地球流体力学に関する GPGPUを用いた数値計算

神戸大学　惑星科学研究センター
西澤誠也

- 地球流体力学とは
  - 地球・惑星に関連がある流体の力学
  - 回転, 重力の影響

  e.g. 大気, 海洋, マントル

- 数値計算は 天気予報 & 弾道軌道予測 から始まった
- ベクトル計算機
  - 地球流体の計算はベクトル長が長いものが多い
- ベクトル計算機の凋落
  - 某社の次世代スパコンからの撤退

- 個人的スパコンの将来予想
  - 個々の演算器はシンプルに
    - 単一演算器の消費電力を減らす
      - トランジスタ数を減らす
        - » 消費電力はトランジスタ数におおよそ比例
        - » 性能はトランジスタ数の平方根におおよそ比例
      - クロック周波数を下げる
        - » 消費電力(クロック周波数におおよそ比例)は電圧の自乗の比例
        - » 性能はクロック周波数に比例
    - 製作コストを下げる (歩留まりをあげる)
      - ダイサイズを大きくしない
  - 演算器の数を増やす (超並列)
    - 比較的単純な汎用プロセッサをたくさん
    - 加速器
      - e.g. GPU, GRAPE

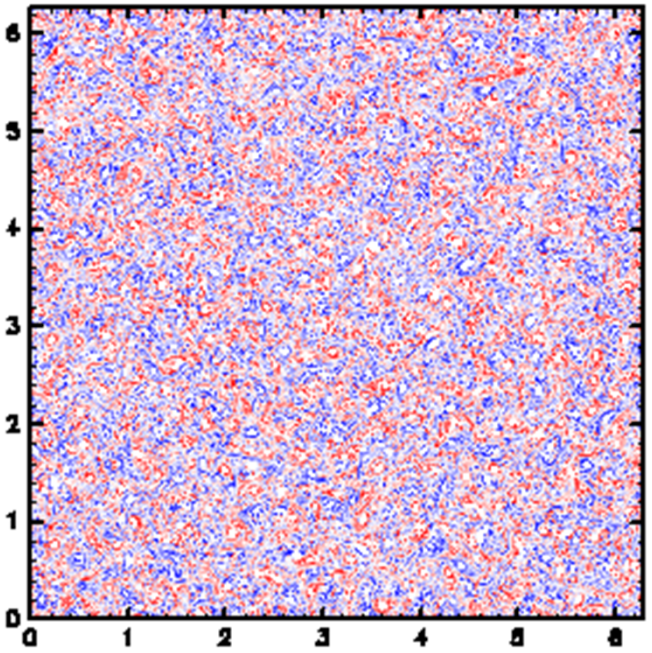
超並列

- コーディングがより困難に
  - メモリアクセス
  - 通信
  - 競合
- 演算は安く，メモリ,ネット,IOバンド幅が高価に
  - 無駄な計算をおしまない
- ノウハウの蓄積が必要
  - まずは簡単な問題から
  - 知識の共有
    - 個々人で別々にやっていては効率が悪い

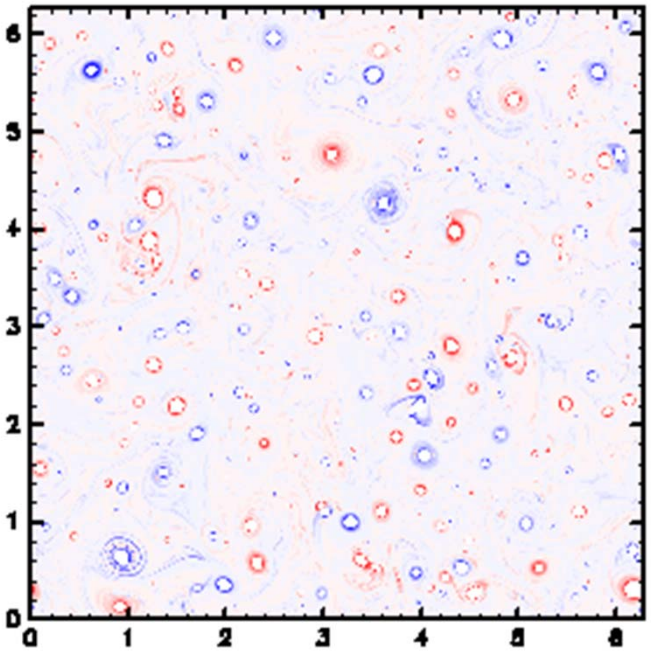
計算事例 1

- ランダムな乱れた流れから秩序構造(大規模渦)をもつ流れへ発展
  - エネルギーの逆カスケード

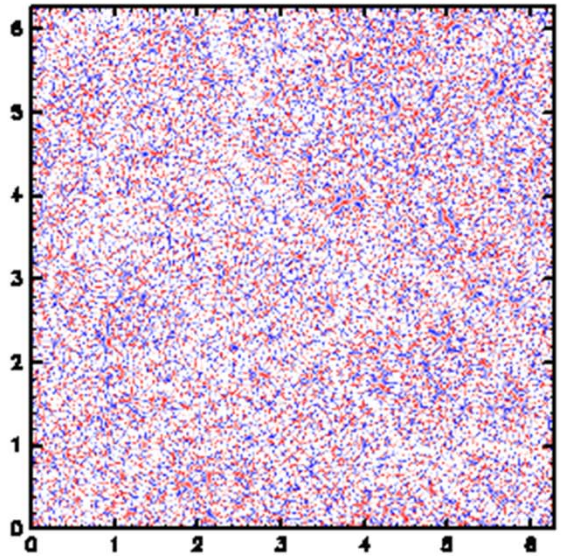
初期時刻

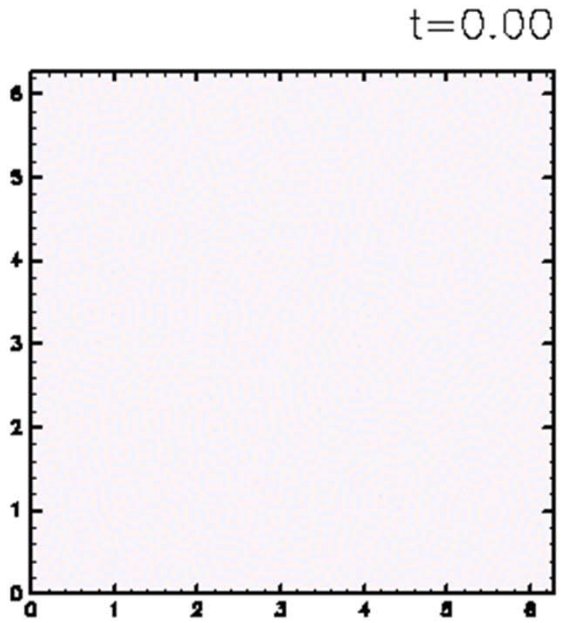


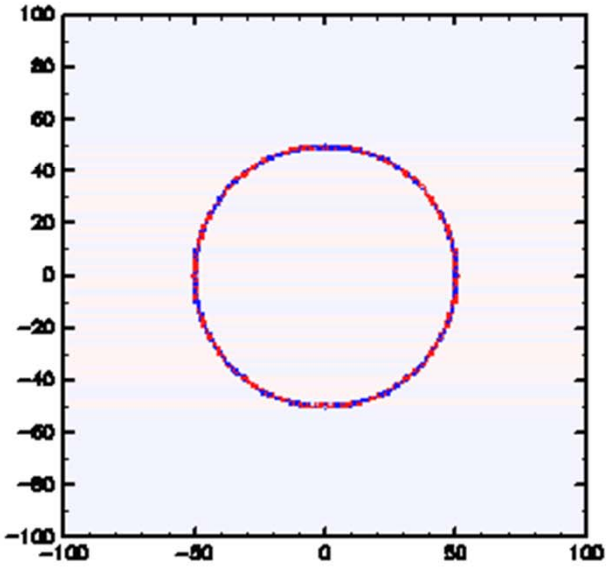
時間発展後

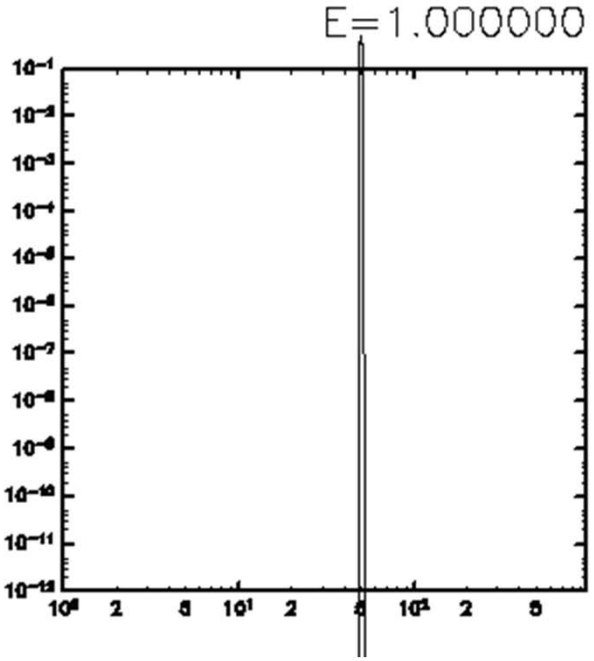


渦度場の時間発展の様子




t=0.00




E=1.000000

- 2次元
  - 現実世界は3次元であるが, 重力, 回転の影響で, 現象は2次元的な振る舞いをする
    - 成層
    - テーラーカラム
- 乱流と大規模渦
  - 現実の現象では, 小さなスケールと大きなスケールの現象が相互に影響し合っている
- 渦
  - 現実の現象には渦が満ちあふれている
    - 低気圧
    - 亜熱帯還流

- 解くべき変数は渦度のみ

$$\frac{\partial \zeta}{\partial t} + u\frac{\partial \zeta}{\partial x} + v\frac{\partial \zeta}{\partial y} + \beta v = \nu \nabla^{2p} \zeta$$

ζ : vorticity
u, v : verocity(x, y direction)
β : latitudinal gradient of planetary vorticity
ν : diffusion coefficient

- もともと Fortran でコードを書いていた
  - 差分化: スペクトル法 (三角関数)
  - 切断波数: 682 (空間解像度: 2048 x 2048)
  - 境界条件: 周期境界条件
  - 時間積分: 4次のルンゲクッタ

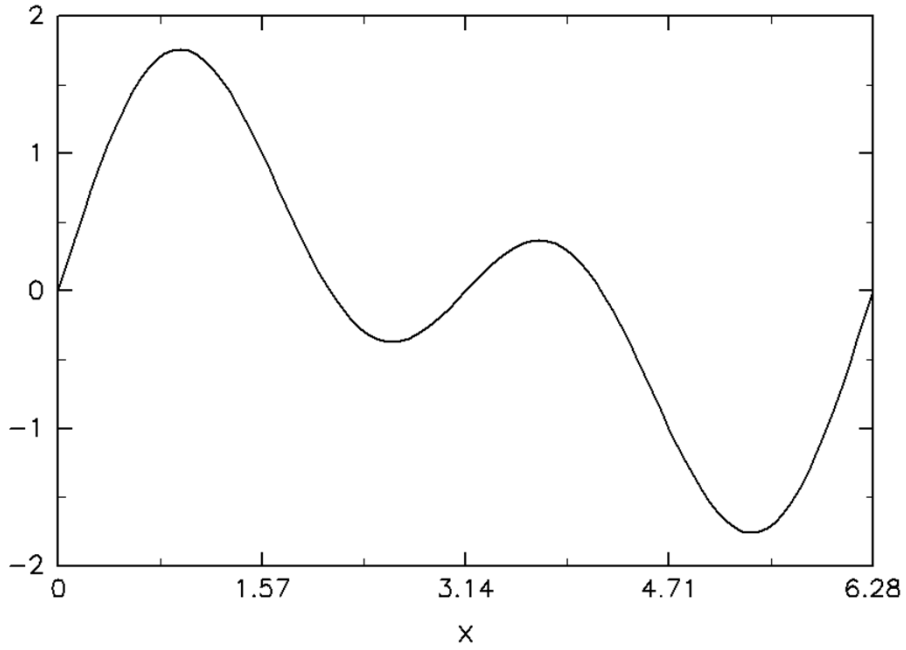
  時間ステップ数 10,000 で、計算時間はおおよそ1ヶ月 (Intel Core 2 Quad 9550)

スペクトル法とは

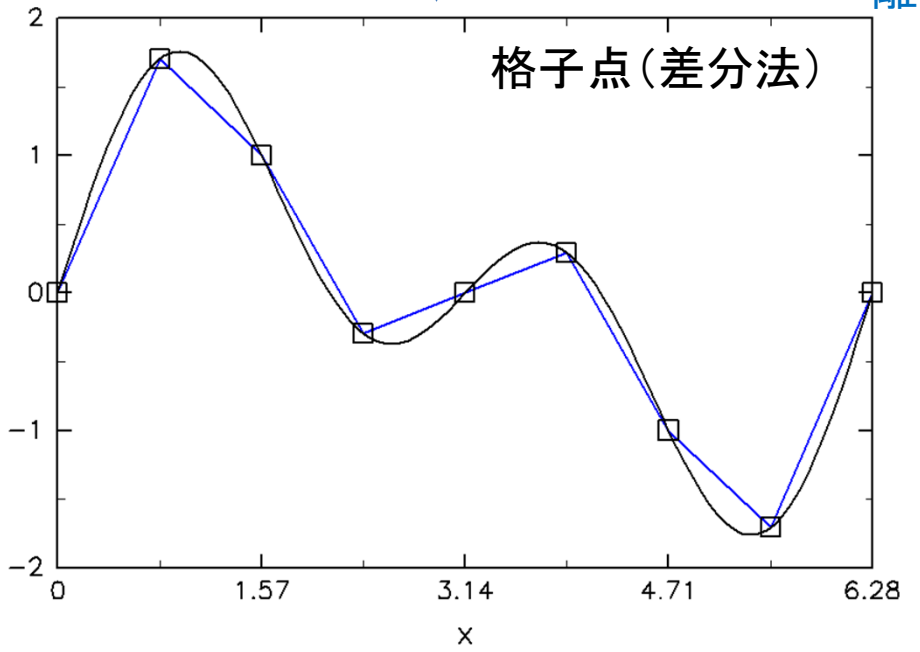
- 偏微分方程式の数値解法の一つ
- 関数を、いくつかの独立な基底関数に展開し、展開係数に対する常微分方程式を計算する
  - e.g. 三角関数(sin,cos)

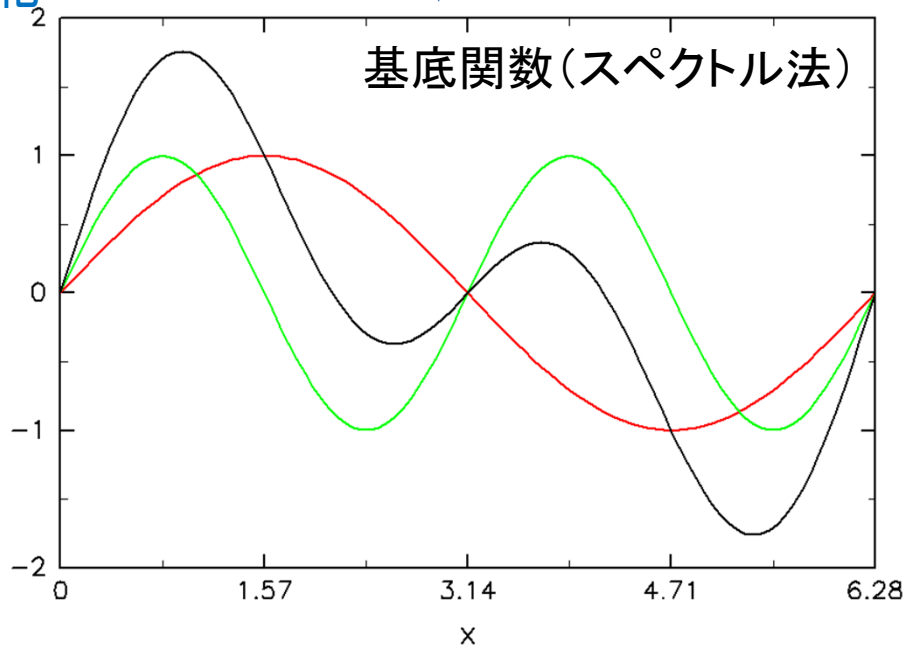
$$f(x) = \sum_{k=0}^{K} a(k)\cos(kx) + b(k)\sin(kx)$$

ある物理量は
x の関数



離散化





- 長所
  - 展開関数系として、なめらかな関数を選ぶことにより、偏微分を差分近似することによる誤差がない (高精度)
  - 展開関数系をうまく選ぶことにより、得られる常微分方程式が非常に簡単になる
- 短所
  - 境界条件や領域の形が複雑な場合は、展開係数が簡単には構成できない
  - 非線形の場合には特別な工夫が必要となる

- 非線形項の計算
  - 非線形項のみ、実空間で計算する
    - たたみ込み積分は計算量が多い
      - 計算量: $O(N^2)$
    - 前後に実-スペクトル変換が必要
    - 高速な変換法が利用できる
      - 高速フーリエ変換 (FFT) の計算量: O(NlogN)
  - エイリアスエラー(エイリアシング)を除去する
    - 実空間における格子点数を切断波数の3倍以上とる必要がある

- GPGPU により高速化
  - CUDA + cuFFT
  - 一週間程度の開発で約20倍 (Tesla C1060 vs Core2Quad Q9550)
  - ほとんどチューニングなしで、まずまずの高速化
    - GPGPU普及には非常に重要な点
    - 深く考えずにとにかくやってみましょう

cuFFT ライブラリ

- FFTをCUDAで動作するように実装したライブラリ
  - CUDA Toolkit に含まれる
    - 特別なものをインストールする必要が無い

- sample code (実数のフーリエ変換)

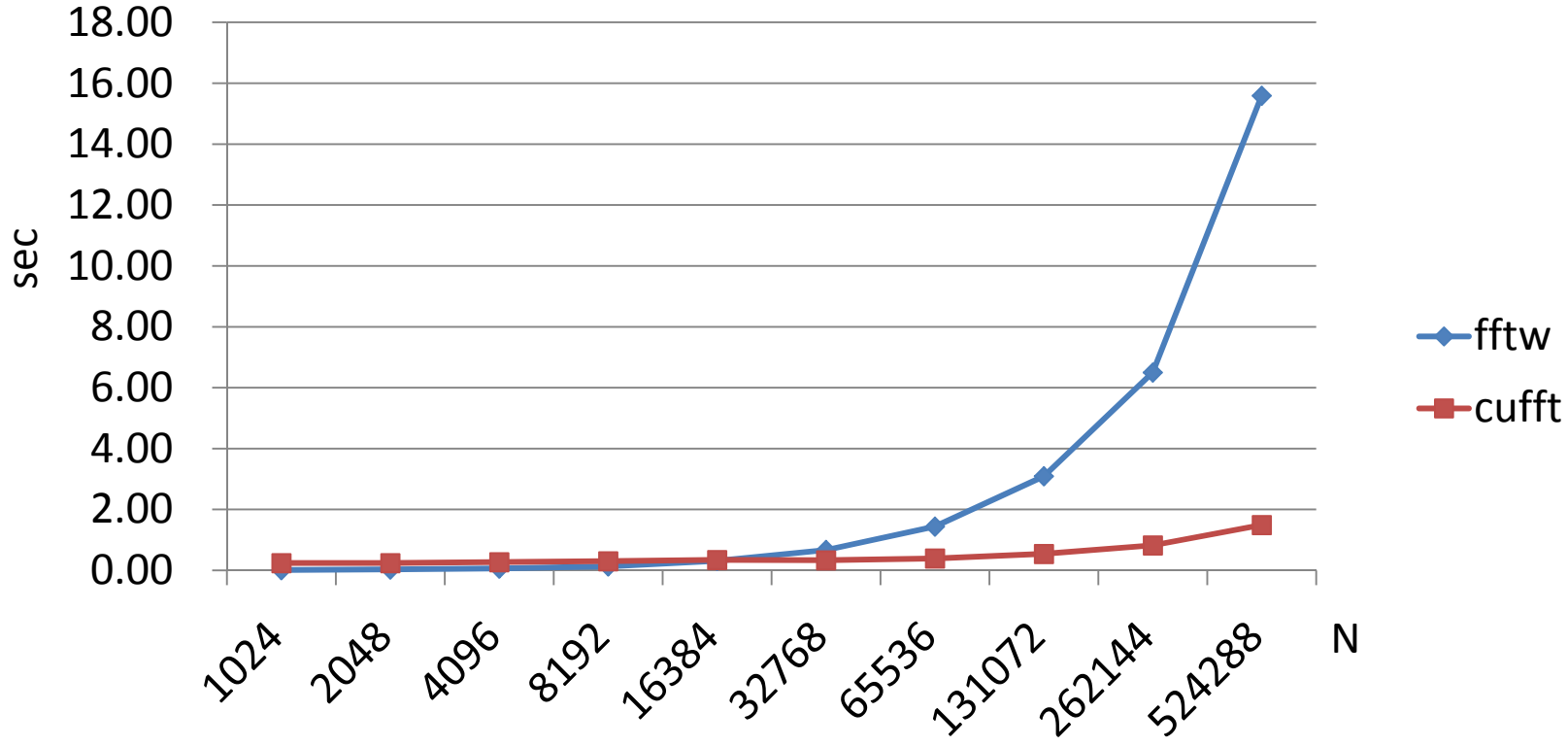
```
#include <cufft.h>
#include <math.h>
#define NX 256

int main(int argc, char **argv)
{
  float data_h[NX+2], *data;
  cufftHandle plan_r2c, plan_c2r;  // 順変換、逆変換用のプランが必要 (r2cの場合)

  cufftPlan1d(&plan_r2c, NX, CUFFT_R2C, 1);  // 順変換用プラン作成
  cufftPlan1d(&plan_c2r, NX, CUFFT_C2R, 1);  // 逆変換用プラン作成

  for (int i=0; i<NX; i++) data_h[i] = sinf( 2*M_PI*i/NX );  // 波数 1 の sin データ作成
  cudaMalloc((void**) &data, sizeof(float)*(NX+2));  // スペクトルデータは, NX+ 2 必要 (r2c の場合)
  cudaMemcpy(data, data_h, sizeof(float)*NX, cudaMemcpyHostToDevice);

  cufftExecR2C(plan_r2c, data, (cufftComplex*) data);  // 順変換実行 (結果は NX 倍されている)
  cufftExecC2R(plan_c2r, (cufftComplex*) data, data);  // 逆変換実行

  cufftDestroy(plan_r2c);  // プラン破棄
  cufftDestroy(plan_c2r);
  cudaFree(data);

  return(0);
}
```

速度比較

1次元実フーリエ 1000回順・逆変換 (計2000回)

- fftw (Intel Core2Quad Q9550)
- cufft (Nvidia Tesla C1060)

N が大きくなればなるほど
GPUが速い

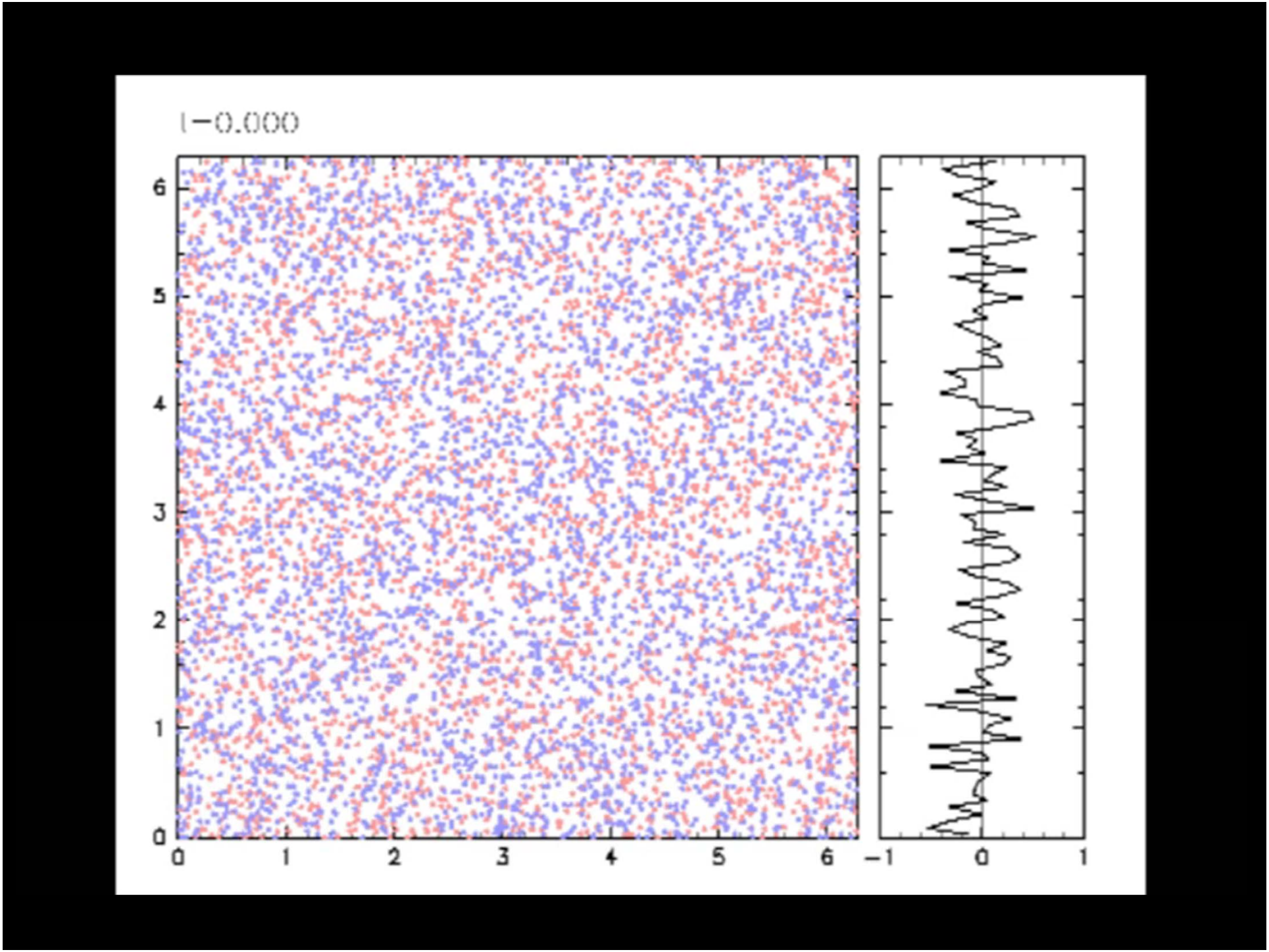


計算事例2

- 2次元点渦
  - 粒子法のようなもの
  - 渦無し流れの中に、面積0 の点渦を配置
  - 点渦から速度場を計算し、点渦を移流させる

$$\psi_i = -\frac{1}{2\pi}\sum_{j\neq i}\Gamma_j \log|\boldsymbol{x}_j - \boldsymbol{x}_i|$$

$$\psi : \text{stream function}, \Gamma : \text{circuration}$$


t=0.000
0
1
2
3
4
5
6
0
1
2
3
4
5
6
−1
0
1

- OpenCL の利用
  - メリット
    - CPUでも, N社GPUでも, A社GPUでも
      - なんだかんだで CPU で動くことは重要 (デバッグがしやすい)
    - 特定の企業に依存しない
  - デメリット
    - 事例/ドキュメントが少ない
    - おきまりの記述が多くて面倒くさい

      → 便利なライブラリを利用する

- Ruby-OpenCL
  - OpenCL(ホストコード部分)の Ruby バインディング
  - OpenCL の 1:1 ラッパー ＋ α (便利な機能)
    - オブジェクト解放から解放
    - 情報取得の簡単化
    - メモリ転送をお任せにすることもできる
  - 他の有用な Ruby ライブラリ群を利用できる
    - IO, 描画, 通信, etc
  - http://ruby-opencl.rubyforge.org/

```
require "quick_opencl"

kernel_source = <<EOF
__kernel void dot_product (__global const float4 *a, __global const float4 *b, __global float *c)
{
  int gid = get_global_id(0);
  c[gid] = dot(a[gid], b[gid]);  // a[0]*b[0] + a[1]*b[1] + a[2]*b[2] + a[3]*b[3];
}
EOF

n = 256
OpenCL::Quick.init
srcA = OpenCL::Quick::VArray.new(OpenCL::VArray::FLOAT4, n)
srcB = OpenCL::Quick::VArray.new(OpenCL::VArray::FLOAT4, n)
dst = OpenCL::Quick::VArray.new(OpenCL::VArray::FLOAT, n)

for i in 0...n
  srcA[i] = OpenCL::Float4.new(i,i,i,i)
  srcB[i] = OpenCL::Float4.new(i,i,i,i)
end

OpenCL::Quick.sources = [kernel_source]
OpenCL::Quick.execute_NDRange("dot_product", [srcA, srcB, dst], [n], [1])
p dst  # => 0.0, 4.0, 16.0, 36.0, …..
```

まとめ

- 今後は超並列時代に
  - ノウハウの蓄積が必要
    - まずは簡単なところから
    - 知識の共有
      - 利用者人口を増やす必要がある
- GPGPU使用事例
  - 2次元乱流
    - お手軽GPGPU化
  - 2次元点渦
    - OpenCL, Ruby-OpenCL の利用

GPGPU の普及に向けて

- “コスト” と “ゲイン” のバランス
  - それなりの労力でまずまずの速度向上
    - チューニング無しでもそれなり
    - Fermi でよりお気軽に?
    - まず GPGPU化してみる
- 将来無駄になるという不安
  - 過渡期であることは間違いない
  - 超並列のノウハウは生き続けるに違いない
- 新しいものへの抵抗感
  - 現状で特に問題ない – 将来でも問題ないか?
  - 新しいものに取り組む時間がない – ・・・・