オブジェクト指向プログラミングに基づいた流体解析プログラムのCUDA Fortran実装の試み

出川智啓（長岡技術科学大学），中道義之（名古屋大学）

研究室運営の悩み

- 学生の卒業と共にコードが死蔵
- コード管理の手間
- コーディングルールを守らない学生
- 手段と目的の混同

FORTRAN 77

- 科学技術計算分野において利用
- 情報処理を専門としない技術者・研究者が利用
- 配列演算の記述が容易
- ポインタの明示的な取り扱いが不要

FortranによるGPGPU

- PGI CUDA Fortranの登場
  - なぜか情報が少ない
- FORTRANからC言語を経由してGPUへ移植
- 可読性や移植性の低さが問題に
  - なってますよね･･･？

コンシューマ向けアプリケーション開発

- C++/C＃/Javaなど
- オブジェクト指向プログラミングを採用
- 拡張性，柔軟性，保守性の向上が可能

OpenFOAMが使われる理由

- オープンソース
- 多機能
- 拡張性
  - オブジェクト指向プログラミングに起因
  - OpenFOAMがC/FORTRAN 77で書かれていたら？

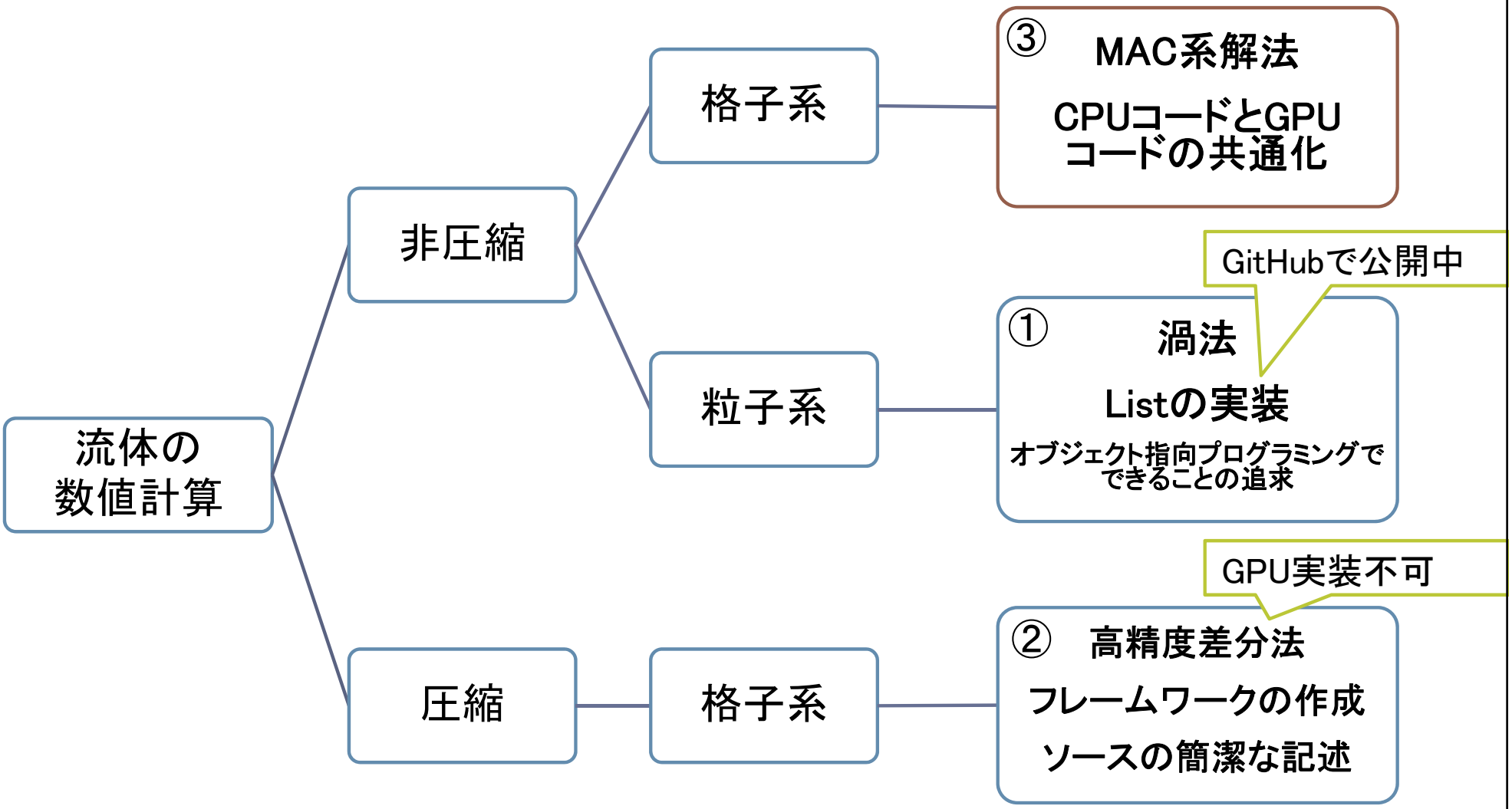
Fortranによるオブジェクト指向プログラミング

- Fortran2003以降で実装
  - Fortran 90/95はObject-Based
- FORTRAN77スタイルのプログラムの取り込み
- 既存のプログラムのシームレスな拡張

目的

- Fortranを用いたオブジェクト指向プログラミング
- 簡単な非圧縮性流体解析プログラムの実装
- PGI CUDA Fortranを用いたGPU実装
- 死蔵されたプログラムを統合する枠組みの作成

概観


流体の
数値計算
非圧縮
圧縮
格子系
粒子系
格子系
③ MAC系解法
CPUコードとGPU
コードの共通化
GitHubで公開中
① 渦法
Listの実装
オブジェクト指向プログラミングで
できることの追求
GPU実装不可
② 高精度差分法
フレームワークの作成
ソースの簡潔な記述

支配方程式

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0$$

連続の式

$$\begin{cases}\dfrac{\partial u}{\partial t}+\dfrac{\partial uu}{\partial x}+\dfrac{\partial vu}{\partial y}=-\dfrac{1}{\rho}\dfrac{\partial p}{\partial x}+\nu\left(\dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 v}{\partial y^2}\right)\\ \dfrac{\partial v}{\partial t}+\dfrac{\partial uv}{\partial x}+\dfrac{\partial vv}{\partial y}=-\dfrac{1}{\rho}\dfrac{\partial p}{\partial y}+\nu\left(\dfrac{\partial^2 v}{\partial x^2}+\dfrac{\partial^2 v}{\partial y^2}\right)\end{cases}$$

Navier-Stokes方程式

x, y ：空間方向　u, v ：x, y方向速度　ν ：動粘度

t ：時間　p ：圧力

Fractional Step法

$$\begin{cases} \tilde{u} = u^n - \Delta t\left[\dfrac{\partial u^n u^n}{\partial x} + \dfrac{\partial u^n v^n}{\partial y} - \dfrac{1}{\mathrm{Re}}\left(\dfrac{\partial^2 u^n}{\partial x^2} + \dfrac{\partial^2 u^n}{\partial y^2}\right)\right] \\ \tilde{v} = v^n - \Delta t\left[\dfrac{\partial u^n v^n}{\partial x} + \dfrac{\partial v^n v^n}{\partial y} - \dfrac{1}{\mathrm{Re}}\left(\dfrac{\partial^2 v^n}{\partial x^2} + \dfrac{\partial^2 v^n}{\partial y^2}\right)\right] \end{cases}$$

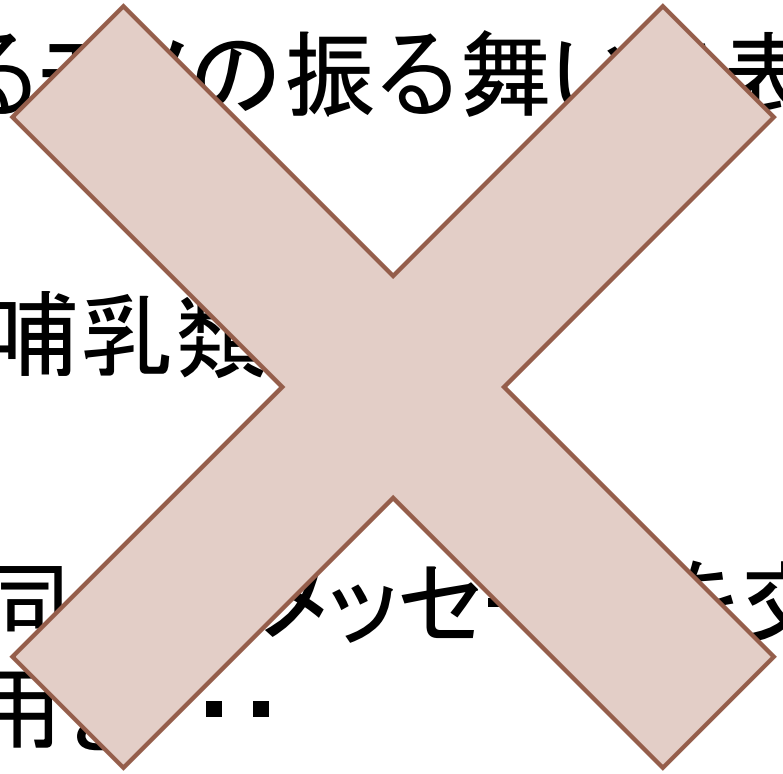
$$\frac{\partial^2 p^{n+1}}{\partial x^2} + \frac{\partial^2 p^{n+1}}{\partial y^2} = \frac{1}{\Delta t}\left(\frac{\partial \tilde{u}}{\partial x} + \frac{\partial \tilde{v}}{\partial y}\right)$$

$$\begin{cases} u^{n+1} = \tilde{u} - \Delta t \dfrac{\partial p^{n+1}}{\partial x} \\ v^{n+1} = \tilde{v} - \Delta t \dfrac{\partial p^{n+1}}{\partial y} \end{cases}$$

$\tilde{u}, \tilde{v}$ ：x, y方向中間速度

オブジェクト指向プログラミング

- この世にある[illegible]の振る舞い[illegible]表現する
- イヌもネコも哺乳類[illegible]
- オブジェクト同[illegible]メッセ[illegible]を交換しあいながら相互作用[illegible]・・



Fortranによるオブジェクト指向プログラミング

- プログラミング方法論の一つ
- 関係するデータと処理を一括して取り扱う
- 派生型type(*)にサブルーチンを追加

用語の対応表

Fortran	Java	C++
derived type	class	class
component	field	data member
type bound procedure	method	virtual member function
parent type	super class	base class
extended type	sub class	extended class
module	package	namespace

何をしたいか

- 流れ場を変える際の手直しを少なく
- パラメータ設定
- 境界条件
- GPUへの移植を容易に

どのように書きたいか

境界条件を動的に決定

```
Parameter cavity    = new Parameter(x, y, t, new BC(m,w,w,w))
FlowFiled fs_cavity = new FlowField(cavity, new SOLVER(ε, α))

fs_cavity.run()
```

Poissonソルバの動的な決定
必要なパラメータの設定

- 流れ場に関するパラメータを一括管理
  - 時間・空間離散化，境界条件
- 計算法に関するパラメータと関数を一括管理

メインプログラム

```
program main
    use :: ...
    implicit none

    type(parameter1d)                :: x,y,t
    type(FieldParameter2d),pointer :: cavity
    type(FractionalStep)  ,pointer :: fs_cavity
    !条件の定義
    x = parameter1d(...)
    y = parameter1d(...)
    t = parameter1d(...)
    cavity => new_FieldParameter2d( x, y, t )
    !計算方法の定義
    fs_cavity => new_FractionalStep(cavity, 1d-9, 1.925d0 )
    call fs_cavity%run()

end program main
```

境界条件の動的な決定は未実装

Poissonソルバの動的な決定は未実装

Fractional Step型の定義

```
type ::  FractionalStep
    type(FieldParameter2d),private,pointer :: parameter
    type(VectorField2d)   ,private,pointer :: velo
    type(ScalarField2d)   ,private,pointer :: pres
    type(VectorField2d)   ,private,pointer :: velo_aux
    type(ScalarField2d)   ,private,pointer :: dive

    real(8),private :: err_tolerance  !反復解法の許容誤差
    real(8),private :: SOR_accel      !SOR用加速係数
contains
    procedure,public,pass :: run
    procedure,public,pass :: computeAuxiliaryVelocity !中間速度
    procedure,public,pass :: computePressure          !圧力(SOR)
    procedure,public,pass :: computeVelocity          !速度
end type FractionalStep
```

ベクトル量（u,v）の定義

```
type ::  VectorField2d
    type(FieldParameter2d),pointer :: parameter
    real(8),public,pointer :: x(:,:) !x方向成分
    real(8),public,pointer :: y(:,:) !y方向成分

contains

    procedure,public,pass :: DirichletBoundary
    procedure,public,pass ::   NeumannBoundary
    procedure,public,pass :: output

end type VectorField2d
```

スカラ量（p）の定義

```
type ::  ScalarField2d
    type(FieldParameter2d),pointer :: parameter
    real(8),public,pointer :: v(:,:) !スカラ量の値

contains

    procedure,public,pass :: DirichletBoundary
    procedure,public,pass ::   NeumannBoundary
    procedure,public,pass :: output

end type ScalarField2d
```

パラメータ型の定義

- x,y,t方向の離散化
  - 長さ，離散点数，離散点間隔
- 物性値
  - 密度，動粘度
- 計算パラメータ
  - レイノルズ数，代表長さ，代表速度

計算実行procedure

```
subroutine run(this)
    implicit none
    class(FlactionalStep) :: this

    integer :: n,Nt
    Nt = this%parameter%t%getNumberOfPoints()

    do n=1,Nt
        print *,n,Nt
        call this%computeAuxiliaryVelocity()
        call this%computePressure()
        call this%computeVelocity()
    end do
    call this%output()

end subroutine run
```

中間速度計算procedure

```
subroutine computeAuxiliaryVelocity(this)
    implicit none
    class(FlactionalStep) :: this

    integer :: i,j,Nx,Ny
    real(8) :: dx,dy,dt,dxdx,dydy

    do j=1,Ny-1
    do i=1,Nx
        !中間速度のx方向成分を計算
    end do
    end do
    !中間速度のy方向成分を計算
    call this%velo%DirichletBoundary()
    !中間速度の発散を計算
end subroutine computeAuxiliaryVelocity
```

圧力計算procedure

```
subroutine computePressure(this)
    implicit none
    class(FlactionalStep) :: this
    integer :: i,j,Nx,Ny
    real(8) :: dx,dy,dt,dxdx,dydy,dp

    do while(err > err_tolerance) !Red & Black SORで圧力を収束
        do j=1          ,Ny-1
        do i=1+mod(j,2),Nx-1,2
            dp = ...
        end do
        end do
        call this%pres%NeumannBoundary()
    end do
    call this%pres%DirichletBoundary()
end subroutine computePressure
```

速度計算procedure

```
subroutine computeVelocity(this)
    implicit none
    class(FlactionalStep) :: this

    integer :: i,j,Nx,Ny
    real(8) :: dx,dy,dt

    do j=1,Ny-1
    do i=1,Nx
        !中間速度のx方向成分を修正し，速度を計算
    end do
    end do
    !中間速度のy方向成分を修正し，速度を計算
    call this%velo%DirichletBoundary()
end subroutine computeVelocity
```

CUDA FortranによるGPU実装

ベクトル量型のGPU実装

```
type ::  VectorField2d
    type(FieldParameter2d),pointer :: parameter
    real(8),public,allocatable,device :: x(:,:) !x方向成分
    real(8),public,allocatable,device :: y(:,:) !y方向成分
    !派生型内でpointer,deviceは宣言できない
contains

    !GPU kernelをtype-bound procedureとして持つことができない
    !procedure,public,pass :: DirichletBoundary
    !procedure,public,pass ::   NeumannBoundary
    procedure,public,pass :: output

end type VectorField2d
```

GPU実装に関わる制約

- GPU kernelをtype-bound procedureとして持つことができない
  - 持てるが(コンパイル可能だが)実行できない

```
!宣言
procedure,public,pass :: DirichletBoundary

!定義
attributes(global) subroutine DirichletBoundary
end subroutine DirichletBoundary

!呼び出し
call this%velo%DirichletBoundary<<<Block, Thread>>>()
                                                ここでエラー
```

GPU実装に関わる制約

- GPU kernelをtype-bound procedureとして持つことができない
  - 持てるが(コンパイル可能だが)実行できない

```
!宣言
procedure,public,pass :: DirichletBoundary<<<Block, Thread>>>
                                                    ここでエラー

!定義
attributes(global) subroutine DirichletBoundary
end subroutine DirichletBoundary

!呼び出し
call this%velo%DirichletBoundary()
```

スカラ量型のGPU実装

```
type ::  ScalarField2d
    type(FieldParameter2d),pointer :: parameter
    real(8),public,allocatable,device :: v(:,:) !スカラ量の値

contains

    !procedure,public,pass :: DirichletBoundary
    !procedure,public,pass ::   NeumannBoundary
    procedure,public,pass :: output

end type ScalarField2d
```

Fractional Step型のGPU実装（変更なし！）

```
type ::  FractionalStep
    type(FieldParameter2d),private,pointer :: parameter
    type(VectorField2d)   ,private,pointer :: velo
    type(ScalarField2d)   ,private,pointer :: pres
    type(VectorField2d)   ,private,pointer :: velo_aux
    type(ScalarField2d)   ,private,pointer :: dive

    real(8),private :: err_tolerance  !反復解法の許容誤差
    real(8),private :: SOR_accel      !SOR用加速係数
contains
    procedure,public,pass :: run
    procedure,public,pass :: computeAuxiliaryVelocity !中間速度
    procedure,public,pass :: computePressure          !圧力(SOR)
    procedure,public,pass :: computeVelocity          !速度
end type FractionalStep
```

計算実行procedure （変更なし！）

```
subroutine run(this)
    implicit none
    class(FlactionalStep) :: this

    integer :: n,Nt
    Nt = this%parameter%t%getNumberOfPoints()

    do n=1,Nt
        print *,n,Nt
        call this%computeAuxiliaryVelocity()
        call this%computePressure()
        call this%computeVelocity()
    end do
    call this%output()

end subroutine run
```

中間速度計算procedure

```
subroutine computeAuxiliaryVelocity(this)
    use :: module_kernel
    implicit none
    class(FlactionalStep) :: this

    integer :: i,j,Nx,Ny
    real(8) :: dx,dy,dt,dxdx,dydy

    !別モジュールで定義したGPU kernelを呼び出し
    !既存のGPU kernelを流用可能，Fortran77のsubroutineも流用可能
    call gpuComputeAuxiliaryVelocity<<<Block,Thread>>> &
                                (this%velo%x,this%velo%y, &
                                 this%velo_aux%x,this%velo_aux%y,…)

end subroutine computeAuxiliaryVelocity
```

圧力計算procedure

```
subroutine computePressure(this)
    use :: module_kernel
    implicit none
    class(FlactionalStep) :: this
    logical :: isConverged = .false.

    do while(.not.isConverged)
       isConverged=gpuComputePressure(this%pres%v,this%dive%v,…)
    end do

end subroutine computePressure
```

圧力計算procedure

```
function gpuComputePressure(pres, div, …) result(isConverged)
    use :: cudafor
    use :: moudle_kernel
    implicit none
    real(8),device :: pres(:,:), div(:,:)
    logical        :: isConverged
    logical,device :: conv

    conv = .true.
    call gpuComputePoissonR<<<Block, Thread>>>(pres,dive,…,conv)
    call gpuComputePoissonB<<<Block, Thread>>>(pres,dive,…,conv)
    stat = cudaThreadSynchronize()
    isConverged = conv

end function gpuComputePressure
```

圧力計算procedure

```
attributes(global)subroutine gpuComputePoissonR(pres,div,…,conv)
    implicit none
    real(8),device :: pres(:,:), div(:,:)
    logical,device :: conv
    integer :: i,j

    j = (blockIdx%y-1)*blockDim%y + threadIdx%y
    i = (blockIdx%x-1)*blockDim%x + threadIdx%x +1+mod(j,2)

    dp = ...
    if(abs(dp) > err_tolerance) conv = .false.
    !L∞ノルムで収束判定

end function gpuComputePoissonR
```

速度計算procedure

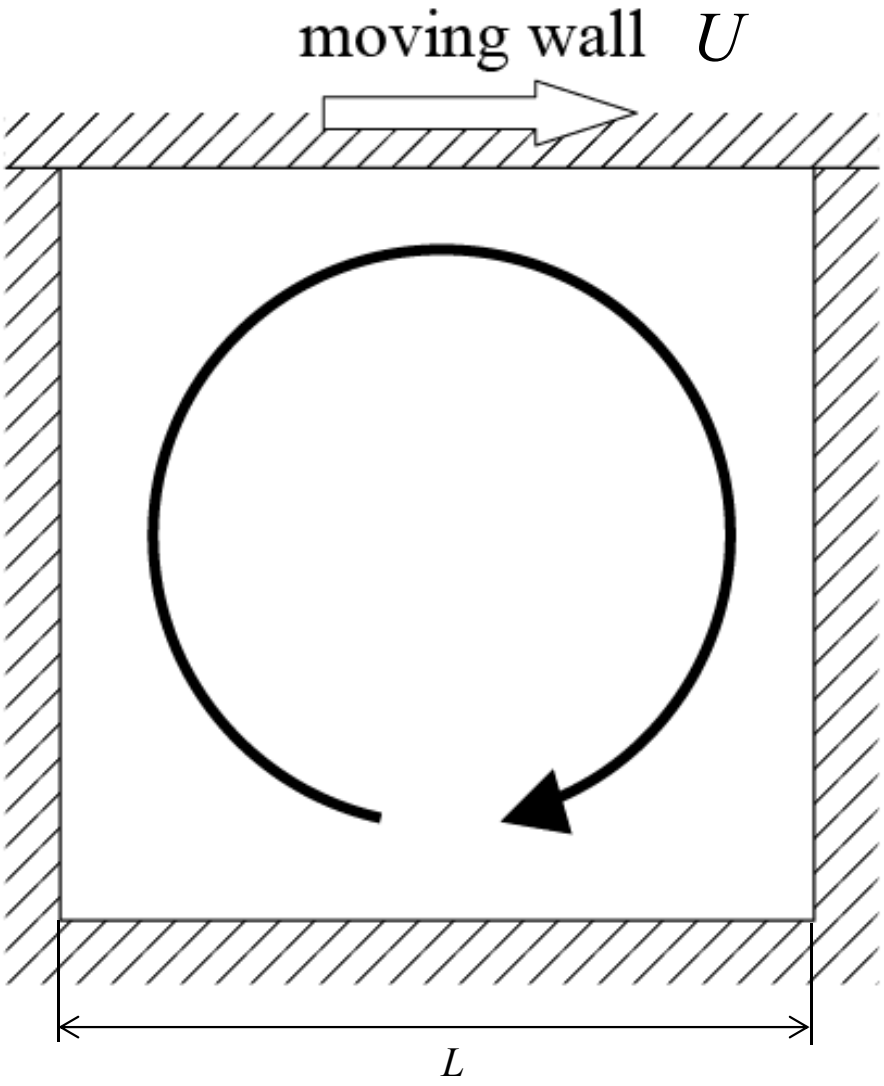
```
subroutine computeVelocity(this)
    implicit none
    class(FlactionalStep) :: this

    integer :: i,j,Nx,Ny
    real(8) :: dx,dy,dt

    !別モジュールで定義したGPU kernelを呼び出し
    !既存のGPU kernelを流用可能，Fortran77のsubroutineも流用可能
    call gpuComputeVelocity<<<Block,Thread>>>
                        (this%velo%x,this%velo%y, this%press%v,…)

end subroutine computeVelocity
```

メインプログラム（変更無し！）

```
program main
    use :: ...
    implicit none

    type(parameter1d)                :: x,y,t
    type(FieldParameter2d),pointer :: cavity
    type(FractionalStep)  ,pointer :: fs_cavity

    x = parameter1d(...)
    y = parameter1d(...)
    t = parameter1d(...)
    cavity => new_FieldParameter2d( x, y, t )

    fs_cavity => new_FractionalStep(cavity, 1d-9, 1.925d0 )
    call fs_cavity%run()

end program main
```

Cavity流れへの適用

計算条件



レイノルズ数　Re=UL/ν=1000

格子分割数　256×256

計算時間間隔 $U\Delta t/\Delta x$=0.1

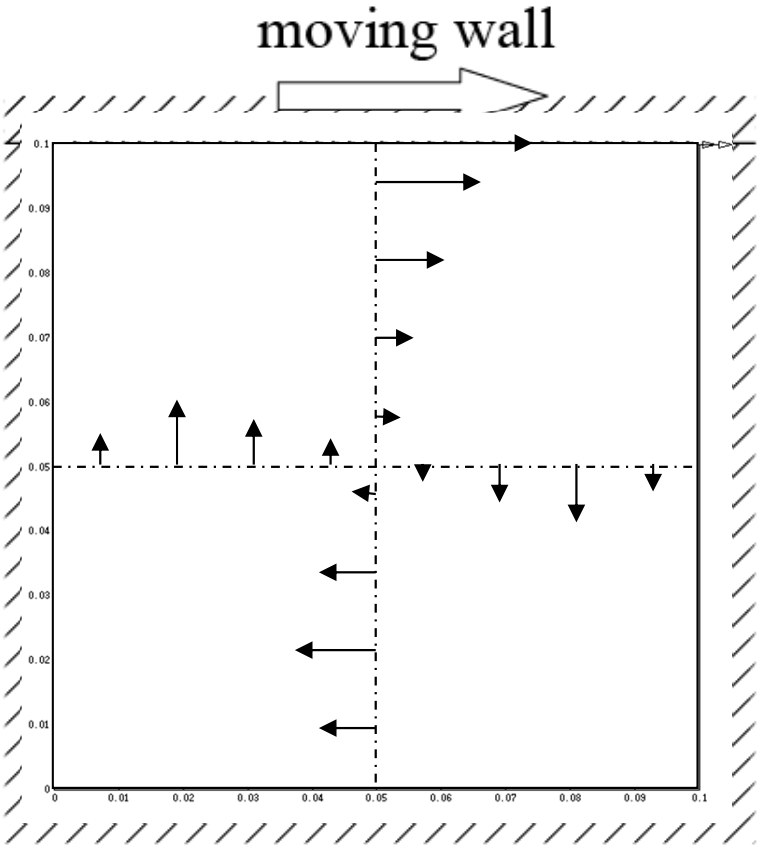
計算環境

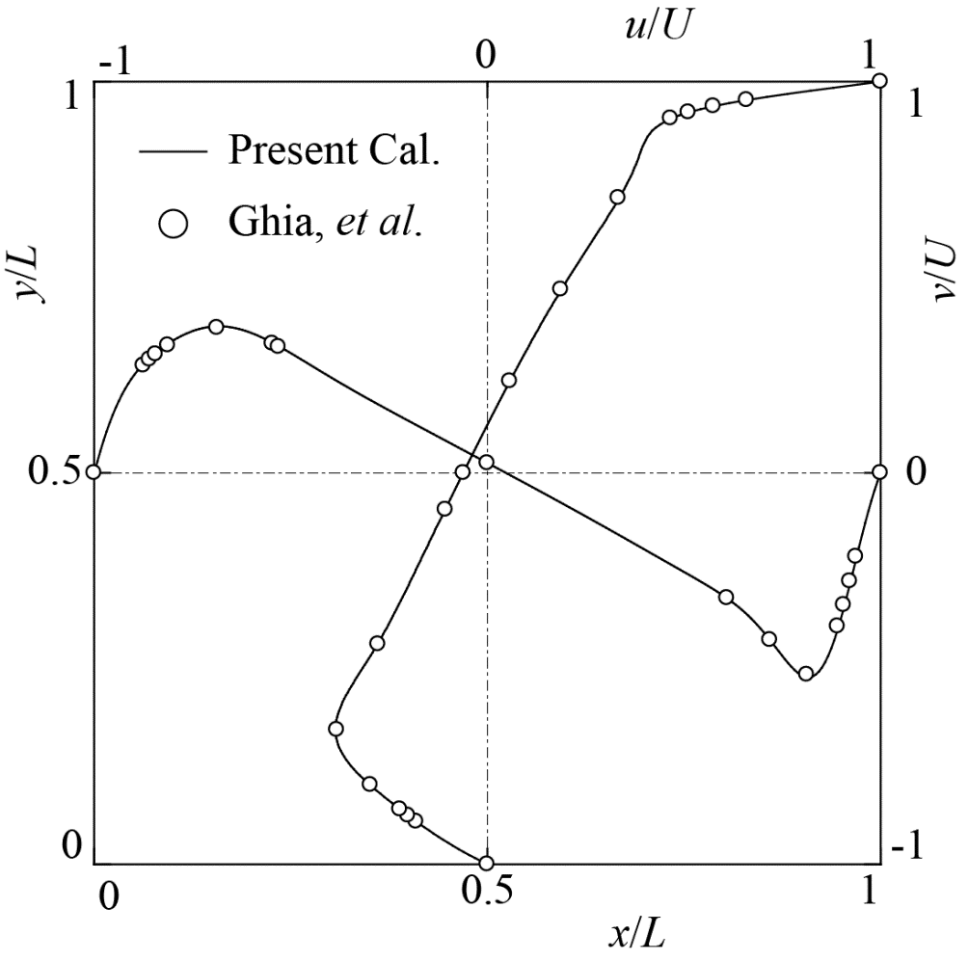
CPU : Core i5 2500
GPU : GTX Titan

コンパイラ

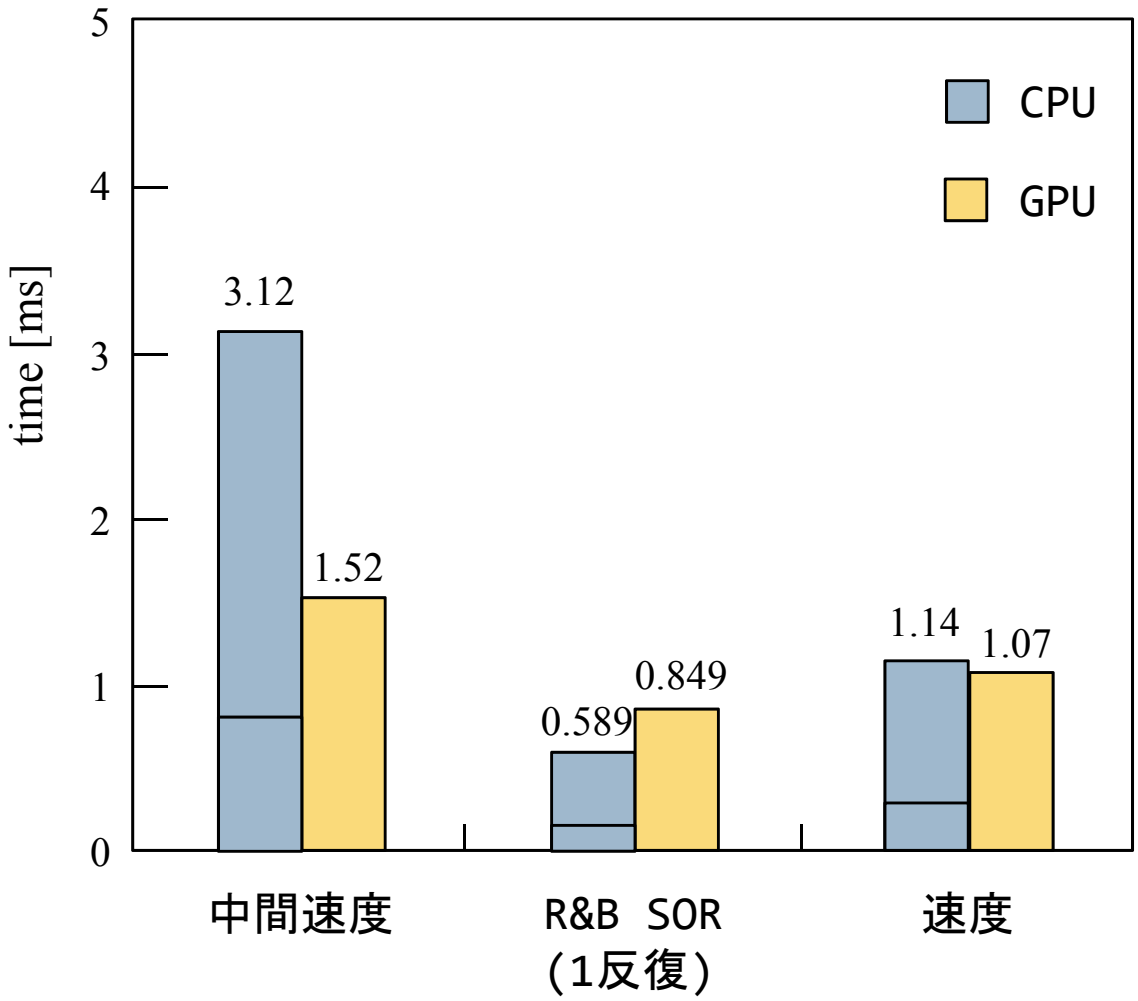
Intel Fortran Composer 2013 update 5
PGI Accelerator Fortran 14.9
CUDA 6.0

計算結果

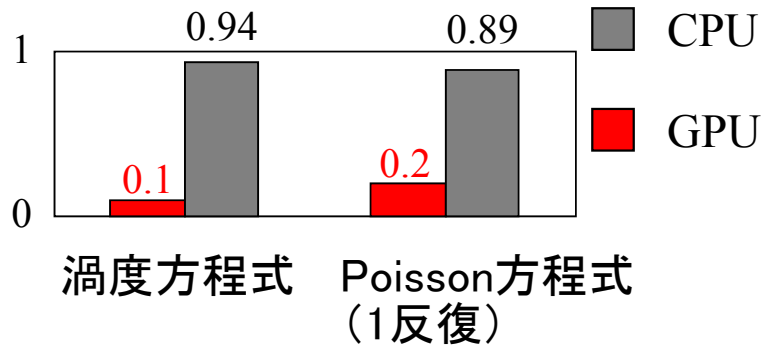




1ステップあたりの計算時間



- CPUと同程度
- チューニング不足
- オブジェクト指向プログラミングが原因ではない



※オープンCAE学会第1回並列計算セミナー

まとめ

- Fortran2003を用い，非圧縮性流体解析プログラムを実装
- PGI CUDA Fortranを用いてGPUへ移植
- CUDA Fortranの制限
- 移植における影響範囲を限定