exp\left(\frac{-1.496\times10^{-2}}{T}\right)\right] \tag{5}$$

定数がいくつか出てきたが，星間物質のスケールで典型的な物理量を使って無次元化をおこなっている．つまり，

$$\text{密度：}\ \rho_w=1.211\times10^{-21}\ \ [\mathrm{kg\,m^{-3}}] \tag{6}$$

$$\text{圧力：}\ p_w=4.832\times10^{-8}\ \ [\mathrm{Pa}] \tag{7}$$

$$\text{長さ：}\ l_w=1\ \mathrm{parsec}=3.086\times10^{16}\ [\mathrm{m}] \tag{8}$$

この研究では一辺20パーセク（約6×10^{17}[m]）の計算空間を設定し，衝撃波が形成されるよう対向する2つのガス流がある初期条件・境界条件をおいた．

開発した流体コードの特性を列挙すれば，MPI GPU Full-Godunov 2nd order MUSCL 3-dimensional uniform mesh Navier-Stokes equations solverとなろうか．このコードはMPI経由で多数のGPUを並列に利用して計算を行うことができる．これに加え，故障率の高いマシンや，キューの実行時間に制限があるマシンでも長時間の計算が行えるよう，シミュレーションの内部状態を適宜出力してそこから復帰できるチェックポインティング機能をつけた．またムービー作成のための不可逆圧縮データ出力機能を，東京工業大学の青木尊之教授のアドバイスも得て実装した．

コード開発やチューニングはおもに長崎大学DEGIMAで，InfiniBandで連結された最大576個のNVIDIA GT200 GPUを用いて行った．このシステムの単精度ピーク演算性能は514.9TFlopsであり，約1280^3の解像度のシミュレーションにおいて40.91TFlopsの実効性能を達成した．

つぎに東工大のGPUスパコンTSUBAMEを，学際大規模情報基盤共同利用という枠組みから利用し，長時間計算を進めた．120個のNVIDIA GT200 GPU型演算器が利用可能，単精度ピーク演算性能は124.2TFlopsのhpc1tes2キューをのべ10日利用し，1440^3の解像度で，約8 sound crossing timeの計算ができ，乱流スペクトルを解析する上で十分な量のデータが得られた．実効演算性能は11.5TFlopsであった．

本研究のシミュレーションではこれまでにない大解像度を達成したことから，その出力データ量も膨大なものとなり，その解析および可視化にも工夫が必要であった．例えば1つのスナップショットのサイズは約

60GBであり，典型的なパソコンのメモリに収まらない．そこでデータの2次元断面程度のメモリしか消費せず，かつディスクアクセスを最小限に抑えて，スペクトル解析や塊検出，可視化などを行うアルゴリズムを開発した．

解析の結果，シミュレートされた乱流のべきは，超音速乱流領域，Kolmogorov様領域，数値的散逸領域の3重構造を示した(**図2**)．全体としての冪はChepurnov[15]の観測による速度・密度スペクトル($\alpha_v=3.87\pm0.11$)($\alpha_\rho=3.0\pm0.1$)と誤差の範囲内で一致した．超音速領域は比較的softer, Kolmogorov 様領域は比較的harderである．これらのことから，Chepurnov[15]のような非Kolmogorov的観測は超音速領域とKolmogorov領域の重ね合わせとして説明できるかもしれない，という示唆を得た．

このデータの可視化にあたっても，ハードディスクに置いたまま，1回あるいは数回のシーケンシャルアクセスのみで，流体速度，密度，化学状態，温度など，さまざまな物理量を可視化できるプログラムを開発した．これにより，熱不安定性を引き起こしている衝撃波面が激しくゆらぎ，強い噴出流が間欠的に突きあげる乱流のありさまが明らかとなった．また，大規模データの可視化を得意とする京都大学小山田研と共同研究により，40面タイルドディスプレイを利用した大画面・高解像度の可視化を行った(**図3**)．最近はこの流体コードの開発経験を元に，同じくマルチGPU向けの磁気流体コードを開発している．

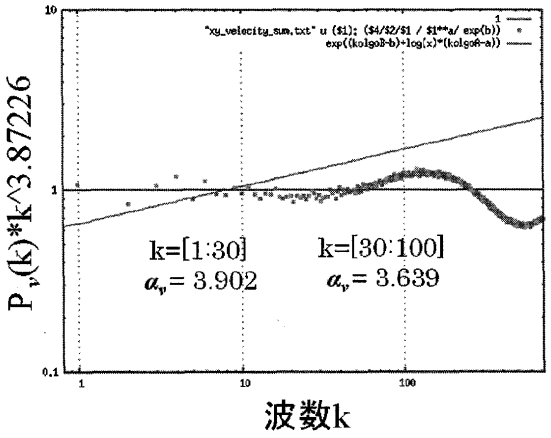

図2　Velocity Spectrum Fit by Single Power Law.

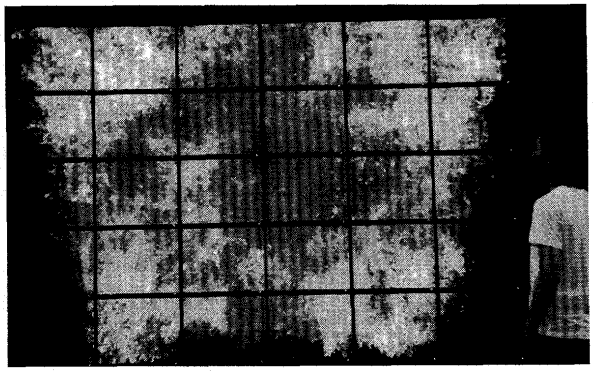

図3　Visualization on 40-face Display.

4. 並列計算の将来展望

MPI+CUDAの流体コード(他の方々のアプリケーションと比べても実に単純だと思う)を書いていても，コードの長さにうんざりし，C++/CUDAによる抽象化をしようとしても，C++の抽象化能力の限界に苛立たせられることは多かった．そこで，偏微分方程式の陽解法，というクラスに限ってだが，アルゴリズムの抽象的で簡潔な記述から，並列計算機上の実装を自動生成するプログラミング言語を作りたいという，以前からの夢を実行することにした．これまでのGPU並列計算の実装経験とプログラム自動生成に関するサーベイを元に，2011年4月よりコード生成器Paraisoの開発を開始している(http://www.paraiso-lang.org/wiki/)．すでに簡単な流体計算プログラムを生成できるようになっており(**図4**)，前途の課題は多いものの，並列度を完全に含んだ抽象度の高い記述からコードが生成できる

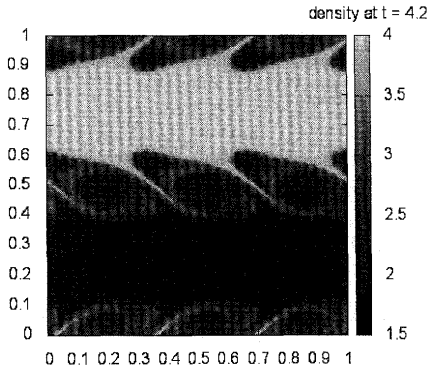

図4　Hydrodynamic Sovler, Generated from Paraiso.

```
-- | Make a binary operator
mkOp2 :: (Vector v, Ring.C g, TRealm r, Typeable c) =>
         A.Operator                  -- ^The operator
      -> (Builder v g (Value r c)) -- ^Input 1
      -> (Builder v g (Value r c)) -- ^Input 2
      -> (Builder v g (Value r c)) -- ^Output
mkOp2 op builder1 builder2 = do
  v1 <- builder1
  v2 <- builder2
  let
      r1 = Val.realm v1
      c1 = Val.content v1
  n1 <- valueToNode v1
  n2 <- valueToNode v2
  n0 <- addNode [n1, n2] (NInst (Arith op) ())
  n01 <- addNode [n0] (NValue (toDyn v1) ())
  return $ FromNode r1 c1 n01
```

図5　A Code Portion of Paraiso that Defines Binary Arithmetic Operator between Programs.

ことを実証したことは意義のある一歩だと考えている．

ParaisoはHaskellという，1995年生まれのプログラミング言語を使って書かれている．Haskellの魅力についてお伝えするだけの紙面も知識も私にはない．興味のある方は最近出版された解説書[17]をご覧になってほしい．1つだけParaisoの中核となっている技法，「プログラムの代数」を紹介する（**図5**）．要素的な計算を行うプログラムの間に，加減乗除や微分，等号といった代数演算を定義する．すると，それら要素プログラムを「方程式に代入」することが意味を持つ．解きたい方程式の各項に要素プログラムを代入した結果は，その方程式を解くプログラムとなるのである．わけのわからない説明で申し訳ない．が，そんなプログラミングの仕方が可能なのだ．この操作は，古典力学の量子化を行うとき，適切な関係を定義した量子オペレータたちを古典力学の方程式に代入することで量子力学の方程式を得る，という操作に似ている．イメージしてください．

かつてこの構想を，請われるままJames Stoneに語ったところ彼は言ったものだ．「すばらしい．だが私は年を取りすぎて，似たような計画が失敗したことを知っている」と．彼の念頭にあったのはHPF（High Performance Fortran）のことだ．HPFについては面白い論文がある．2007年に，計画のリーダーであったKennedy自身が，HPFはなぜ失敗したのか，という論文を，歴史的教訓のために残しているのだ[17]．汎用的で，書きやすくて，高性能で，多くのマシンで動くという願いそのものが矛盾を孕んでいた．またプログラミング言語は企業に多額の報酬を支払って作るものであり，その予算が枯渇した，という．同じような希望を掲げて生まれ，消えていった並列言語は数知れない．私に勝ち目はあるのか... 言語の開発はずっと容易になった．HPFが挑んだほどの汎用性は求めていない．実用レベルの効率が達成できるかは，私もわからない．たとえ失敗するとしても，ずっとC++の亜流を手で書き続ける人生よりはましだとは思う．そして何度でも言い返すだろう．抽象化と高速化は両立する，いや抽象化こそが高速化への道なのだ，と．

現に，フーリエ変換や行列計算の上質なライブラリには，すでに自動生成に基づいて作られているものがある[18,19]．人間がアルゴリズムを実装する代わりに，コンピューターに線形計算を構築するための要素と法則を教え，それらを組み合わせ，指数関数的な場合の数の実装の中からベンチマークと機械学習により高速なものを探す．このような芸をコンピューターに仕込むには，行いたい計算や施すべき変形を伝えられるだけの抽象化能力をもった言語が不可欠だ．特定の機種に絞って職人がとことんチューニングしたコードの性能を10割とすれば，自動最適化は過去と未来の全ての機種で9割方の性能を発揮できる．人の手を借りずに，だ．

やがてシミュレーション科学者たちが，いま必要に応じてプログラムを書いているのと同程度の手軽さで，必要に応じてプログラミング言語を設計するような時代が来るかもしれない．そうやってプログラム生成過程を制御下に置くことで，解きたい問題を乗り換えたり，ハードウェアを変更したりしても，プログラムをすべて1から書き直さなくても済むようになる．並列化やチューニングといったプログラムに施す変換も，一度編み出したテクニックを再利用可能な形で保存し，体系化できる．プログラムを変換するときに忍び込むミスや，多次元座標のある成分だけうっかり扱いを取り違えるといったくだらないバグは，すべて生まれる前に消し去ることができる．

Kennedyの論文は最後に，次のような希望的観測で終わる．A decade of research and development has overcome many of the implementation impedements. Perhaps the time is right for the HPC community to once again embrace new data-parallel programming models similar to the one supported by HPF. その通り．プログラミング言語の設計を支援するための普遍的な概念が多数研究され，実装され，それら概念の助けが借りられる時代になった．だからこそ，私のような駆け出しでも，言語の設計という，物理学者には向かない作業を始めることができた．

自らが希望となることを願って．

参 考 文 献

1) C. Harris, K.Haines and L. S. Smith: GPU accelerated radio astronomy signal convolution, Exp Astron **22**, 129/141 (2008)
2) R.B. Wayth, L.J. Greenhill, F.H. Briggs: A GPU-based Real-time Software Correlation System for the Murchison Widefield Array Prototype, Publications of the Astronomical Society of the Pacific **121**-882, 857/865 (2009)
3) A.C. Thompson, C.J. Fluke, D.G. Barnes and B.R. Barsdell: Teraflop per second gravitational lensing ray-shooting using graphics processing units, New Astronomy **15**, 16/23 (2009)
4) P. Jonsson, J. Primack: Accelerating dust temperature calculations with graphics-processing units, New Astronomy **15**-6, 509/514 (2010)
5) R.G. Belleman, B. Joroen, S.F.P.Zwart, High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA, New Astronomy **13**-2, 103/112 (2008)

6) B. Pang, U. Pen, M. Perrone: Magnetohydrodynamics on Heterogeneous architectures: a performance comparison, arXiv: 1004.1680 (2010)
7) V.W. Lee. et. al: Debunking the 100X GPU vs. CPU myth an evaluation of throughput computing on CPU and GPU, Computer architecture news, **38**-3, 451/460 (2010)
8) T. Hamada et. al.: 42 TFlops Hierarchical N-body Simulations on GPUs with Applications in both Astrophysics and Turbulence, High Performance Computing Networking, Storage and Analysis - SC '09
9) T. Hamada and K. Nitadori: 190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs, High Performance Computing Networking, Storage and Analysis - SC '10
10) H. Schive, Y. Tsai, T. Chiueh: GAMER: A Graphic Processing Unit Accelerated Adaptive-Mesh-Refinement Code for Astrophysics, Astrophysical Journal Supplement, **186**-2, 457/484 (2010)
11) E. Gaburav, K. Nitadori: Astrophysical Weighted Particle Magnetohydrodynamics, arXiv: 1006, 4159
12) B. Zink: HORIZON: Accelerated General Relativistic Magnetohydrodynamics, arXiv: 1102. 5202
13) H. Koyama and S. Inutsuka (2000), Astrophysical Journal, 532, 980
14) T. Inoue and S. Inutsuka (2008), Astrophysical Journal, 687: 303-310
15) A. Chepurnov A and A. Lazarian (2010), Astrophysical Journal, 710, 853
16) M. Lipovaca: Learn you a Haskell for Great Good!, 400, No Starch Press (2011)
17) K. Kennedy, C. Koelbel, H. Zima: The rise and fall of High Performance Fortran: an historical object lesson, ACM SIGPLAN, 7-1/7-21 (2007)
18) M. Frigo and S. G. Johnson: The Design and Implementation of FFTW3, Proceedings of the IEEE **93** (2), 216–231 (2005)
19) M. Puschel et al.: SPIRAL: Code Generation for DSP Transforms, IEEE **93** (2), 232-275 (2005)