CUDA Fortranを用いた
圧縮性･非圧縮性流体の差分法計算

出川智啓
電気通信大学
情報理工学研究科
知能機械工学専攻

発表内容

- 背景
  - GPUを用いた流体計算
  - CUDA Fortran
- GPUを用いた流体計算の実装
  - 圧縮性流れ
    - 差分計算の方法による実行速度の変化
    - CPUとの実行時間の比較
    - CUDA Cとの実行時間の比較
  - 非圧縮性流れ
    - 流れ関数ー渦度法（マイナー解法ですが・・・）

流体の数値計算に対する立場

- 専門は流体の数値計算
  - 気液二相流（気泡流）の数値計算法の開発・高精度化
  - 高解像度差分法を用いた圧縮性流体の直接数値計算
- 計算法の開発・適用や実装が主
  - 計算法の開発・適用7：現象の解析3
  - プログラミング言語はFortran 90/95を使用
  - 最適化はあまりやらない
    - コンパイラ頼み
- 並列計算の経験は約2年
  - OpenMP（2年），GPGPU（1年半），MPI（1年未満）
- OSはWindowsを使用
  - Windowsでできそうなことは全部Windowsで

背景

- GPUを用いた流体の差分法計算
  - 圧縮性流体 ― GPU向き
    - 比較的初期から計算例が報告
    - 支配方程式は移流拡散型
    - 差分計算以外は1点完結
    - 衝撃波等の不連続捕獲を行うとメモリの読み書きに対する計算量が増加
  - 非圧縮性流体
    - 非圧縮条件を満たすために連立方程式を解く
      - 連立方程式がボトルネック
    - 近年，多くの適用例が報告

背景

- GPUで実行するためのプログラム開発
  - CUDA, ATI Stream SDK, OpenCL, OpenGL+GLSL…
  - Fortranでの開発は
    - CUDA Fortran

CUDA Fortran使用前

Fortran ⇨ C ⇨ CUDA C

CUDA Fortran使用後

Fortran ⇨ CUDA Fortran

目的

CUDA Fortranを用いて問題を解き，使い勝手やCUDA Cとの違いなどについて報告

CUDA Fortran

- Fortranの拡張
  - PGI Fortran 10.0以降で利用可能
    - PGIアクセラレータコンパイラ製品
    - ※マイナーバージョンによって動作が異なる
  - CUDA Cと比較してコーディングが簡単
    - エラー処理を考えなければ，CPUでプログラムを組む様な感覚
  - CUDA Cとの違い
    - Block IDとThread IDが1から開始
    - 配列添字は任意の範囲で確保可能
      - ○`a(1:100)`，○`a(0:99)`，○`a(-50:50)`
    - Texture memoryは使用不可

CUDA Cのサンプルプログラム

add.c

```c
#define N (512)

void add_CPU(float *a, float *b, float *c){

    for(int i=0; i<N; i++){
         c[i] = a[i] + b[i];
    }
}

void init_CPU(float *a, float *b, float *c){

    for(int i=0;i<N;i++){
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }
}

int main(){

    float *a,*b,*c;

    a = (float *) malloc( N*sizeof(float) );
    b = (float *) malloc( N*sizeof(float) );
    c = (float *) malloc( N*sizeof(float) );
    init_CPU(a, b, c);
    add_CPU(a, b, c);

    free(a);
    free(b);
    free(c);
    return 0;
}
```

add.cu

```c
#define N (512)

__global__ void add_GPU(float *a, float *b, float *c){

    int i = blockIdx.x*blockDim.x + threadIdx.x;
        c[i] = a[i] + b[i];

}

void init_GPU(float *a, float *b, float *c){

    int i = blockIdx.x*blockDim.x + threadIdx.x;
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;

}

main(){

    float *a,*b,*c;

    cudaMalloc((void **) &a, N*sizeof(float) );
    cudaMalloc((void **) &b, N*sizeof(float) );
    cudaMalloc((void **) &c, N*sizeof(float) );
    init_GPU<<< 1, N >>>(a, b, c);
    add_GPU<<< 1, N >>>(a, b, c);

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
    return 0;
}
```

CUDA C専用関数

BlockやThreadのIDと配列添字を対応

CUDA Fortranのサンプルプログラム

add.f90

```
module kernel
    implicit none
    contains
    subroutine add_CPU(a,b,c,n)
        implicit none

        integer,value :: n
        real :: a(n),b(n),c(n)
        integer :: i

        do i=1,n
            c(i) = a(i)+b(i)
        end do
     end subroutine add_CPU
end module kernel

program add
    use kernel
    implicit none

    integer,parameter :: n = 512
    real,allocatable :: a(:), b(:), c(:)

    allocate(a(n)); a = 1.0
    allocate(b(n)); b = 2.0
    allocate(c(n)); c = 0.0

    call add_CPU(a, b, c, n)

    deallocate(a)
    deallocate(b)
    deallocate(c)

end program add
```

メモリ属性を指定するため，コンパイラが容易に判断

add.cuf

```
module kernel
    implicit none
    contains
    attributes(global) subroutine add_GPU(a,b,c,n)
        implicit none

        integer,value :: n
        real :: a(n),b(n),c(n)
        integer :: i

        i = (blockIdx%x-1)*blockDim%x + threadIdx%x
            c(i) = a(i)+b(i)

     end subroutine add_GPU
end module kernel

program add
    use kernel
    implicit none

    integer,parameter :: n = 512
    real,allocatable,device :: a(:), b(:), c(:)

    allocate(a(n)); a = 1.0
    allocate(b(n)); b = 2.0
    allocate(c(n)); c = 0.0

    call add_GPU<<<1,n>>>(a, b, c, n)

    deallocate(a)
    deallocate(b)
    deallocate(c)

end program add
```

BlockやThreadのIDと配列添字を対応

CUDA Cよりも簡潔な記述が可能
（※エラー処理を考えなければ）

GPUを用いた流体の差分法計算

- 圧縮性流体
  - 一般には高速な気流を対象
    - 航空機，高速車両，ジェットエンジン，燃焼 等
    - 解法の開発は，衝撃波などの不連続捕獲と高い空間解像度の両立が主流
    - 空力音の直接数値計算など，低マッハ数でも利用
  - 比較的初期から計算例が報告
  - 支配方程式は移流拡散型
  - 差分計算以外は1点完結
  - 衝撃波等の不連続捕獲を行うとメモリの読み書きに対する計算量が増加
    - 衝撃波は考慮しない（正確には可変密度流体）

支配方程式

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0$$ （質量保存式）

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} + \frac{\partial p}{\partial x_i} = \frac{\partial \tau_{ij}}{\partial x_j}$$ （運動量保存式）

$$\frac{\partial \rho E}{\partial t} + \frac{\partial \rho u_i E}{\partial x_i} + \frac{\partial u_i p}{\partial x_i} = \frac{\partial u_i \tau_{ij}}{\partial x_i} - \frac{\partial q_i}{\partial x_i}$$ （エネルギ保存式）

空間離散化：2次精度中心差分

時間離散化：4次精度Runge-Kutta法

ρ :密度　p :圧力　u :速度　E:全エネルギ=$u_k u_k/2+\varepsilon$　ε :内部エネルギ=$p/\{\rho(\gamma-1)\}$

q :熱流束　τ_{ij} :粘性応力 $= \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial u_k}{\partial x_k}\right)$　$\mu = \mu_0\left(\frac{T_{ref} + T_{\text{const.}}}{T + T_{\text{const.}}}\right)\left(\frac{T}{T_0}\right)^{\frac{3}{2}}$

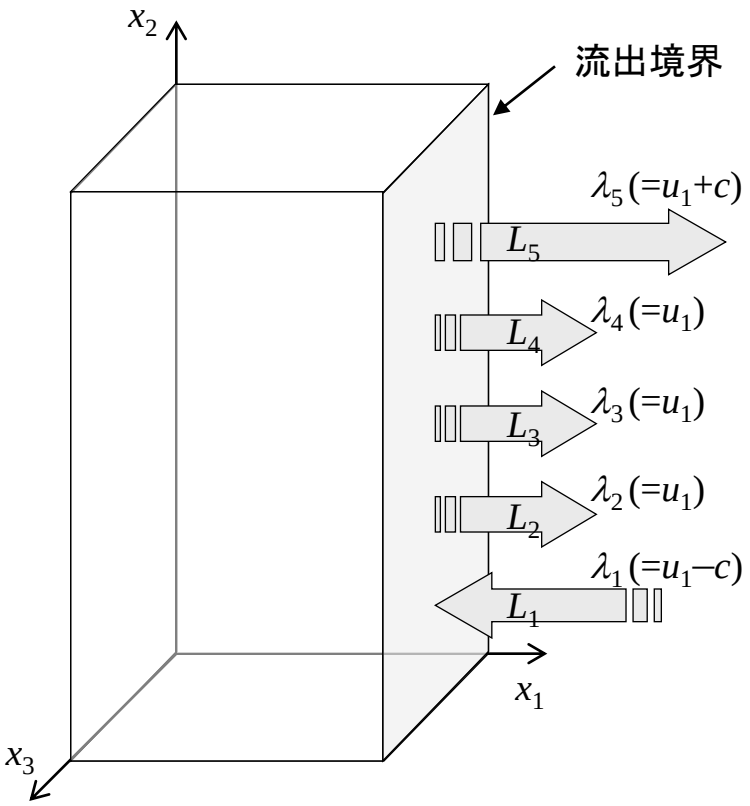
境界条件

- 圧縮性流れの数値計算における境界条件
  - 局所境界条件
    - 境界となる格子点において，計算領域内部と異なる手続きを行う
    - 特性波解析に基づく境界条件
      - Navier-Stokes Characteristic Boundary Condition(1)
      - 空間多次元無反射境界条件
  - 領域境界条件
    - 計算領域外に付加的な領域を設け，その領域内において，計算領域内部と異なる手続きを行う
    - 人工的な拡散項，移流項の付加，フィルタによる反射波の減衰

(1) Poinsot, T.J. and Lele, S.K., J. Comput. Phys., 101, 104-129, 1992.

NSCBC(Poinsot-Lele,1992)

- Navier-Stokes Characteristic Boundary Condition (NSCBC)
  - Euler方程式の特性波解析に基づいた境界条件をNavier-Stokes方程式に拡張
  - 実装が容易であり，経験的に頑健であることが知られている



特性波の振幅の時間変化

$$\begin{pmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \\ L_5 \end{pmatrix} = \begin{pmatrix} \lambda_1\left(\frac{\partial p}{\partial x_1} - \rho c \frac{\partial u_1}{\partial x_1}\right) \\ \lambda_2\left(c^2 \frac{\partial \rho}{\partial x_1} - \frac{\partial p}{\partial x_1}\right) \\ \lambda_3 \frac{\partial u_2}{\partial x_1} \\ \lambda_4 \frac{\partial u_3}{\partial x_1} \\ \lambda_5\left(\frac{\partial p}{\partial x_1} + \rho c \frac{\partial u_1}{\partial x_1}\right) \end{pmatrix}$$

特性波の速度

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \end{pmatrix} = \begin{pmatrix} u_1 - c \\ u_1 \\ u_1 \\ u_1 \\ u_1 + c \end{pmatrix}$$

$\lambda_i > 0$のとき，特性波は計算領域から外向きに伝播→特性波の時間変化L_iを計算領域内部から推定

$\lambda_i < 0$のとき，特性波は計算領域外から入射→特性波の時間変化L_iを0

(1) Poinsot, T.J. and Lele, S.K., J. Comput. Phys., 101, 104-129, 1992.

NSCBCを適用した支配方程式

境界となる格子点(x_1=一定)において用いる式

$$\frac{\partial \rho}{\partial t} + d_1 + \frac{\partial \rho u_2}{\partial x_2} + \frac{\partial \rho u_3}{\partial x_3} = 0$$

（質量保存式）

$$\begin{cases} \dfrac{\partial \rho u_1}{\partial t} + u_1 d_1 + \rho d_3 + \dfrac{\partial \rho u_1 u_2}{\partial x_2} + \dfrac{\partial \rho u_1 u_3}{\partial x_3} = \dfrac{\partial \tau_{1j}}{\partial x_j} \\ \dfrac{\partial \rho u_2}{\partial t} + u_2 d_1 + \rho d_4 + \dfrac{\partial \rho u_2 u_2}{\partial x_2} + \dfrac{\partial \rho u_2 u_3}{\partial x_3} + \dfrac{\partial p}{\partial x_2} = \dfrac{\partial \tau_{2j}}{\partial x_j} \\ \dfrac{\partial \rho u_3}{\partial t} + u_3 d_1 + \rho d_5 + \dfrac{\partial \rho u_3 u_2}{\partial x_2} + \dfrac{\partial \rho u_3 u_3}{\partial x_3} + \dfrac{\partial p}{\partial x_3} = \dfrac{\partial \tau_{3j}}{\partial x_j} \end{cases}$$

（運動量保存式）

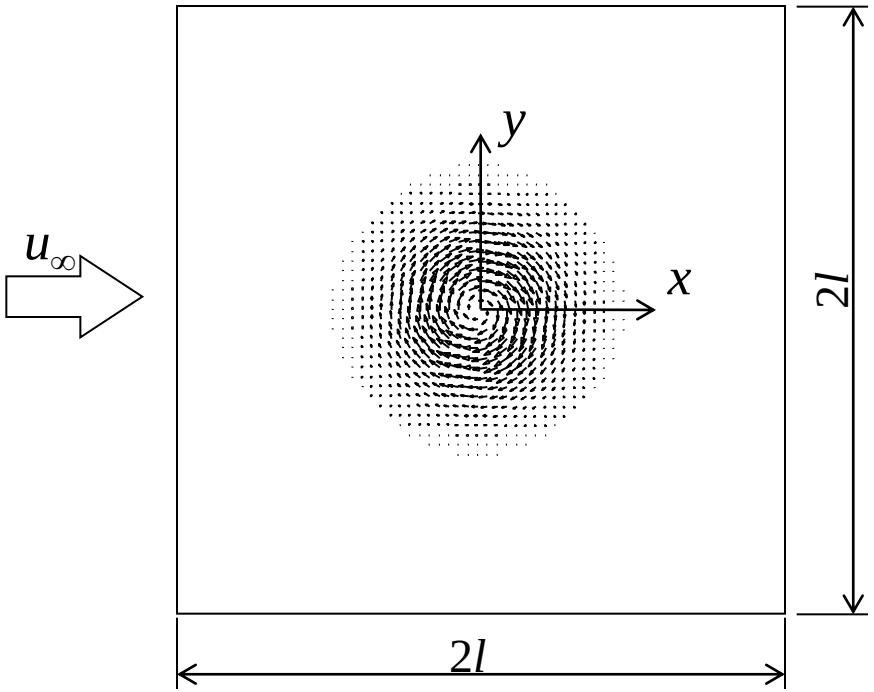
$$\frac{\partial \rho E}{\partial t} + \frac{1}{2}(u_k u_k) d_1 + \frac{d_2}{\gamma - 1} + \rho u_1 d_3 + \rho u_2 d_4 + \rho u_3 d_5 + \frac{\partial \rho u_2 E}{\partial x_2} + \frac{\partial \rho u_3 E}{\partial x_3} + \frac{\partial u_2 p}{\partial x_2} + \frac{\partial u_3 p}{\partial x_3} = \frac{\partial u_j \tau_{ij}}{\partial x_i} - \frac{\partial q_i}{\partial x_i}$$

（エネルギ保存式）

$$\begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{pmatrix} = \begin{pmatrix} \dfrac{1}{c^2}\left[L_2 + \dfrac{1}{2}(L_5 + L_1) \right] \\ \dfrac{1}{2}(L_5 + L_1) \\ \dfrac{1}{2\rho c}(L_5 - L_1) \\ L_3 \\ L_4 \end{pmatrix}$$

計算対象

- 超音速一様流によって流出する単一渦(1)



格子分割数: 512×512

計算時間間隔: $c_\infty \Delta t/l=10^{-4}$

$c_\infty t/l=2$まで2000回時間進行

条件

マッハ数: $\mathrm{Ma}=u_\infty/c_\infty=1.1$

レイノルズ数: $\mathrm{Re}=u_\infty l/\nu=10000$

単一渦

$$\psi = \Gamma \exp\left(-\frac{x^2+y^2}{2r_c^2}\right)$$

$$p - p_\infty = -\rho \frac{\Gamma^2}{2r_c^2} \exp\left(-\frac{x^2+y^2}{r_c^2}\right)$$

$\Gamma/(c_\infty l)= -0.0005$

$r_c/l=0.15$

(1) Poinsot, T.J. and Lele, S.K., J. Comput. Phys., 101, 104-129, 1992.

CUDA Fortranでの実装

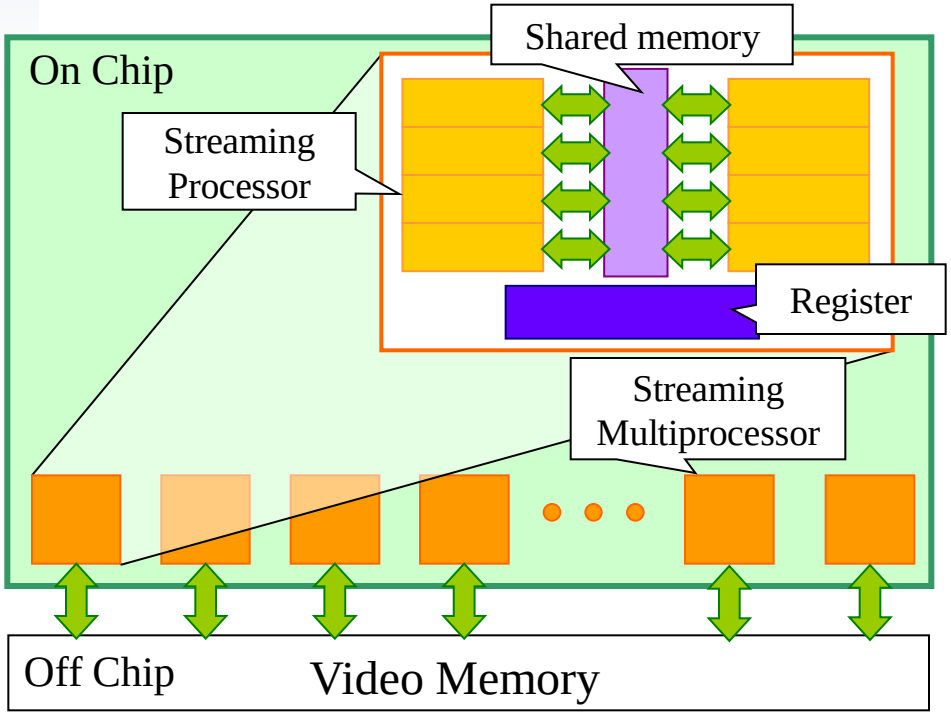
- 1 Threadが1格子点の計算を担当
- *x*方向と*y*方向の差分はkernelを分割する
  - 小川・青木(2)の用いた，*x*方向差分と*y*方向差分を同時に計算する方法は使用しない
- 物理量の微少な変化を捉えるため，計算は倍精度で行う

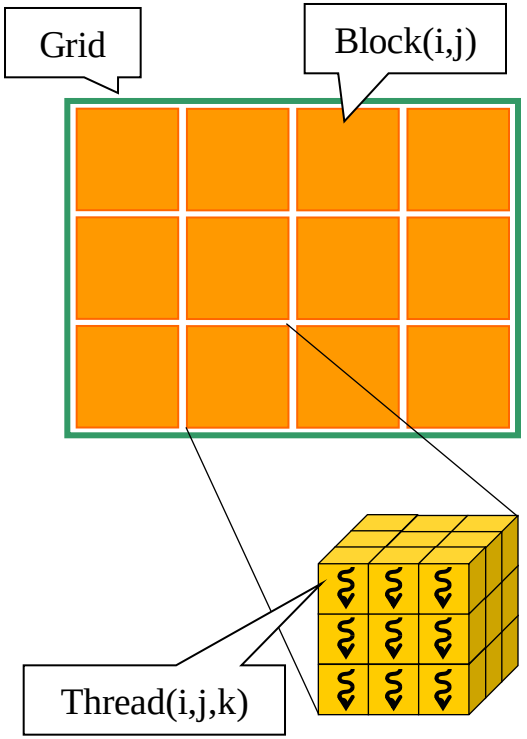
- 計算環境
  - CPU：Core i7 920
  - OS：Windows HPC Server 2008 (64bit)
  - GPU：NVIDIA Tesla C1060
  - CUDA 2.3 (for Windows Vista 64bit)
  - PGI アクセラレータコンパイラ 10.9 (for Windows 64bit)

(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

GPUの模式図

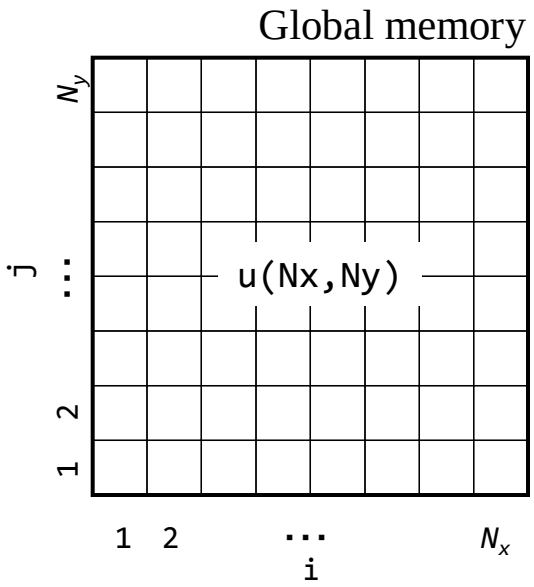
- NVIDIA Tesla C1060
  - 30基のマルチプロセッサを搭載
  - 各マルチプロセッサにストリーミングプロセッサ8基と共有メモリを搭載
- CUDA 2.3
  - GPUにおけるスレッドグループの階層化（Grid, Block, Thread），スレッド同期，共有メモリ操作を行う



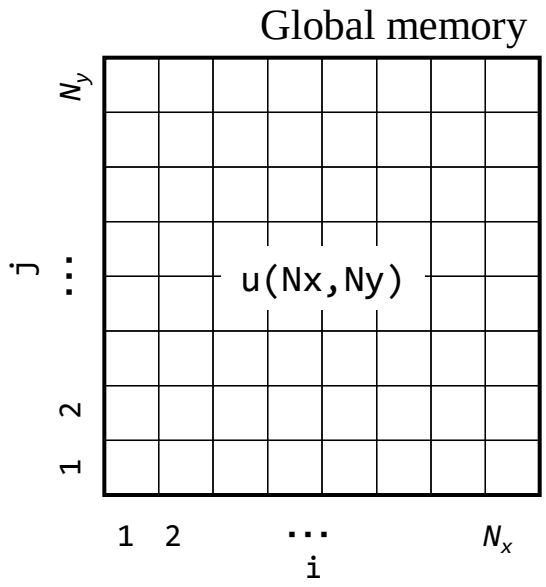


1階差分計算(Case1)

x_1方向差分 $\dfrac{\partial u}{\partial x_1} \approx \dfrac{u_{i+1,\,j} - u_{i-1,\,j}}{2\,\Delta x_1}$

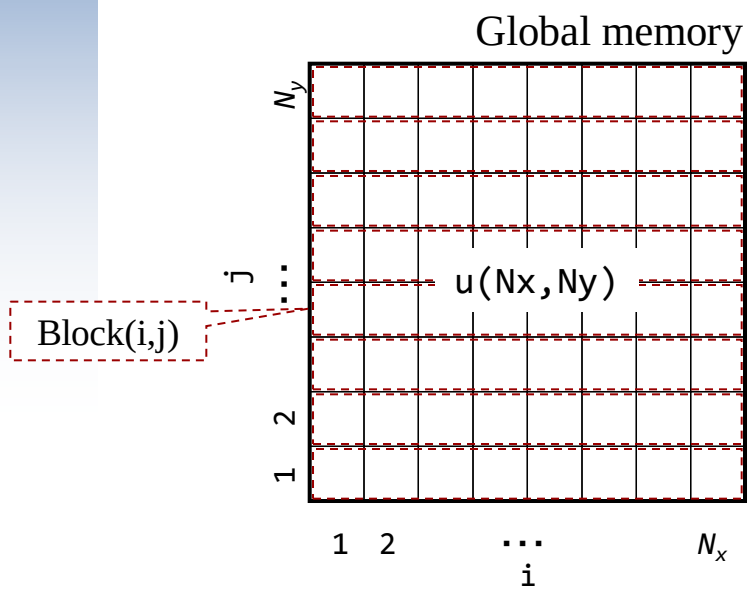


x_2方向差分 $\dfrac{\partial u}{\partial x_2} \approx \dfrac{u_{i,\,j+1} - u_{i,\,j-1}}{2\,\Delta x_2}$

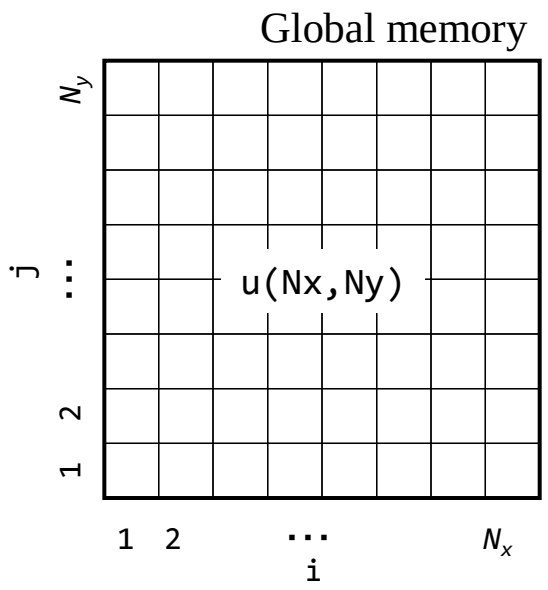


1階差分計算(Case1)

x_1方向差分 $\frac{\partial u}{\partial x_1} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x_1}$

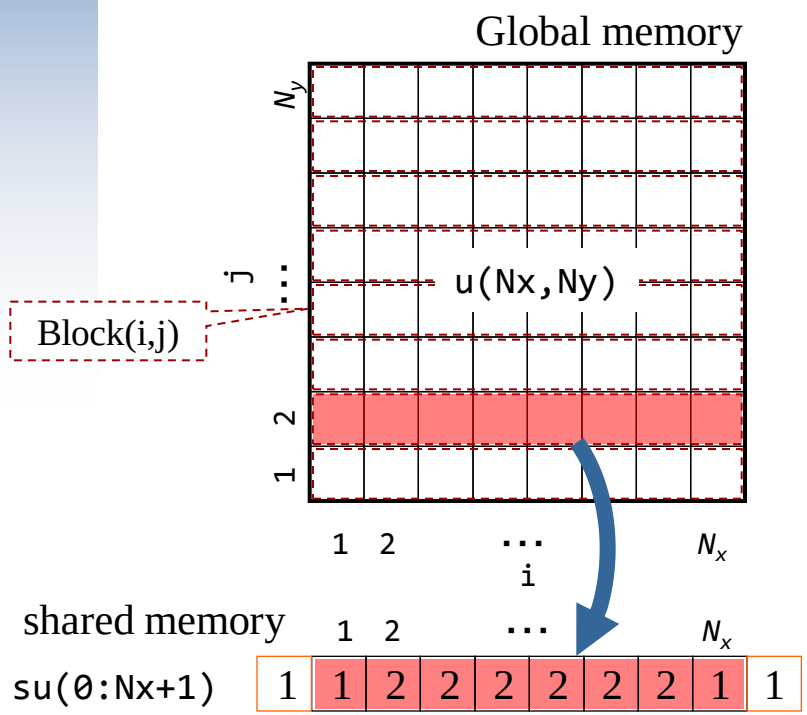


x_2方向差分 $\frac{\partial u}{\partial x_2} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta x_2}$

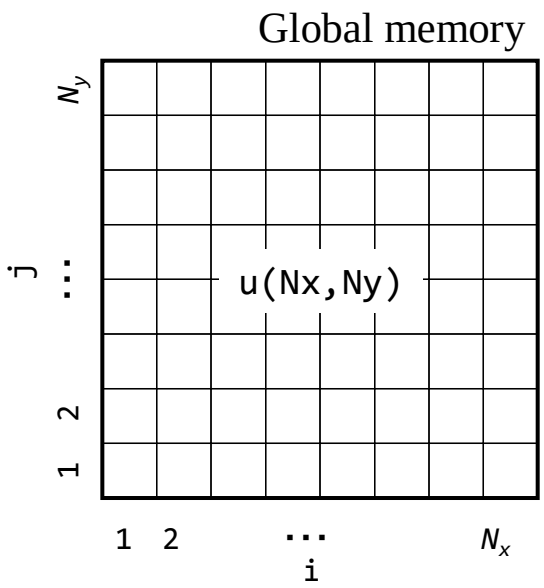


1階差分計算(Case1)

x_1方向差分 $\frac{\partial u}{\partial x_1} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x_1}$

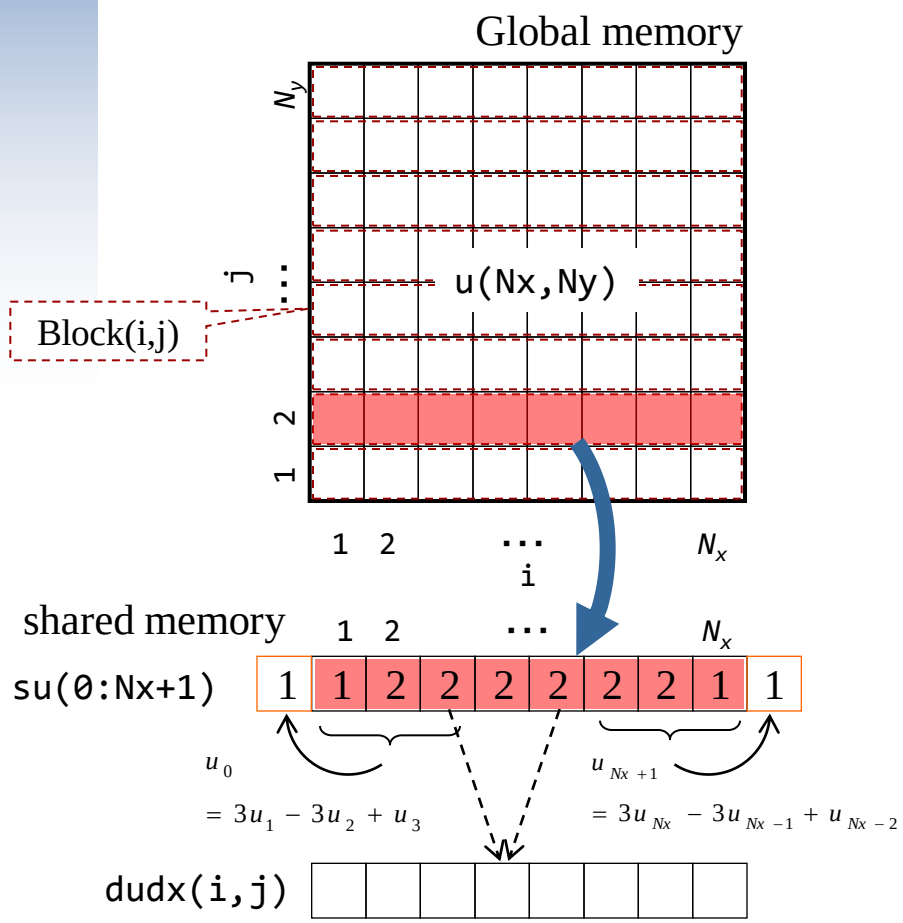


x_2方向差分 $\frac{\partial u}{\partial x_2} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta x_2}$

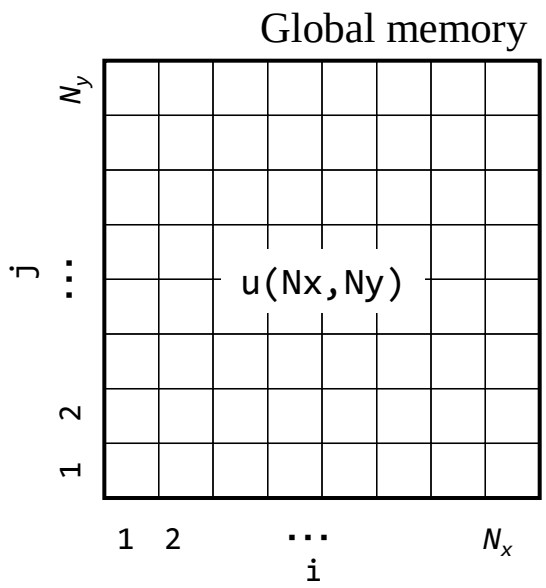


1階差分計算(Case1)

x_1方向差分 $\dfrac{\partial u}{\partial x_1} \approx \dfrac{u_{i+1,\,j} - u_{i-1,\,j}}{2\,\Delta x_1}$

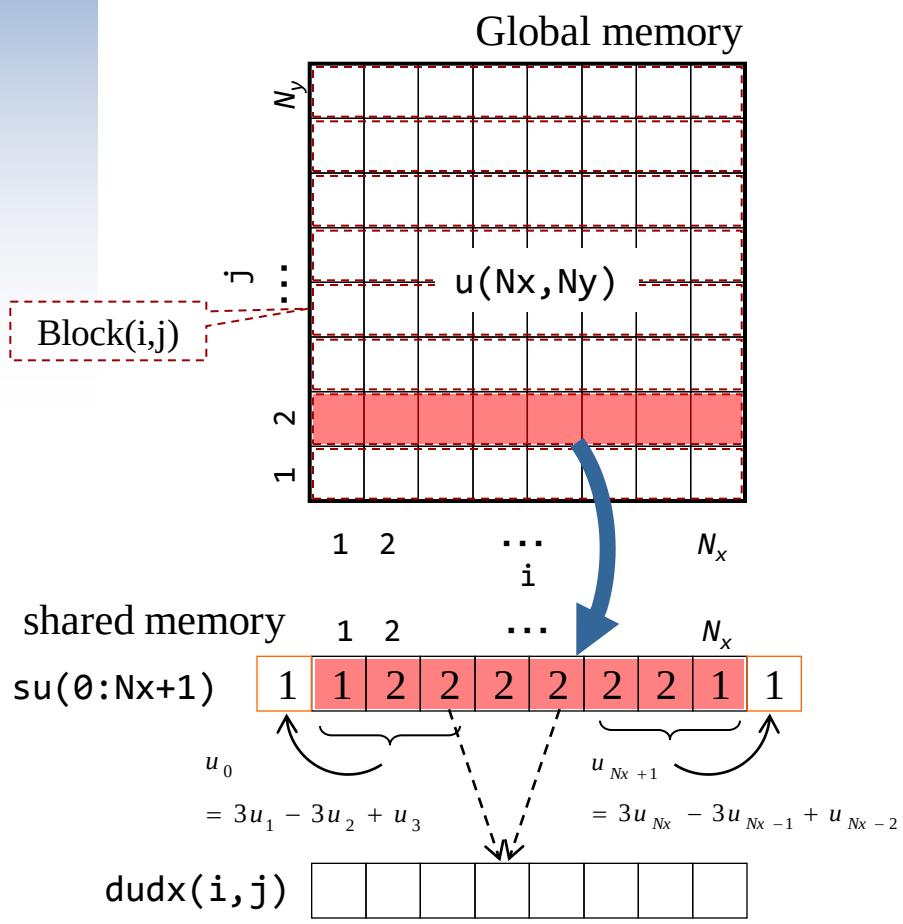


x_2方向差分 $\dfrac{\partial u}{\partial x_2} \approx \dfrac{u_{i,\,j+1} - u_{i,\,j-1}}{2\,\Delta x_2}$

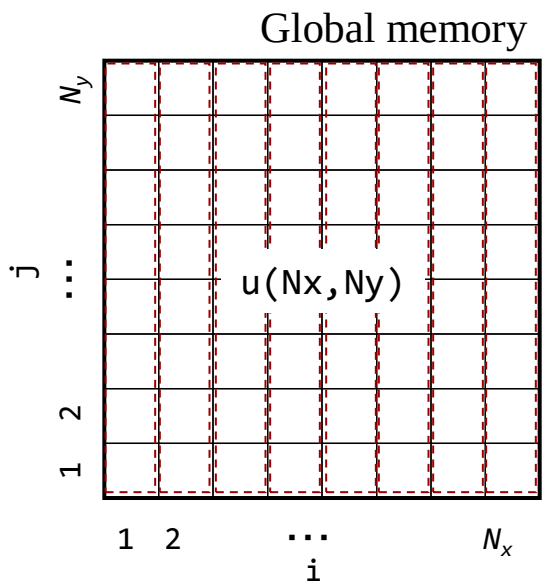


1階差分計算(Case1)

x_1方向差分 $\frac{\partial u}{\partial x_1} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x_1}$

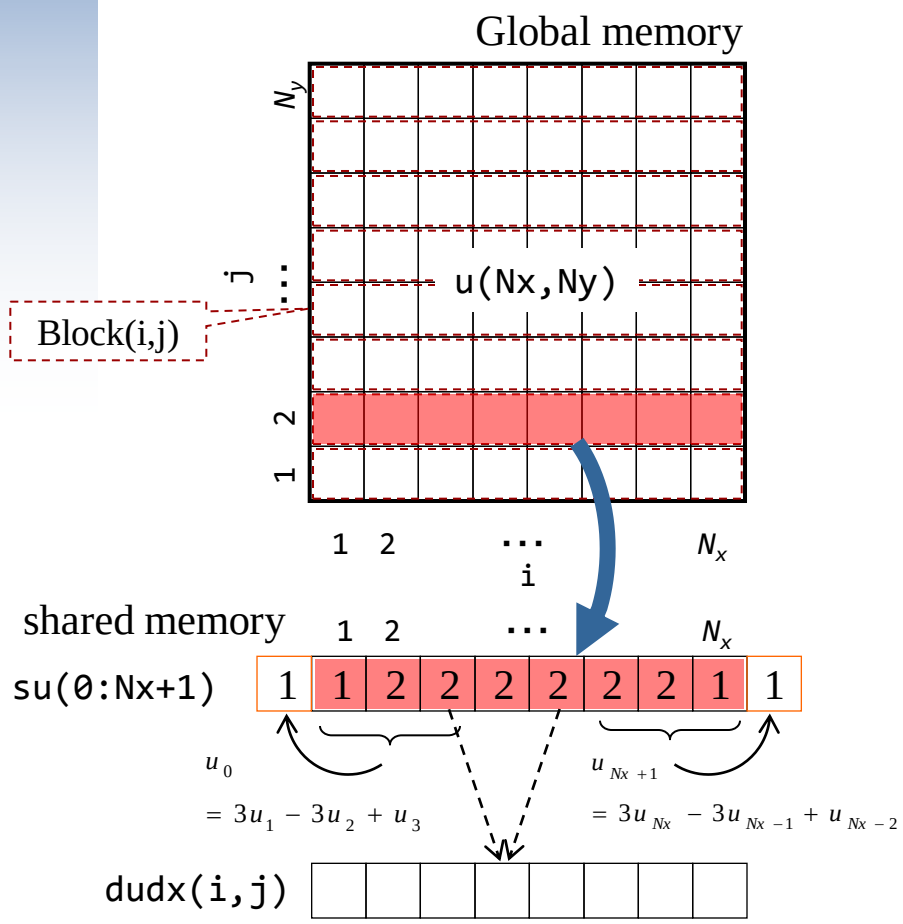


x_2方向差分 $\frac{\partial u}{\partial x_2} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta x_2}$

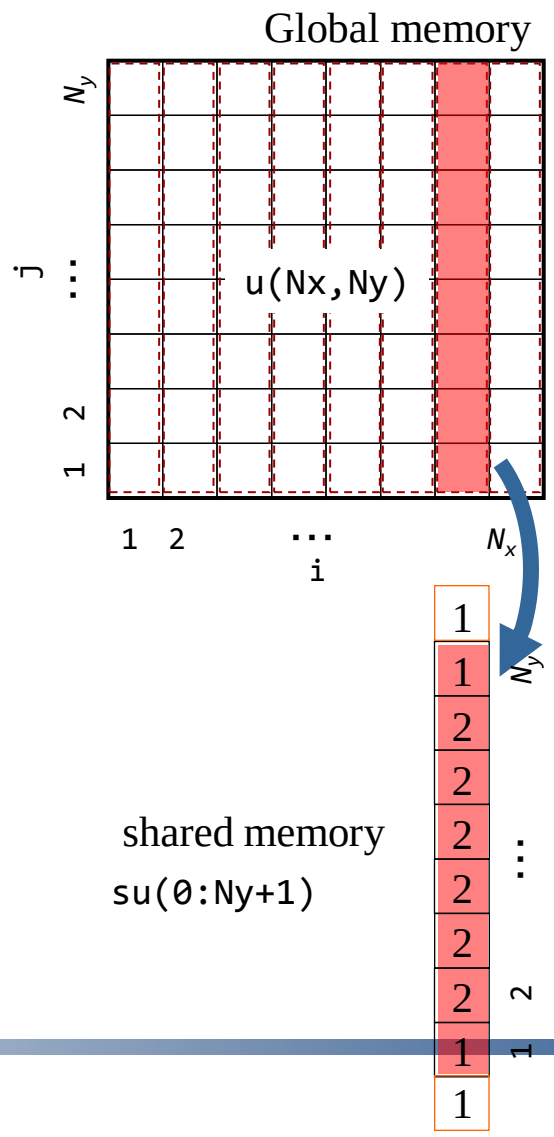


1階差分計算(Case1)

x_1方向差分 $\frac{\partial u}{\partial x_1} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x_1}$

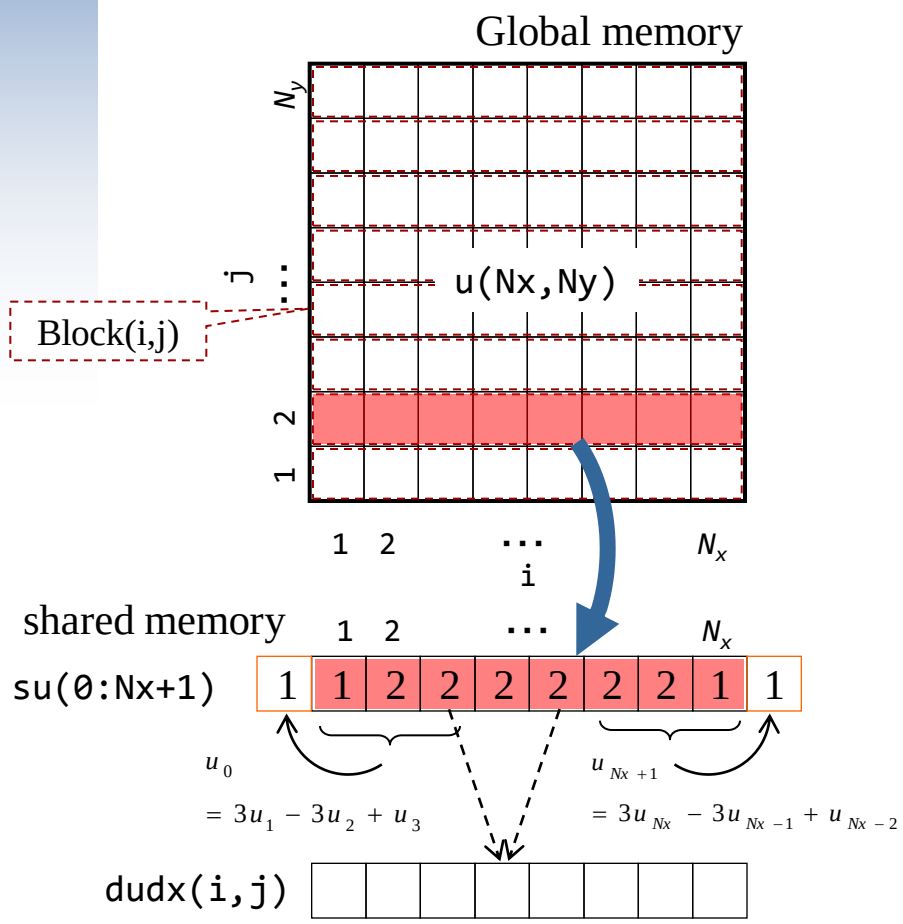


x_2方向差分 $\frac{\partial u}{\partial x_2} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta x_2}$

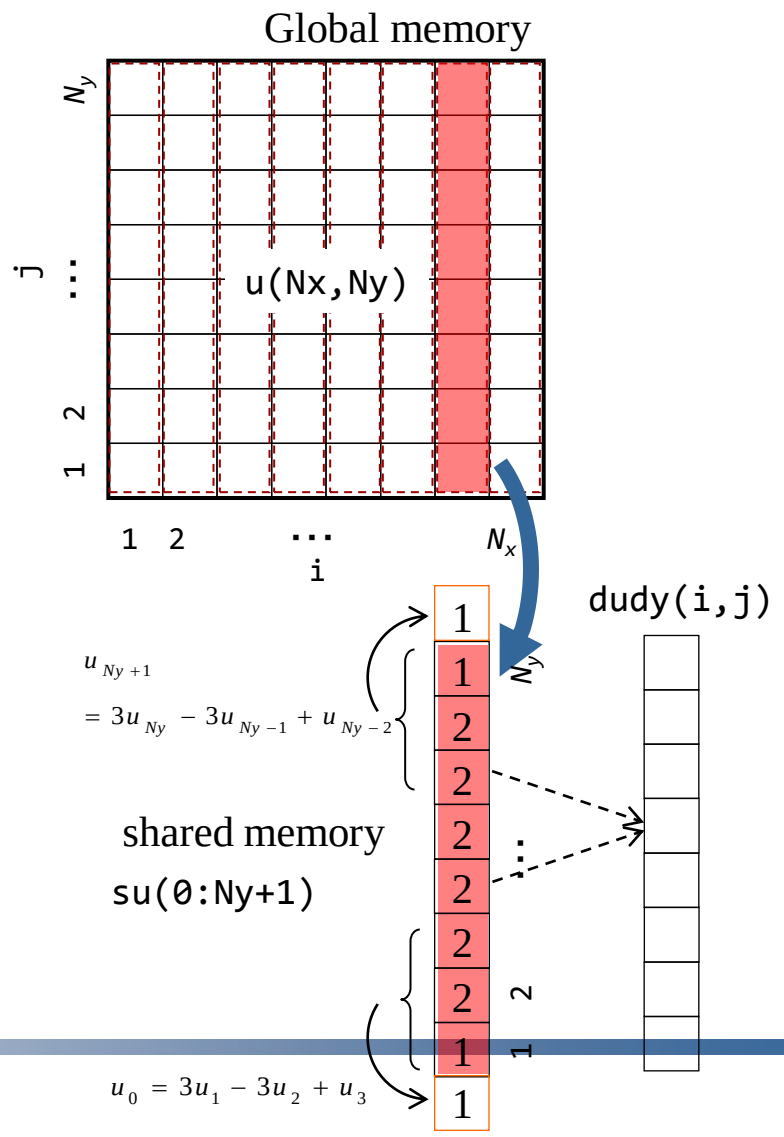


1階差分計算(Case1)

x_1方向差分 $\frac{\partial u}{\partial x_1} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x_1}$



x_2方向差分 $\frac{\partial u}{\partial x_2} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta x_2}$

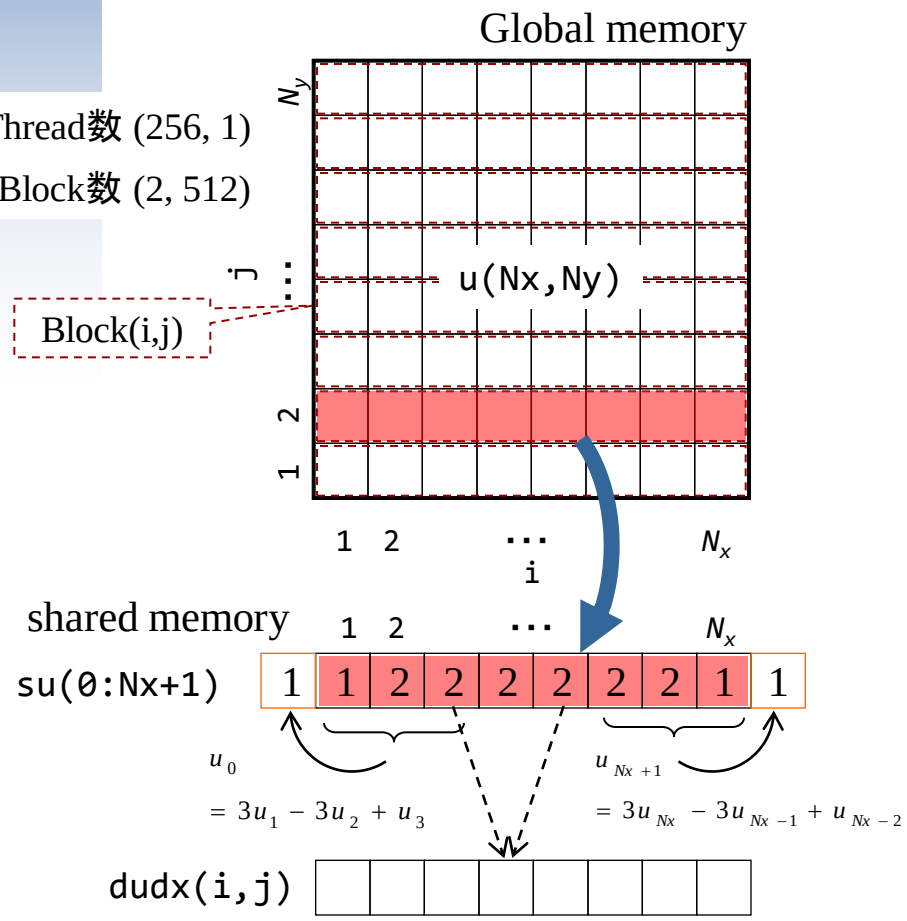


1階差分計算(Case1)

x_1方向差分 $\frac{\partial u}{\partial x_1} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x_1}$

Thread数 (256, 1)

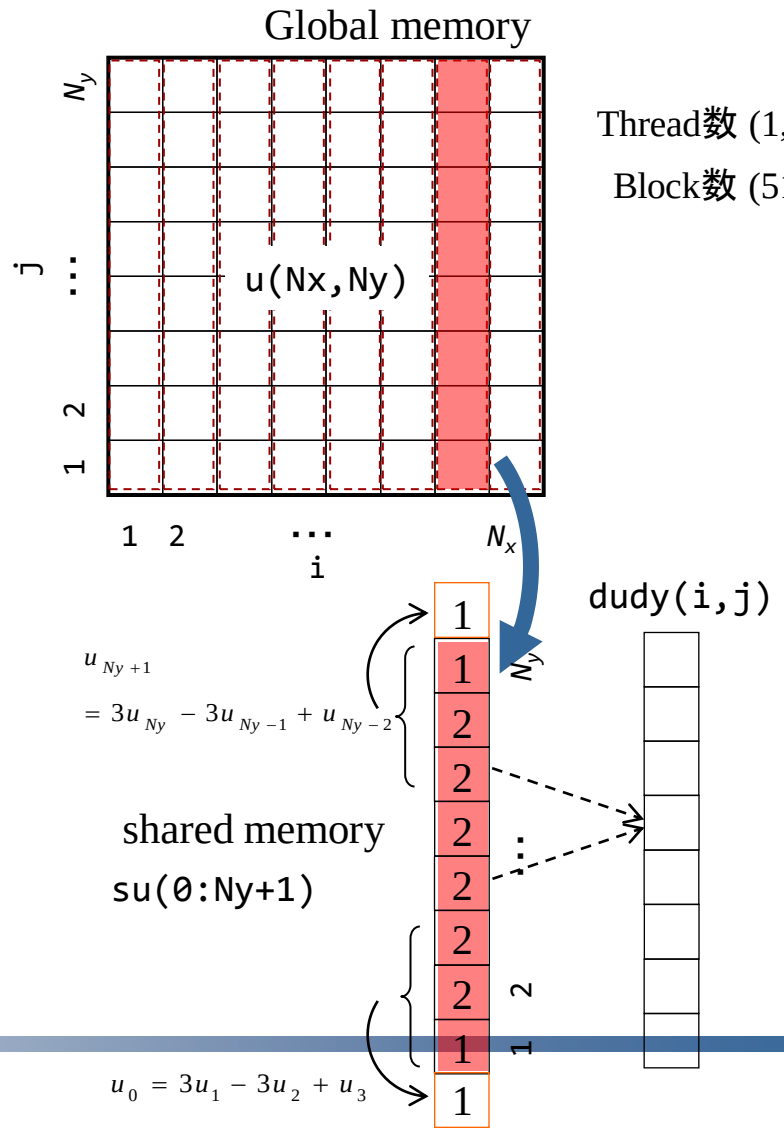
Block数 (2, 512)



x_2方向差分 $\frac{\partial u}{\partial x_2} \approx \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta x_2}$

Thread数 (1, 256)

Block数 (512, 2)



shared memoryの利用の際に

CUDA Fortr

配列添字を任意の範囲で宣言

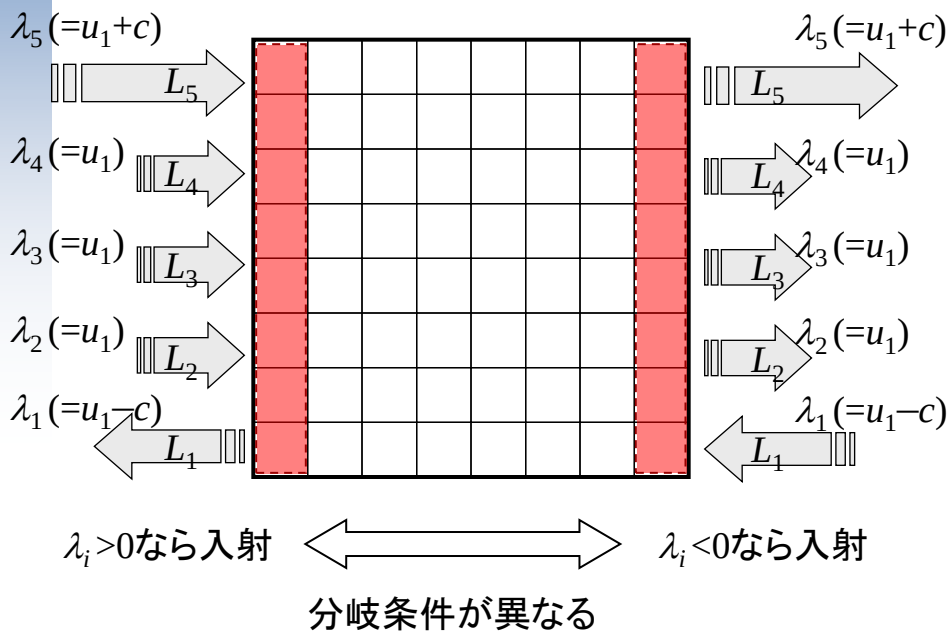
```
real         ::  u(Nx,Ny)
real,shared  :: su(0:Nx+1)
integer :: i,j,tx

tx= threadIdx%x
i = (blockIdx%x-1)*blockDim%x + threadIdx%x
j = (blockIdx%y-1)*blockDim%y + threadIdx%y

!shared memoryにコピー
su(tx) = u(i,j)
call syncthreads()
!袖領域の値を設定
if(i== 1) su(tx-1) = ...
if(i==Nx) su(tx+1) = ...
call syncthreads()

dudx(i,j) = (-su(tx-1)+su(tx+1))/(2*dx)
```

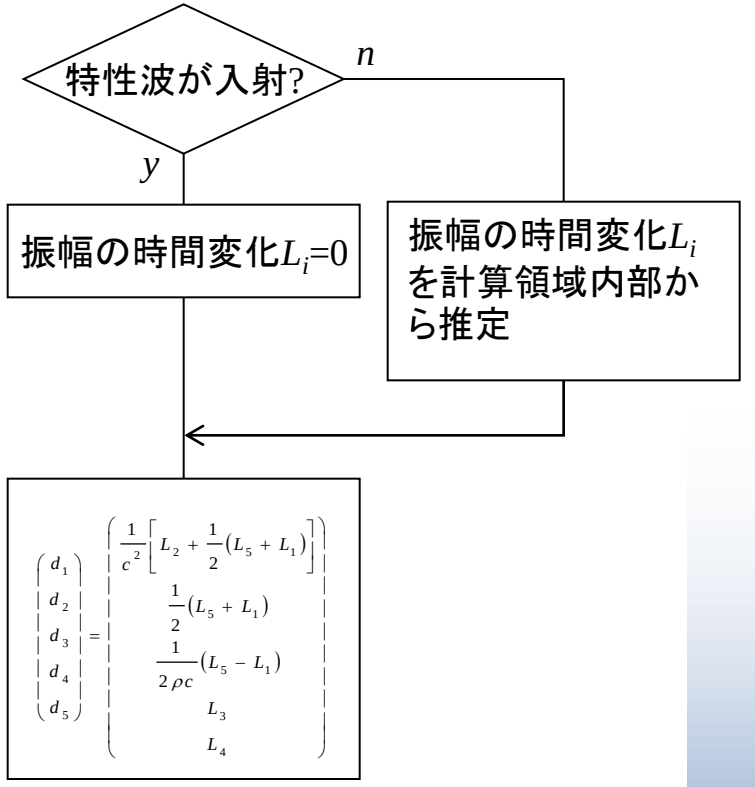
CUDA C

```
          float  u[Nx*Ny];
__shared__ float su[Nx+2]; //+2は袖領域
int i,j,tx;

tx= threadIdx.x +1;
i = blockIdx.x*blockD   .x
j = blockIdx.y*blockDi

//shared memoryにコピー
su[tx] = u[i+Nx*j];
__syncthreads();
//袖領域の値を設定
if(i==   0) su[tx-1] = ...
if(i==Nx-1) su[tx+1] = ...
__syncthreads();

dudx[i+Nx*j] = (-su[tx-1]+su[tx+1])/(2*dx);
```

袖領域を考慮するため，Thread IDと配列添字の対応がずれる

境界(NSCBC)用kernel



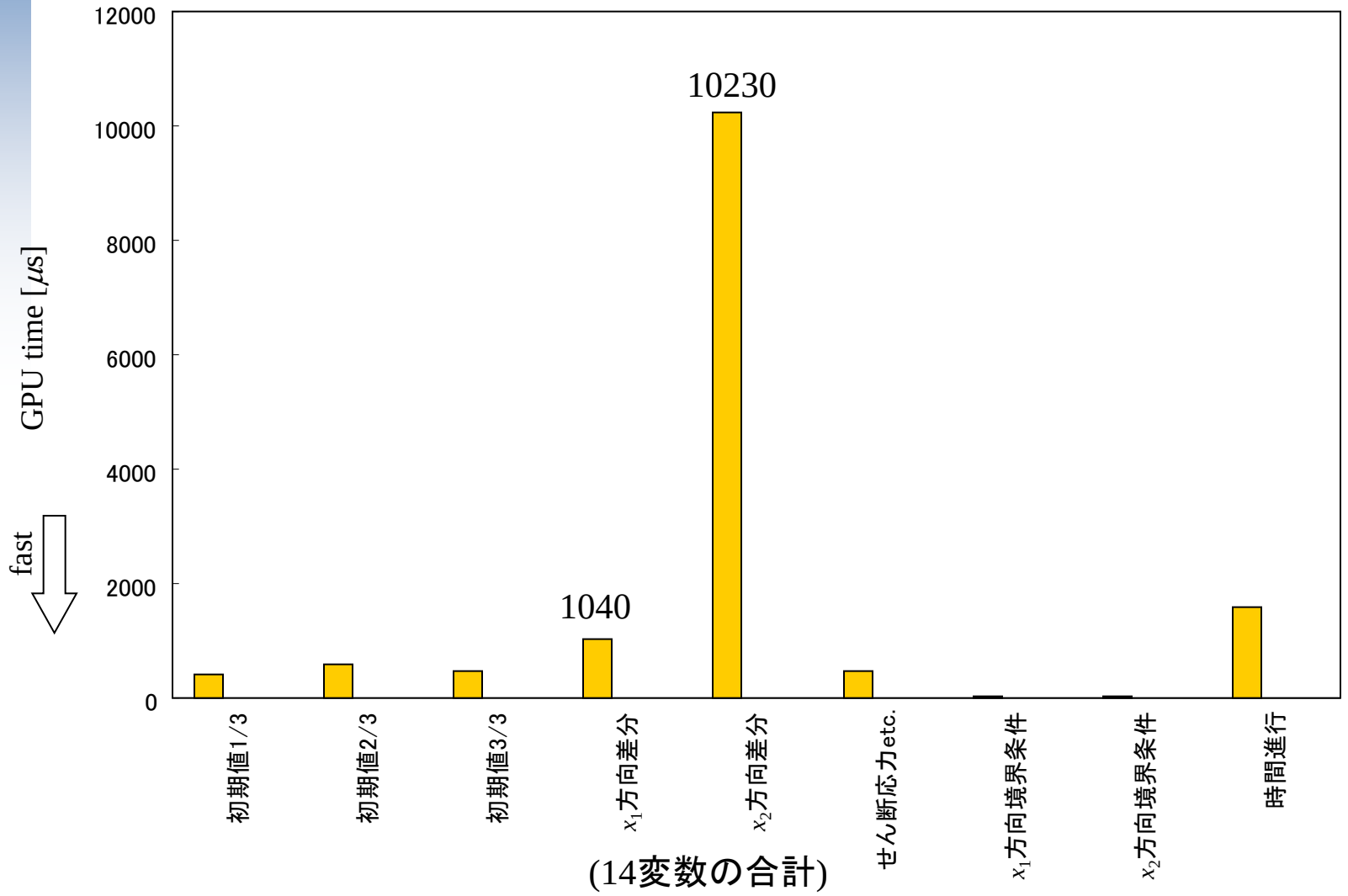


- 1 Threadが流入・流出の2点を計算
- Blockを2にしてBlock IDに応じて別の評価式を使用

Thread数 (256, 1)

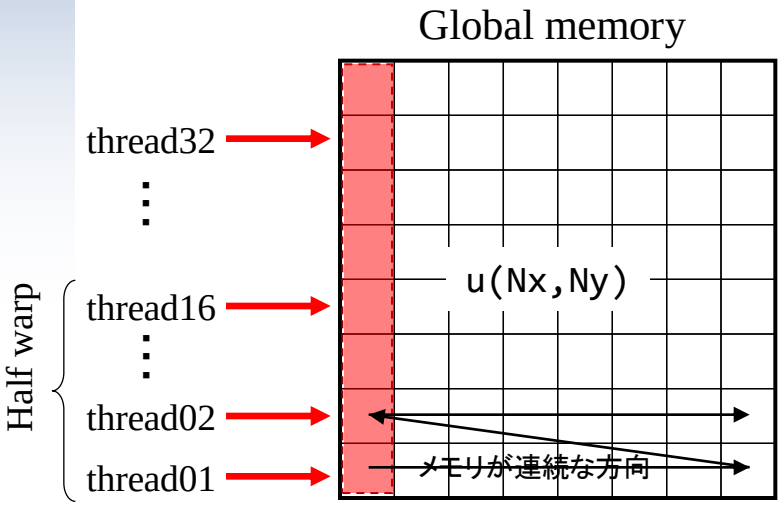
Block数 (2, 2)

各kernelの計算時間(Case1)



x_2方向差分計算の修正(Case2)

- x_1方向と比較して，x_2方向の差分が遅い
  - 実行時間で10倍程度

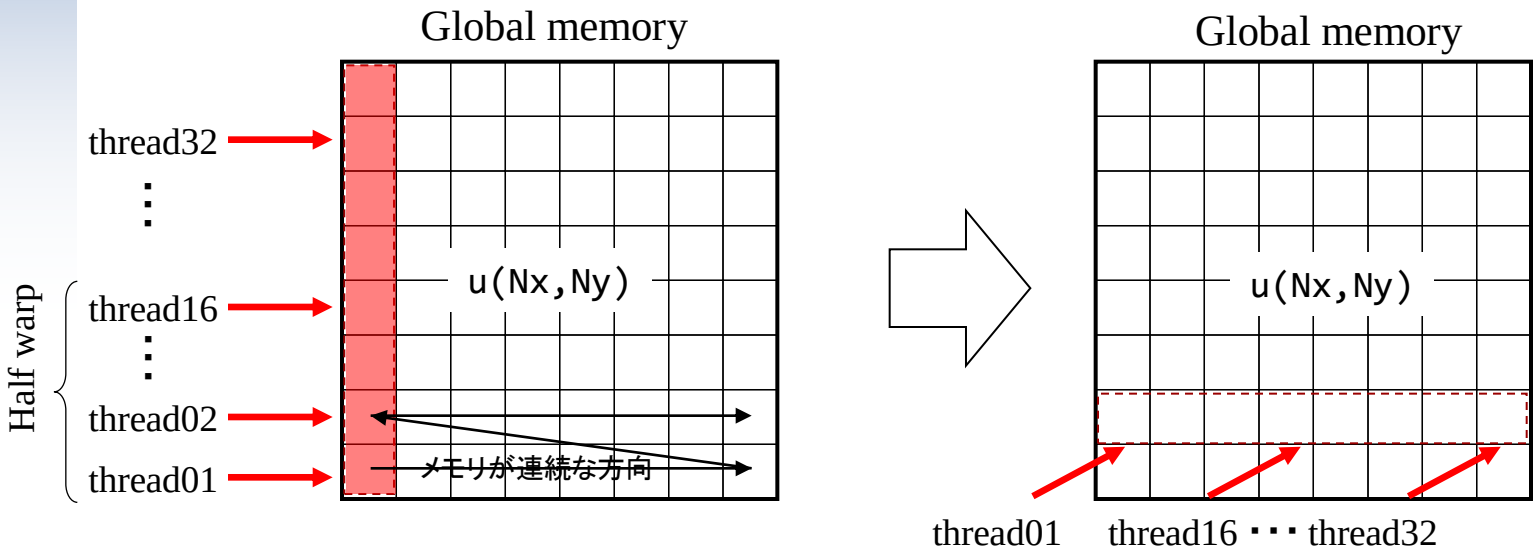


Half warpの各threadが，同一配列に対して
一定アドレスおきに不連続メモリアクセス

Non-Coalesced Access

x_2方向差分計算の修正(Case2)

- x_1方向と比較して，x_2方向の差分が遅い
  - 実行時間で10倍程度

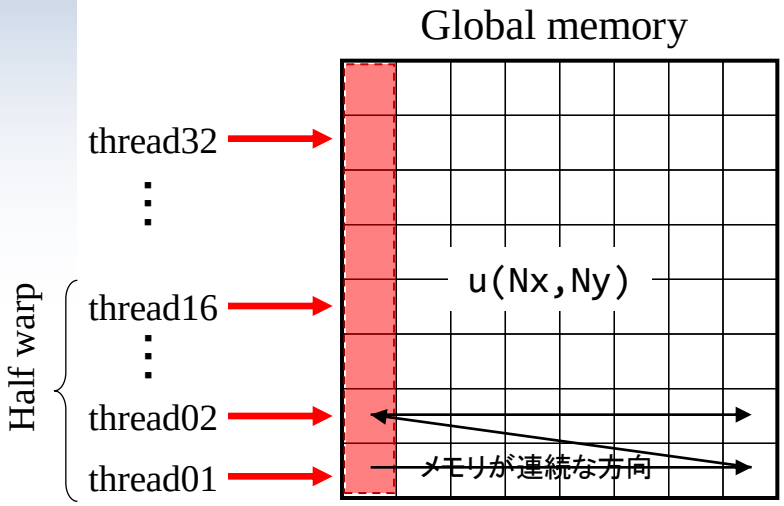


Half warpの各threadが，同一配列に対して
一定アドレスおきに不連続メモリアクセス

Non-Coalesced Access

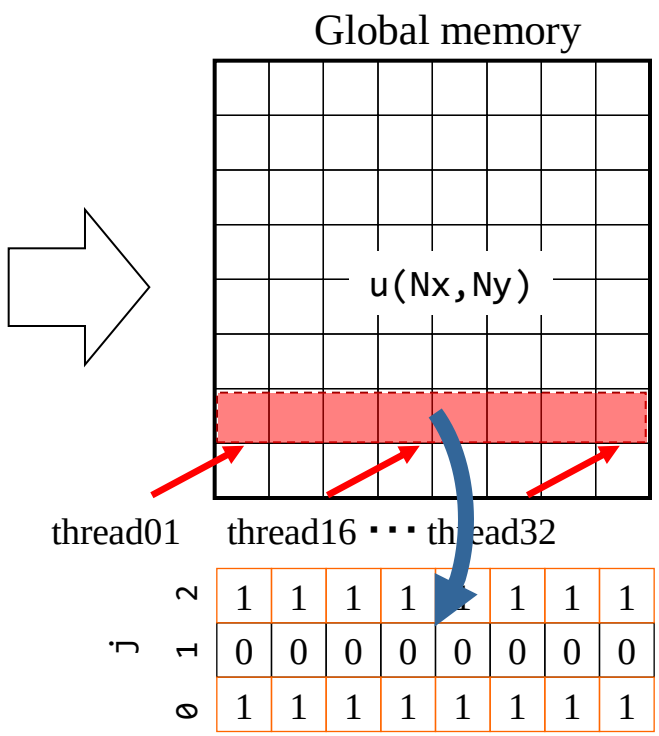
x_2方向差分計算の修正(Case2)

- x_1方向と比較して，x_2方向の差分が遅い
  - 実行時間で10倍程度



Half warpの各threadが，同一配列に対して
一定アドレスおきに不連続メモリアクセス

Non-Coalesced Access

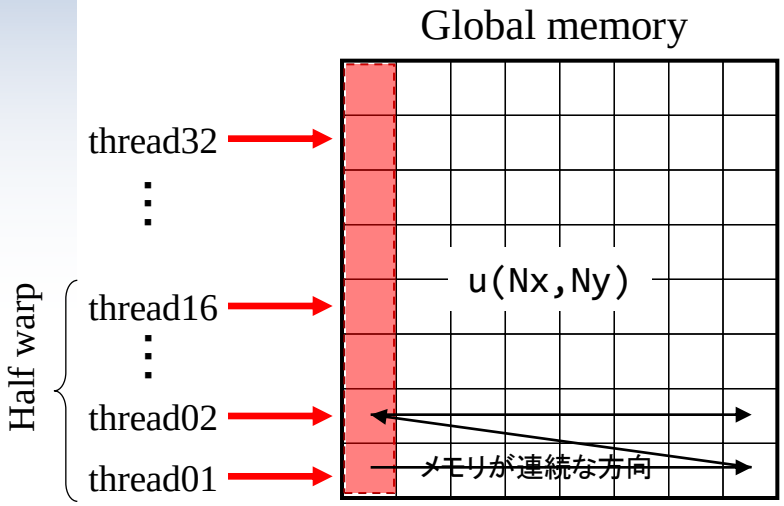


x_1方向に細長くBlockを設定

Half warpの各threadが，同一配列に対して
連続したメモリアドレスでアクセス

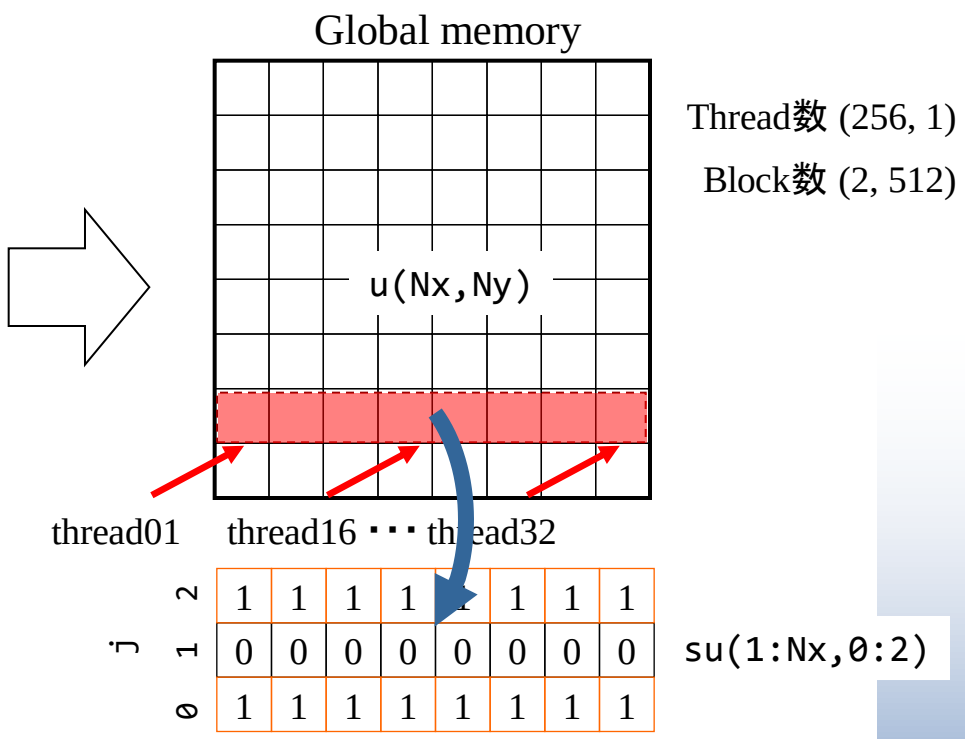
x_2方向差分計算の修正(Case2)

- x_1方向と比較して，x_2方向の差分が遅い
  - 実行時間で10倍程度



Half warpの各threadが，同一配列に対して
一定アドレスおきに不連続メモリアクセス

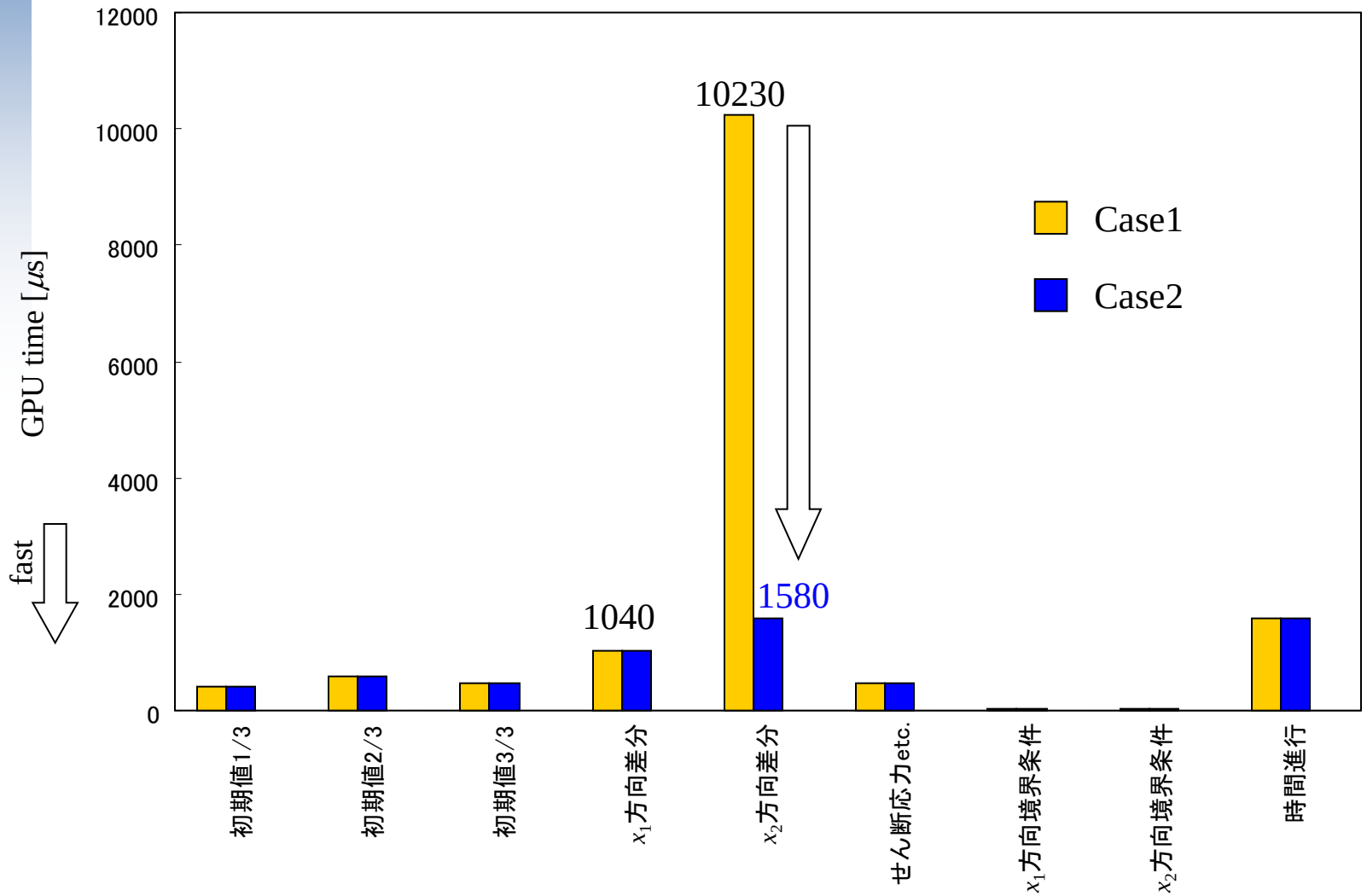
Non-Coalesced Access



x_1方向に細長くBlockを設定

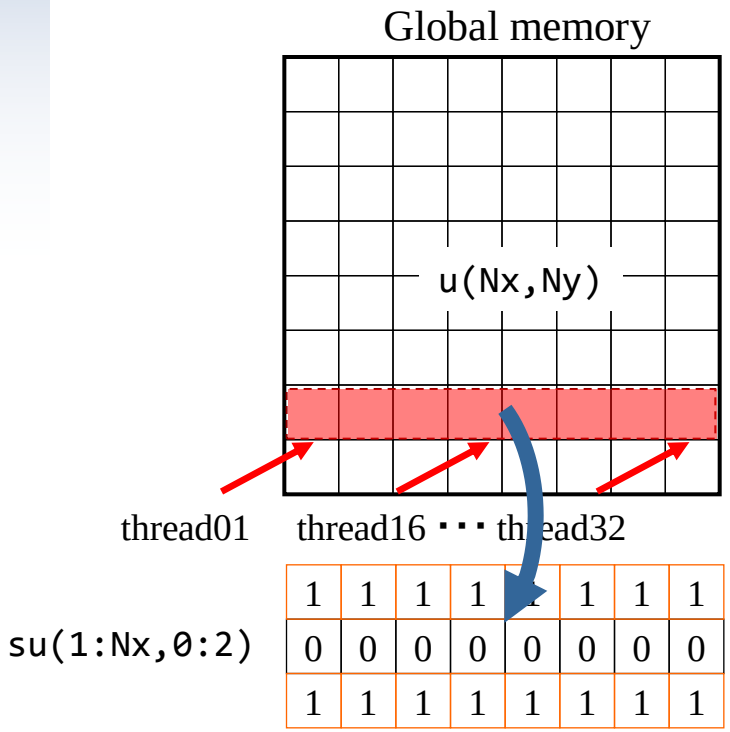
Half warpの各threadが，同一配列に対して
連続したメモリアドレスでアクセス

各kernelの計算時間(Case2)



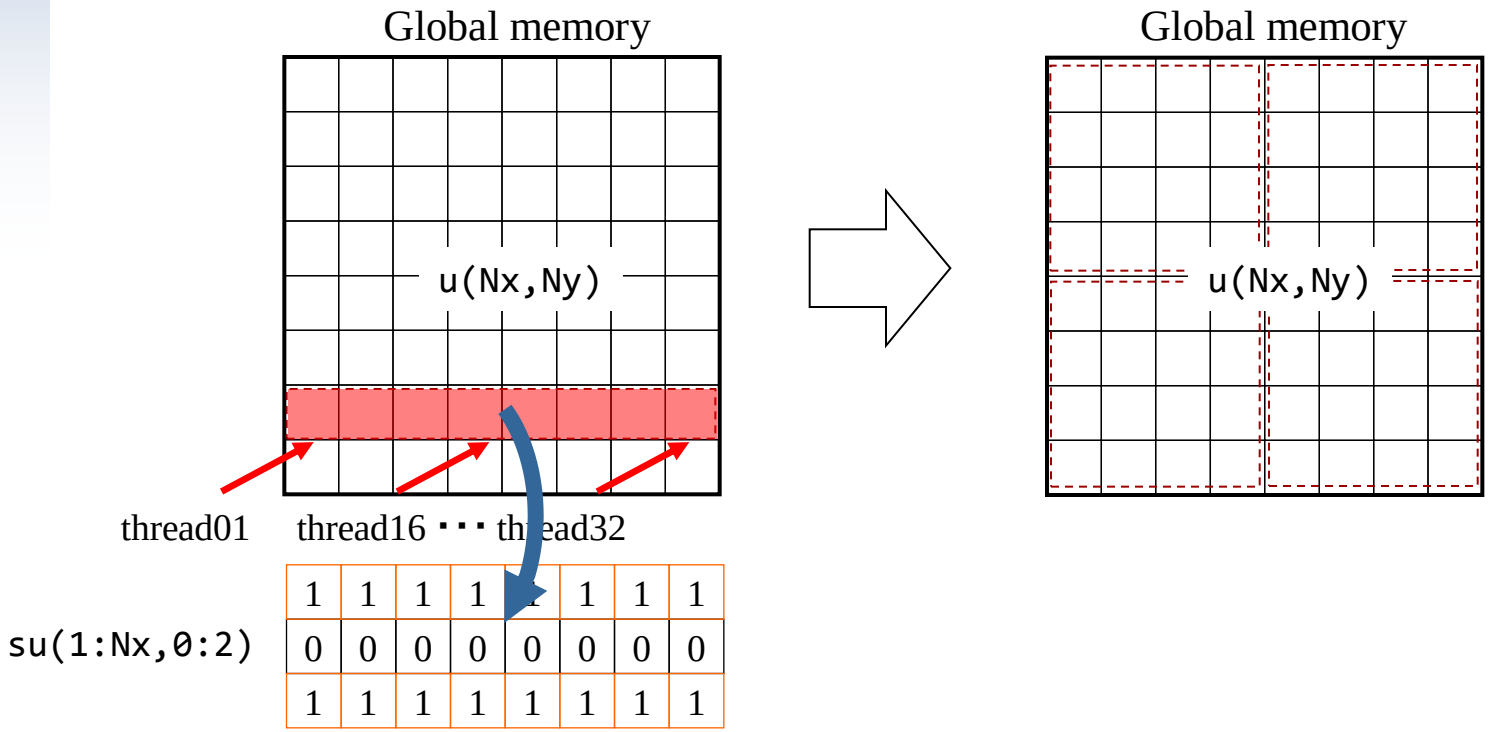
BlockとThread数の調整(Case3)

- Case2でのSharedメモリの使用方法は非効率的
- Blockを2次元的に設定



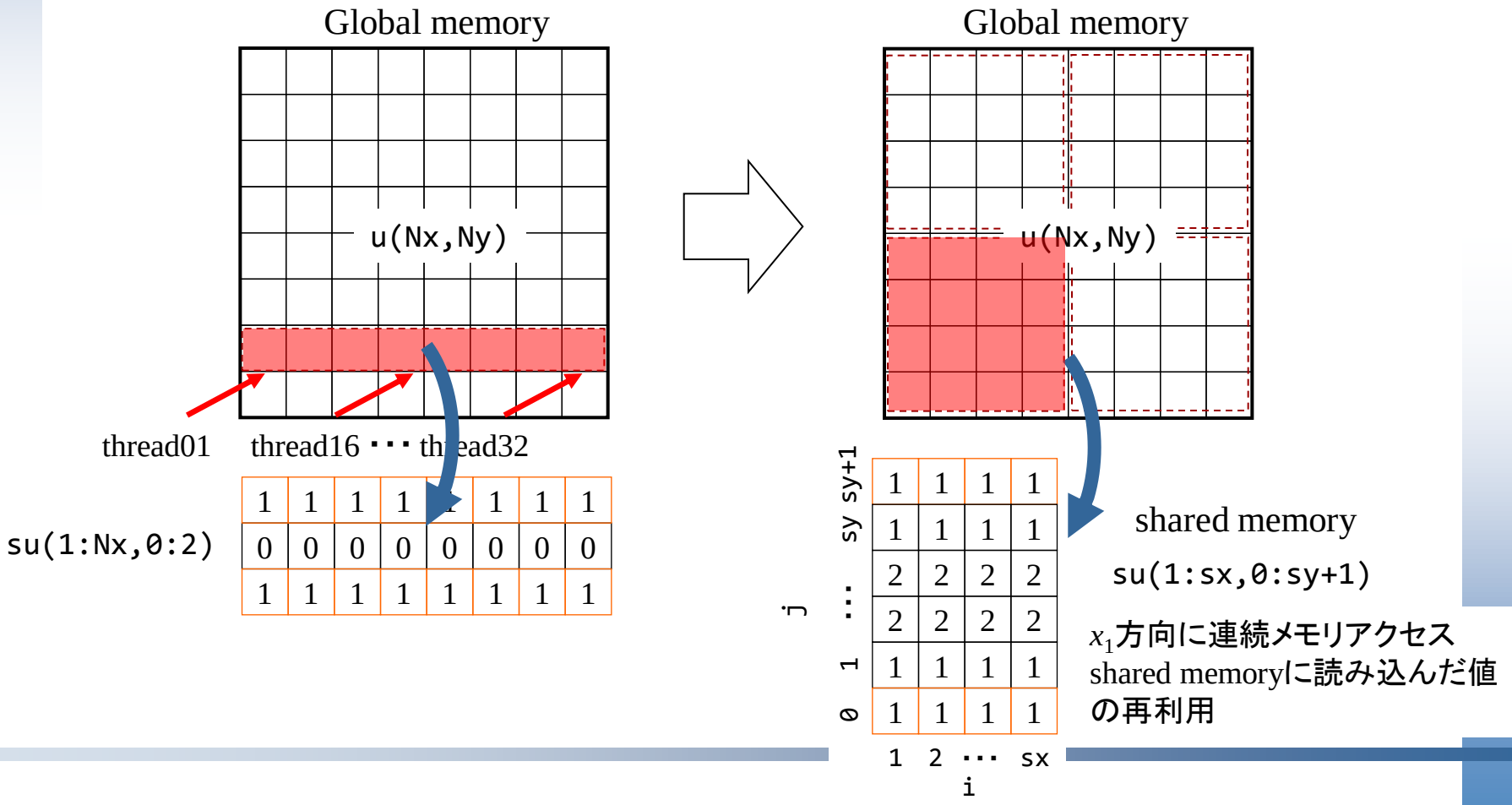
BlockとThread数の調整(Case3)

- Case2でのSharedメモリの使用方法は非効率的
- Blockを2次元的に設定



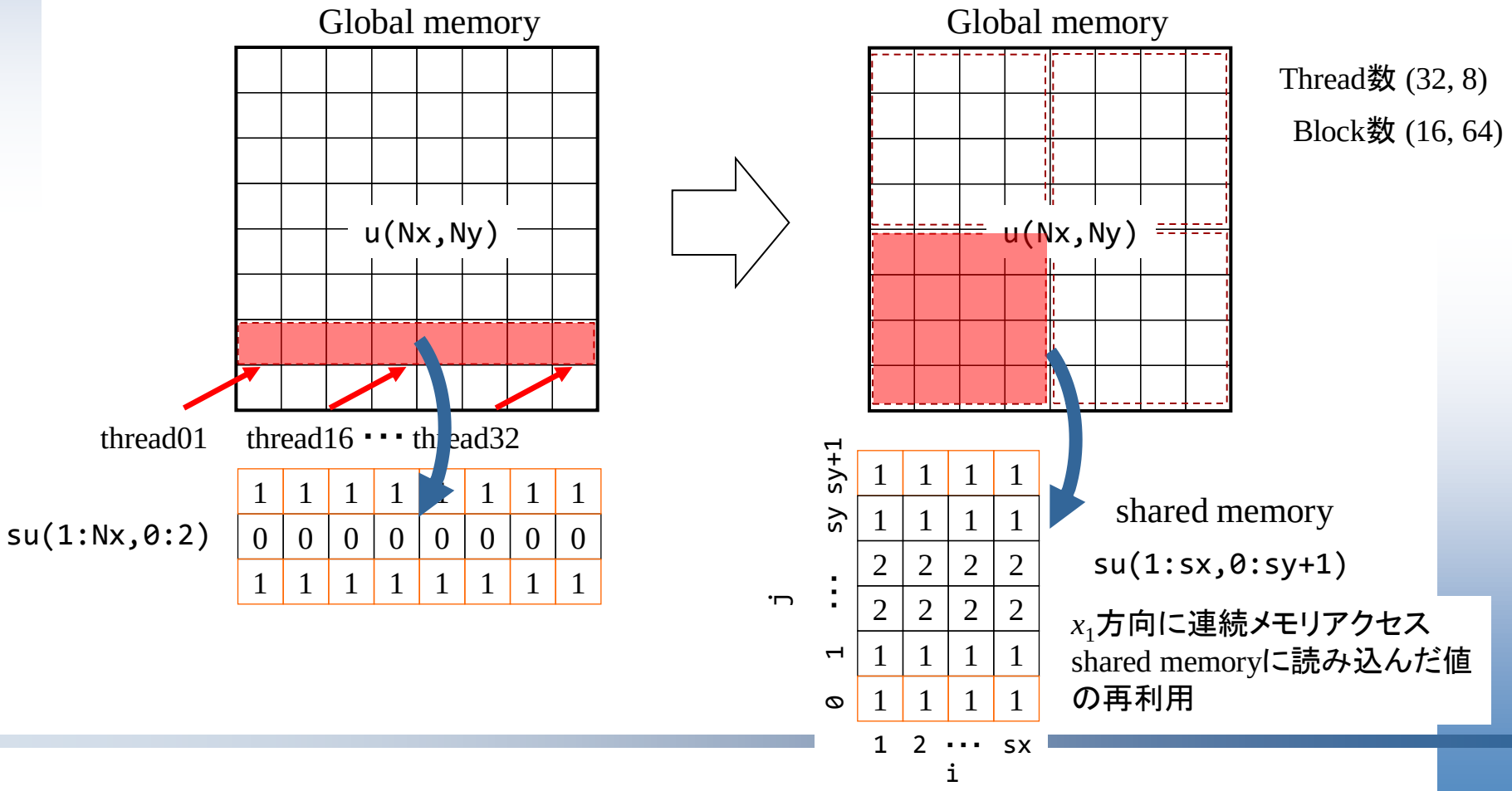
BlockとThread数の調整(Case3)

- Case2でのSharedメモリの使用方法は非効率的
- Blockを2次元的に設定

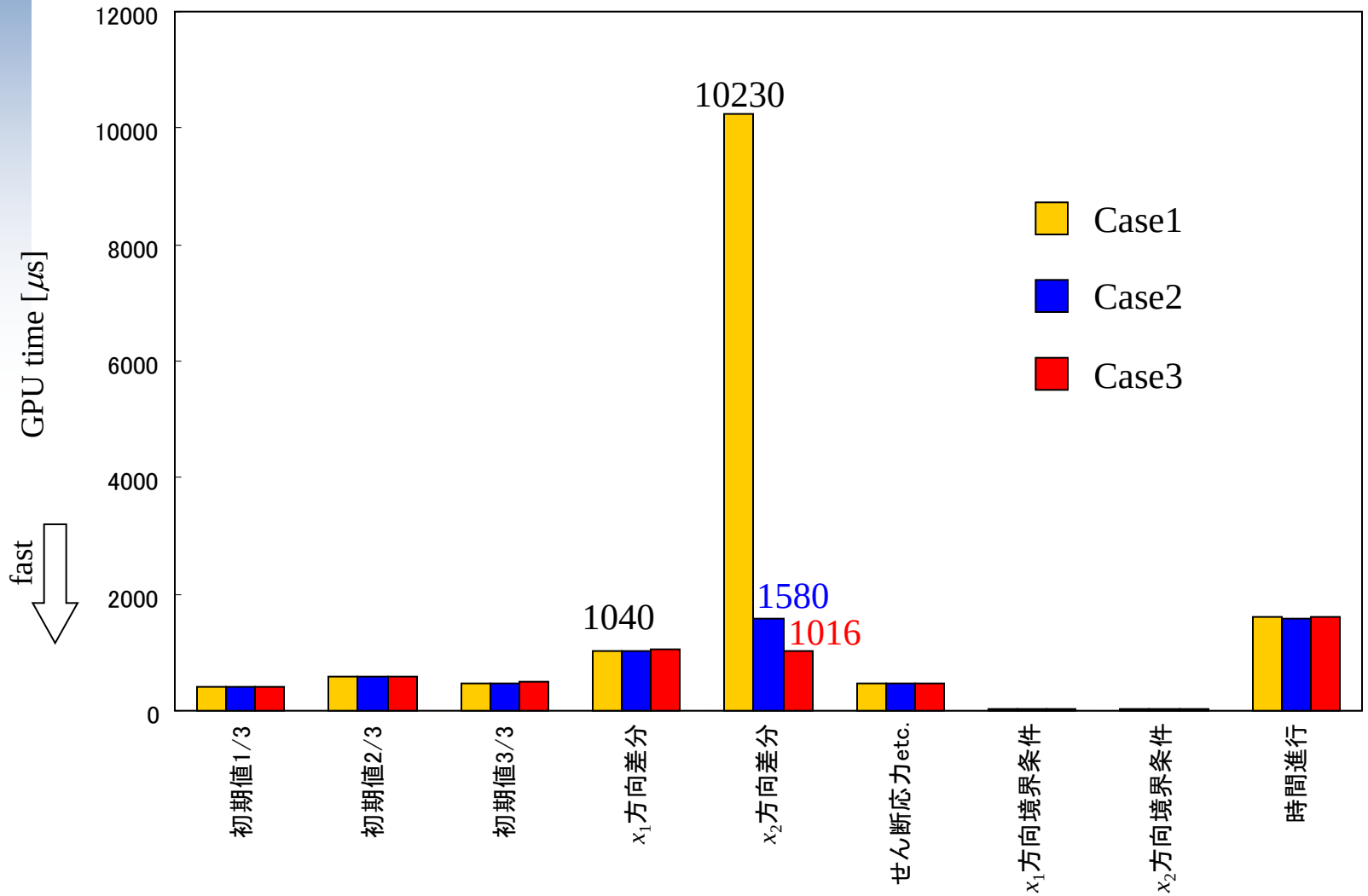


BlockとThread数の調整(Case3)

- Case2でのSharedメモリの使用方法は非効率的
- Blockを2次元的に設定



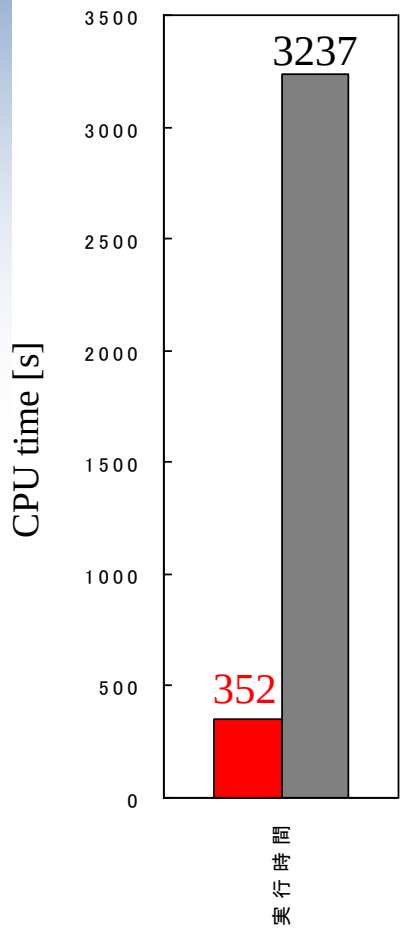
各kernelの計算時間(Case3)

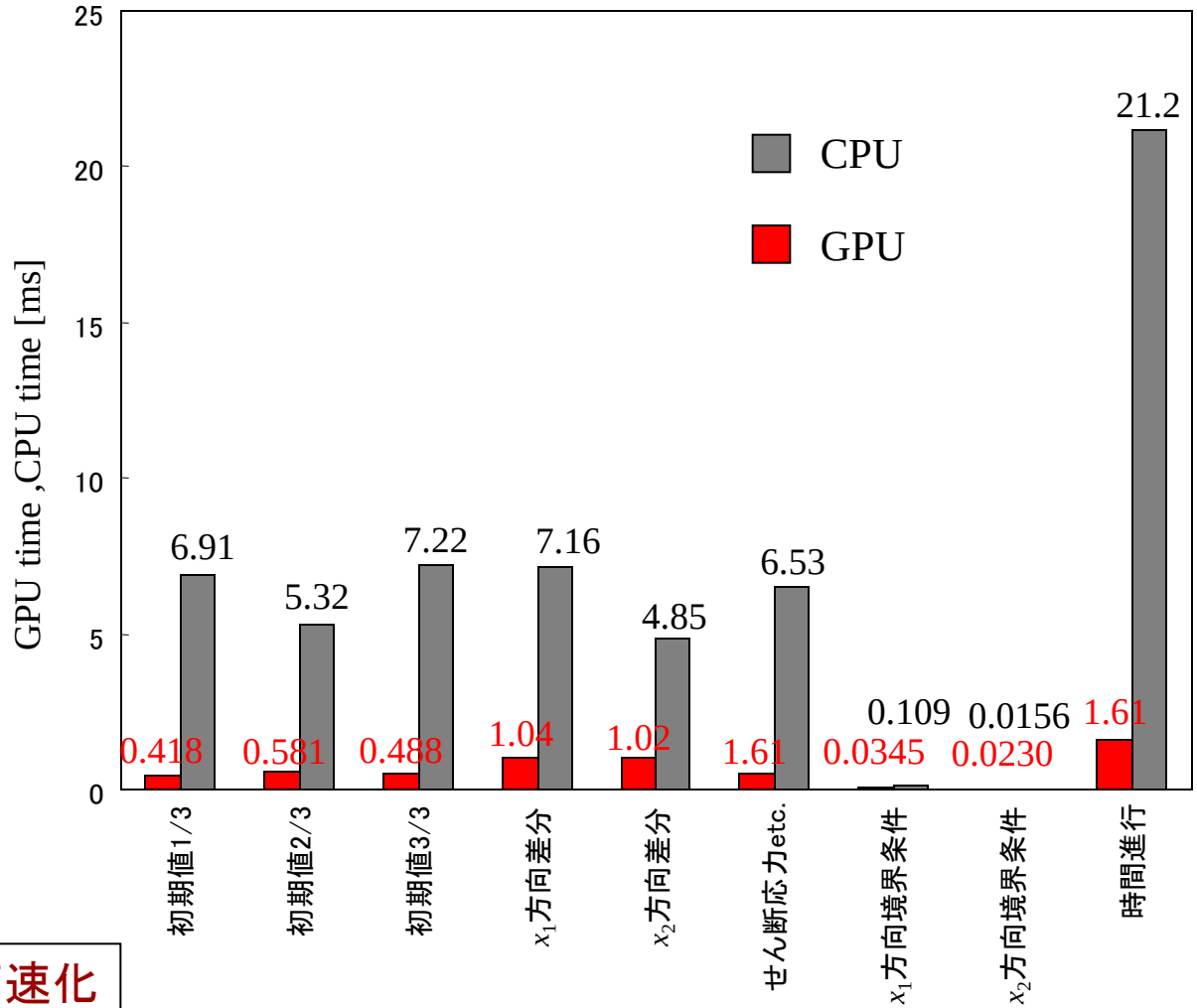


CPUでの実行時間との比較

- GPU用コードとほぼ同じCPUコードを作成
  - Fortran→CUDA Fortran→Fortran
    - CPUの処理に適したように適宜書き換え
  - CPU：Core i7 920
  - OS：Windows HPC Server 2008 (64bit)
  - コンパイラ：Intel Fortran 11.1
  - コンパイルオプション： /O3 /QaxSSE4.2 /QxSSE4.2
    （高度の最適化，Core2シリーズ専用コード生成）

実行時間の比較





実行時間基準で約10倍高速化

CUDA Cとの比較

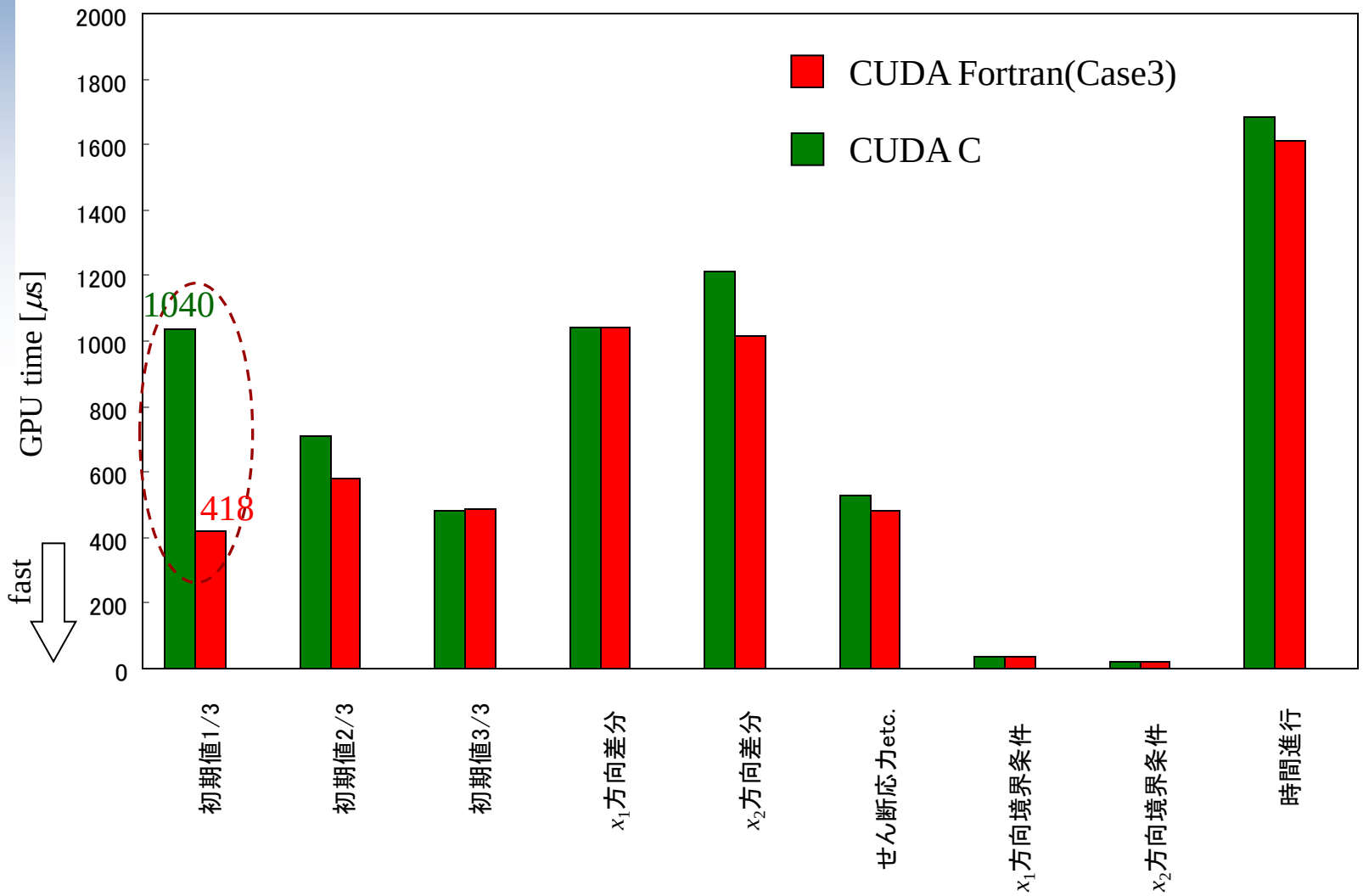
- CUDA FortranからCUDA Cへ移植

  CUDA FortranからCUDA Cへの移植に需要はないと思いますが・・・

  - 同じプログラムを実行することで実行時間を比較

  - 変数の宣言・確保，kernelのインタフェース以外は移植性高い
    - 個人的には，Fortran→C→CUDA C と移植するより，
      Fortran→CUDA Fortran→CUDA C とする方が容易

各kernelの実行時間の比較



初期値計算

$$u = -\frac{\Gamma}{2\pi}\frac{y}{r_c^2}\exp\left(-\frac{x^2+y^2}{2r_c^2}\right)+u_\infty$$

$$v = \frac{\Gamma}{2\pi}\frac{x}{r_c^2}\exp\left(-\frac{x^2+y^2}{2r_c^2}\right)$$

$$p = -\rho_\infty\frac{\Gamma^2}{2(2\pi r_c)^2}\exp\left(-\frac{x^2+y^2}{r_c^2}\right)+p_\infty$$

$$\rho = \rho_\infty$$

初期値1/3
文献に記された初期の速度・圧力・密度場の設定

式中のパラメータの参照

CUDA Fortran：外部モジュールで宣言したパラメータ

CUDA C：　　constant memory

CPU側変数はconst修飾してもGPU側から参照不可能（int型は参照可能）

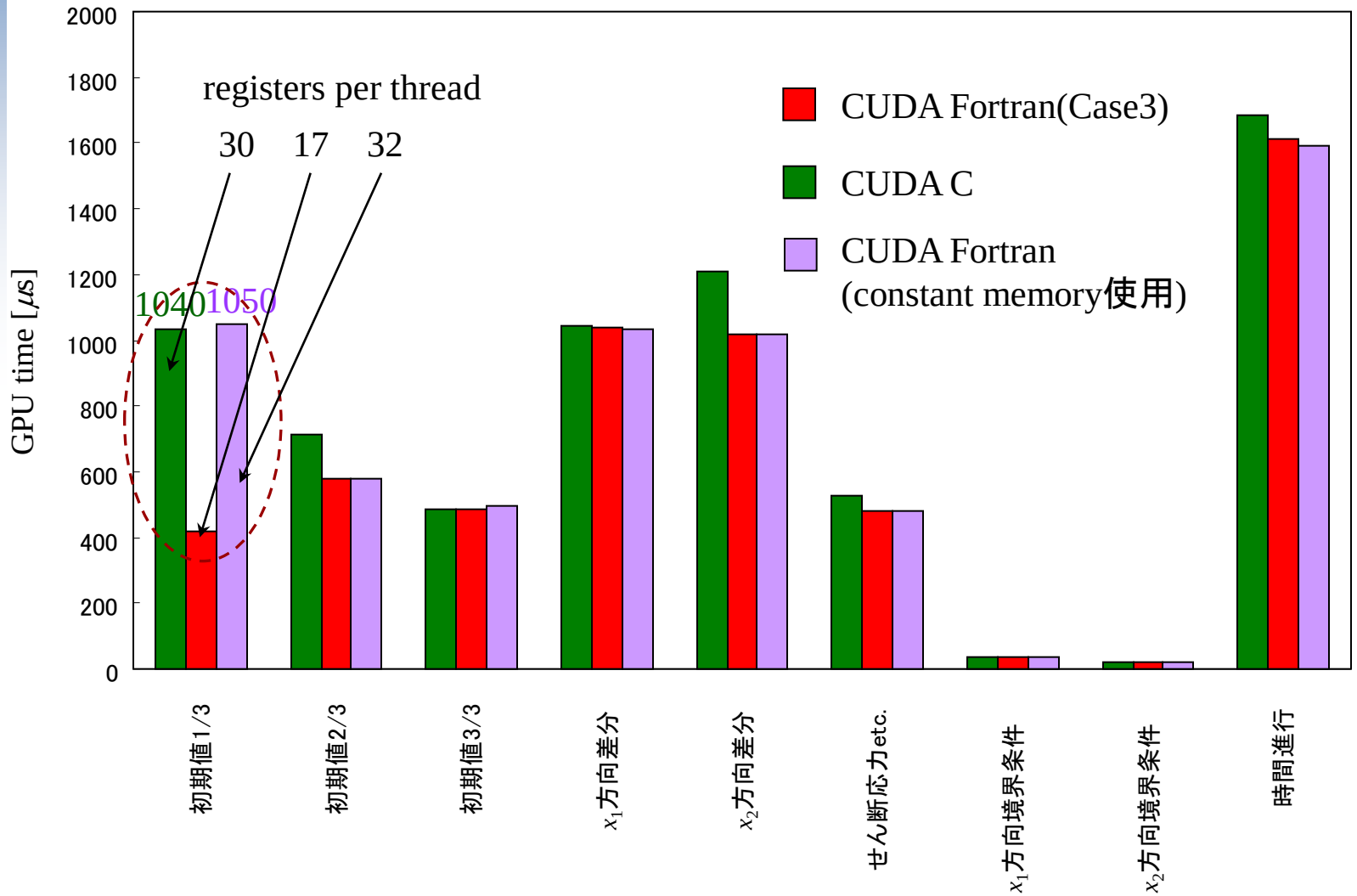
```
const int N = 512;
const float v = 1.0;
__constant__ float const_v = 1.0; //constant memory

__global__ void init_GPU(float *a){

    int i = blockDim.x*blockIdx.x + threadIdx.x;
        a[i] = v;         //error
        a[i] = (float)N; //OK
        a[i] = const_v;  //OK

}
```

モジュール内外で宣言したパラメータをGPU側から参照可能

```
module kernel
    use param,only:z !外部モジュールで宣言したパラメータ
    implicit none

    integer,parameter :: n = 512
    real   ,parameter :: v = 1.0
    contains
    attributes(global) subroutine init_GPU(a)
        implicit none

        real    :: a(n) !OK
        integer :: i

        i = (blockIdx%x-1)*blockDim%x + threadIdx%x
            a(i) = v !OK
            a(i) = z !OK
```

CUDA Fortranでconstant memoryを使用すると・・・

constant memory使用時の実行時間



計算の効率化

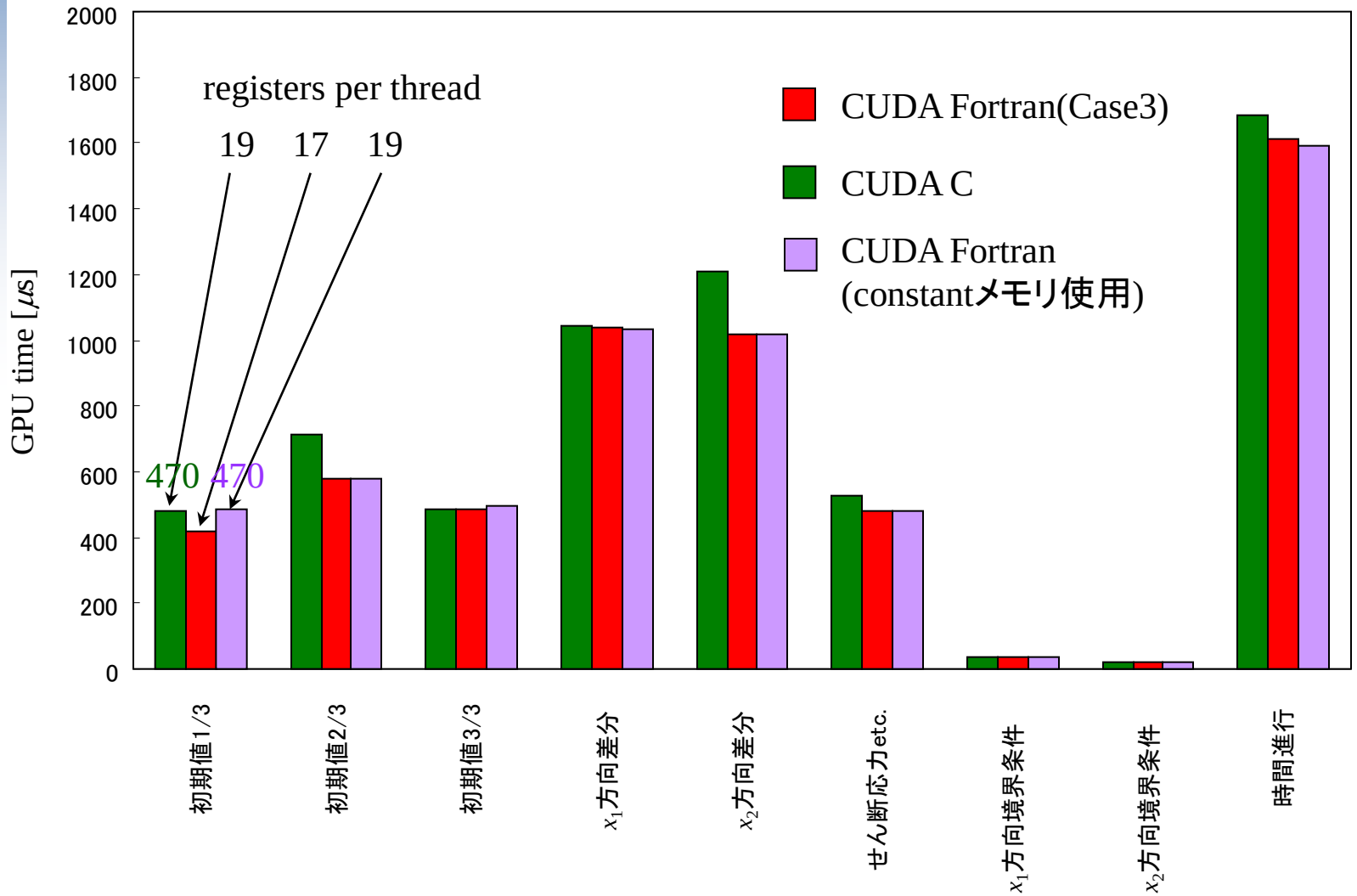
- 式中の既知パラメータを用いた箇所をあらかじめ計算
  - 演算結果をconstant memoryへ格納
  - 除算を乗算へ置き換え

$$
\left.
\begin{aligned}
u &= -\Gamma \frac{y}{r_c^{\,2}} \exp\left(-\frac{x^2+y^2}{2r_c^{\,2}}\right) + u_\infty \\
v &= \Gamma \frac{x}{r_c^{\,2}} \exp\left(-\frac{x^2+y^2}{2r_c^{\,2}}\right) \\
p &= -\rho_\infty \frac{\Gamma^2}{2r_c^{\,2}} \exp\left(-\frac{x^2+y^2}{r_c^{\,2}}\right) + p_\infty \\
\rho &= \rho_\infty
\end{aligned}
\right\}
\Rightarrow
\left\{
\begin{aligned}
u &= -\Gamma y A \exp\left[-\left(x^2+y^2\right)B\right] + u_\infty \\
v &= \Gamma x A \exp\left[-\left(x^2+y^2\right)B\right] \\
p &= -\rho_\infty \Gamma^2 B \exp\left[-\left(x^2+y^2\right)A\right] + p_\infty \\
\rho &= \rho_\infty
\end{aligned}
\right.
$$

$$A = \frac{1}{r_c^{\,2}}$$

$$B = \frac{1}{2r_c^{\,2}}$$

各kernelの実行時間の比較



CUDA Cとの比較

- CUDA Cとの違い
  - 同じようにプログラムを作成すれば，性能に違いはない
  - 使用レジスタ数に若干の相違
- CUDA Fortranにおけるconstant memoryの利用
  - CUDA Cでは定数参照として利用される（こともある）
  - CUDA FortranではCPUで宣言したparameterをGPUから参照可能
    →定数参照にconstant memoryを利用することは意味がない
    - CPUで宣言したparameterをGPUから参照した場合，コンパイル時に最適化されている
  - constant memoryの使用が悪影響を及ぼす訳ではない
    - 手動で最適化すれば性能はきちんと出る
  - 外部モジュールで宣言したconstant memoryは使用しない方がよい（数値を参照できないことがある）

ここまでのまとめ

- 性能はCUDA Cと同じ
- エラー処理を考えなければ，CUDA Cよりも簡潔な記述が可能
- CPUで宣言したparameterをGPUから参照可能
  - parameterを使用すると，コンパイル時に最適化されている
- 配列添字を任意の範囲で宣言できるので，袖領域など仮想的な領域を考える際に便利

GPUを用いた流体の差分法計算

- 非圧縮性流体
  - 流体の圧縮性が無視できる場合
  - 工業上多くの場面で観察される重要な流れ
  - 非圧縮条件を満たすために連立方程式を解く
    - 連立方程式がボトルネック
  - 近年，多くの適用例が報告

支配方程式

$$\nabla \cdot \boldsymbol{u} = 0$$ （連続の式）

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\frac{1}{\rho}\nabla p \frac{\partial p}{\partial x_i} + \nu \nabla^2 \boldsymbol{u}$$ （Navier-Stokes方程式）

$\nabla \times$

$$\frac{\partial \omega}{\partial t} + (\boldsymbol{u} \cdot \nabla)\omega + \omega \nabla \cdot \boldsymbol{u} = \nu \nabla^2 \omega$$ （渦度方程式）

$$\boldsymbol{u} = \nabla \phi + \nabla \times \boldsymbol{\psi} \quad \begin{cases} \nabla^2 \phi = 0 \\ \nabla^2 \boldsymbol{\psi} = -\boldsymbol{\omega} \end{cases}$$ （速度場）

p :圧力　　u :速度 $= \nabla \phi + \nabla \times \boldsymbol{\psi}$　　ω :渦度 $= \nabla \times \boldsymbol{u}$　　ϕ :スカラポテンシャル

ρ :密度　　τ_{ij} :粘性応力 $= \mu\left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^{\mathrm{T}} - \frac{2}{3}\boldsymbol{I}\nabla \cdot \boldsymbol{u} \right)$　　ψ :ベクトルポテンシャル

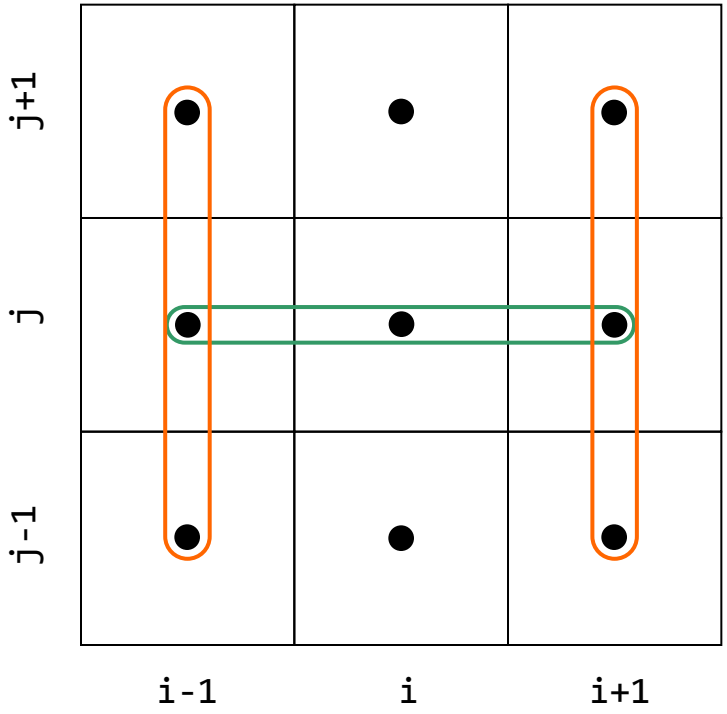
離散化計算

$$\frac{\partial \omega}{\partial t} + \frac{\partial}{\partial x}\left(\frac{\partial \psi}{\partial y}\omega\right) - \frac{\partial}{\partial y}\left(\frac{\partial \psi}{\partial x}\omega\right) = \nu\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right)$$

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = -\omega$$

空間離散化：2次精度中心差分

時間離散化：1次精度Euler法



渦度ωは，(i，j)点から上下方向の4点
流れ関数ψは，(i，j)点から斜め方向の4点

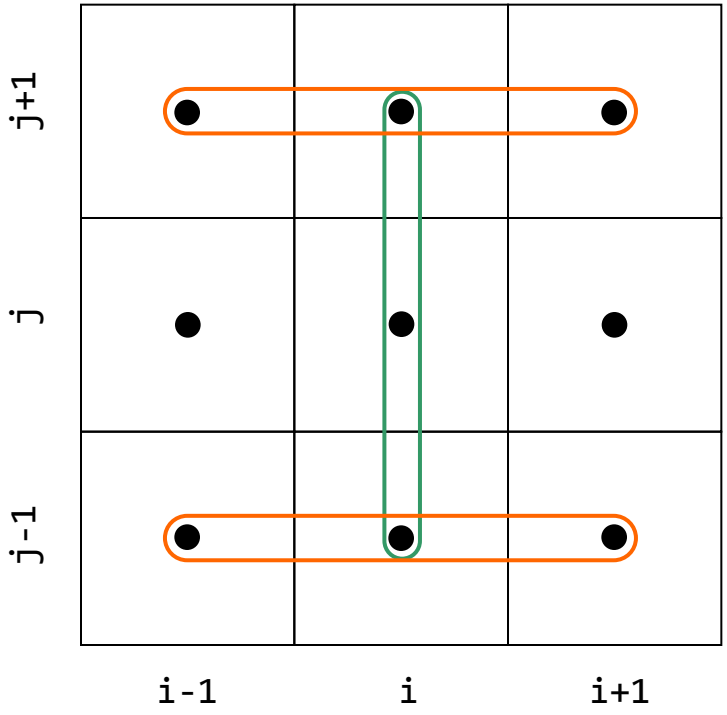
離散化計算

$$\frac{\partial \omega}{\partial t} + \frac{\partial}{\partial x}\left(\frac{\partial \psi}{\partial y}\omega\right) - \frac{\partial}{\partial y}\left(\frac{\partial \psi}{\partial x}\omega\right) = \nu\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right)$$

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = -\omega$$

空間離散化：2次精度中心差分

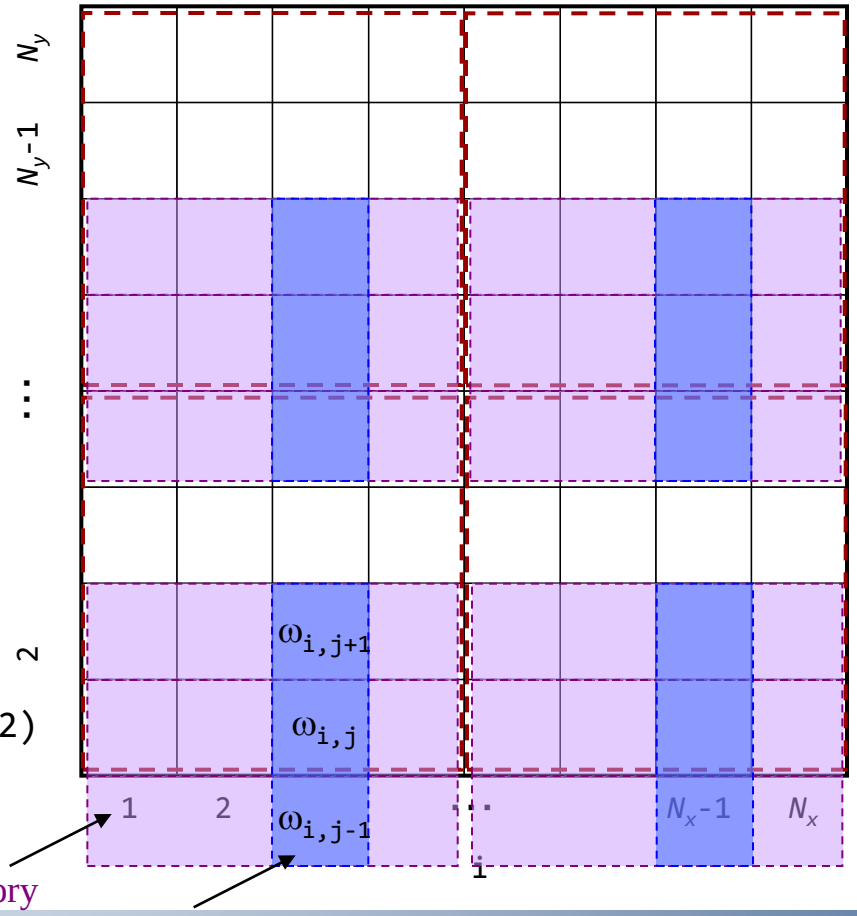
時間離散化：1次精度Euler法



渦度ωは，(i，j)点から上下方向の4点
流れ関数ψは，(i，j)点から斜め方向の4点

渦度方程式の計算

- 空間差分計算，渦度方程式の時間発展
  - 小川・青木(2)の方法を拡張



ψ用shared memoryを3列

ω用shared memoryを1列
registerを3本
確保し，データを極力再利用

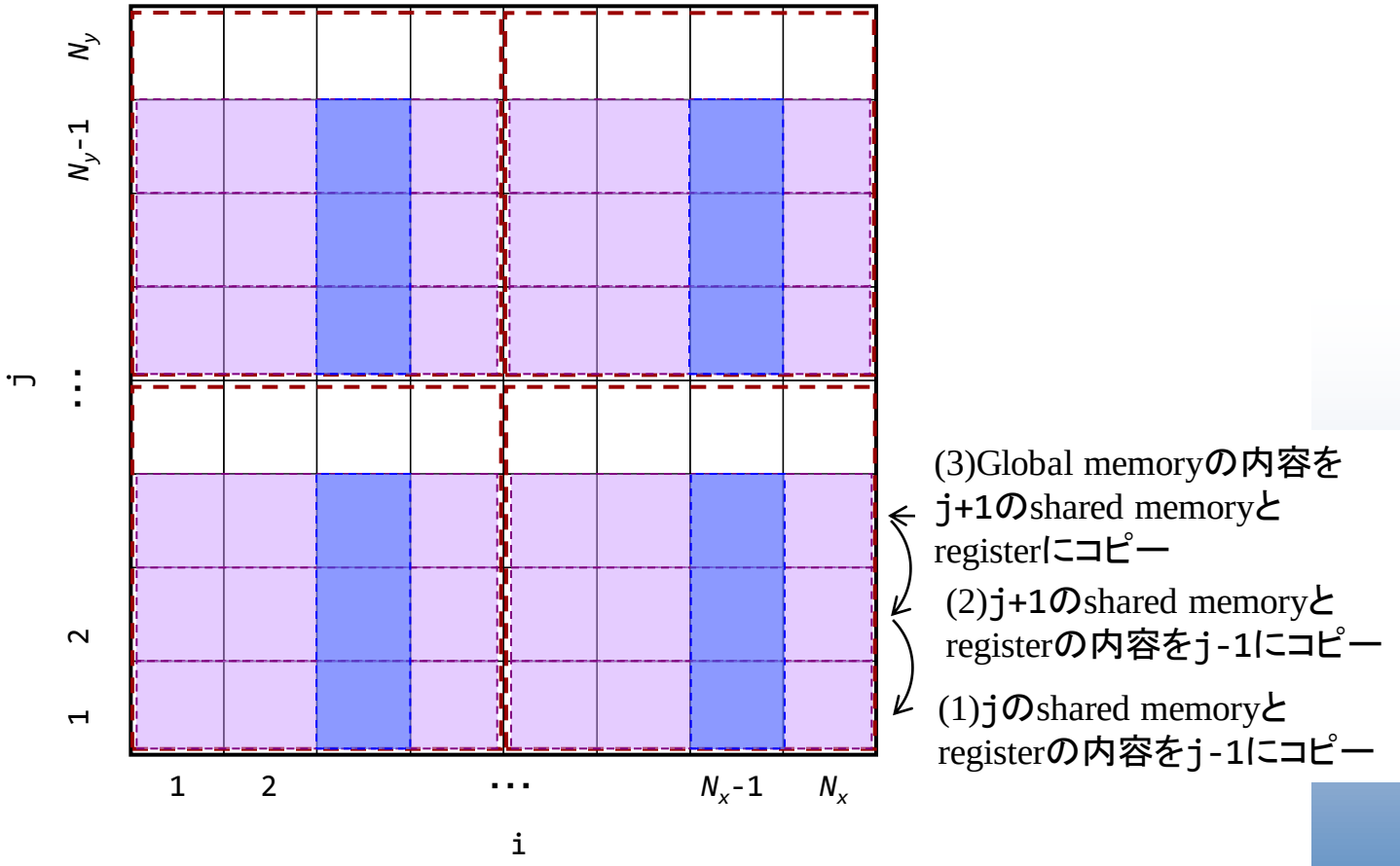
sω(0:sx+1)

sψ(0:sx+1,0:2)

(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

渦度方程式の計算

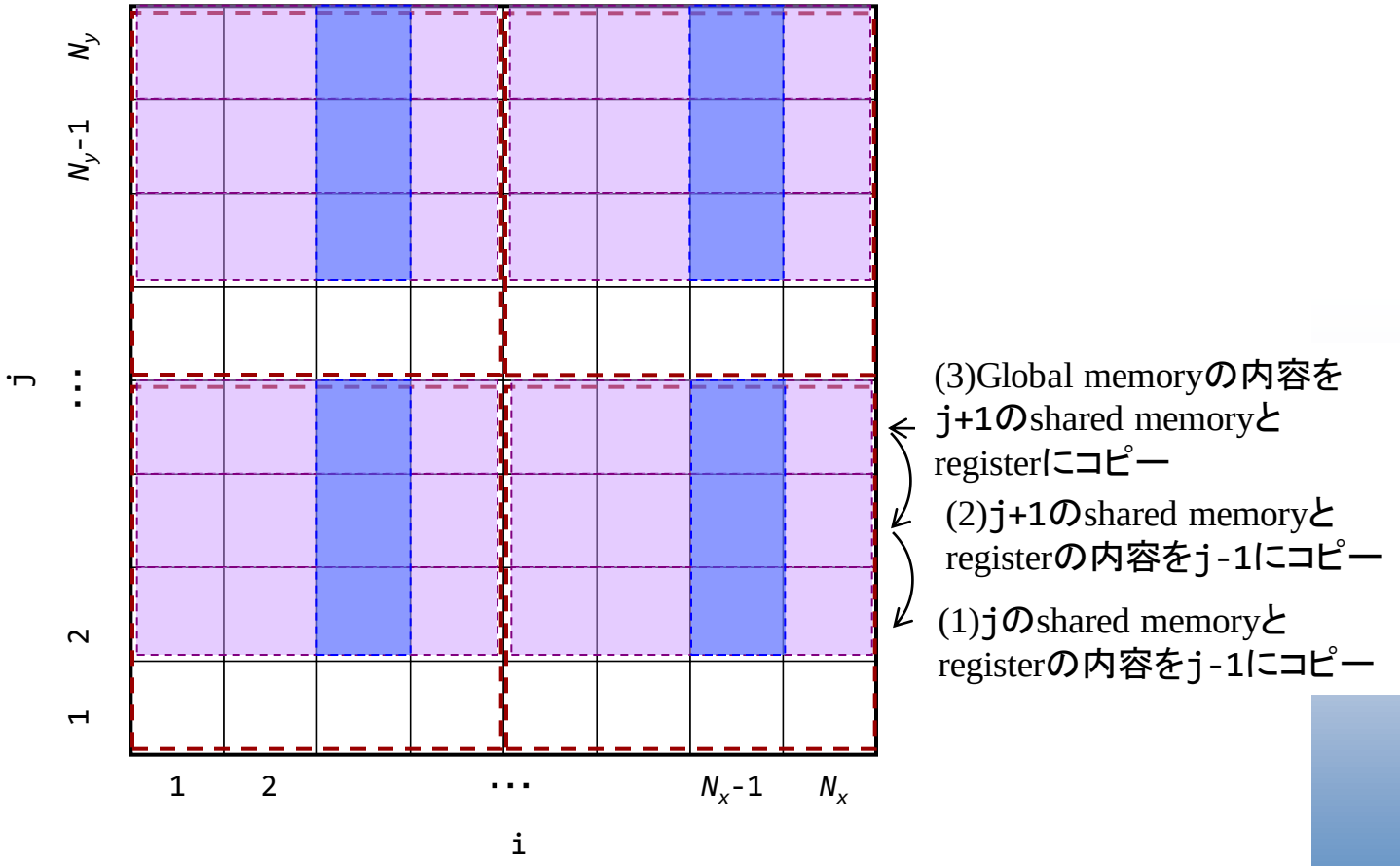
- 空間差分計算，渦度方程式の時間発展
  - 小川・青木(2)の方法を拡張



(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

渦度方程式の計算

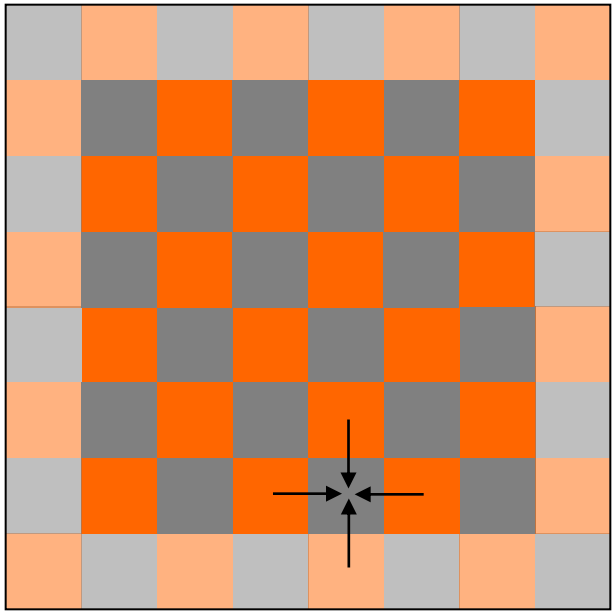
- 空間差分計算，渦度方程式の時間発展
  - 小川・青木(2)の方法を拡張



(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

Poisson方程式の計算

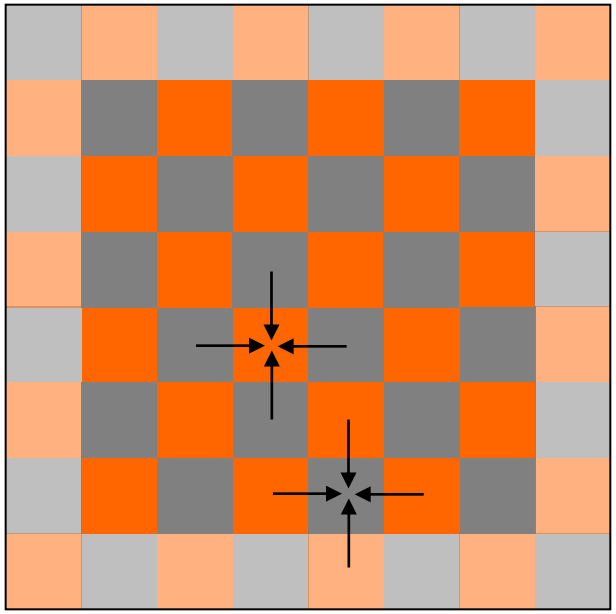
- Red＆Black SOR
  - ストライドアクセスはデータ転送レートの低下を招く
  - Red格子点に対する配列とBlack格子点に対する配列に分割(2)



(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

Poisson方程式の計算

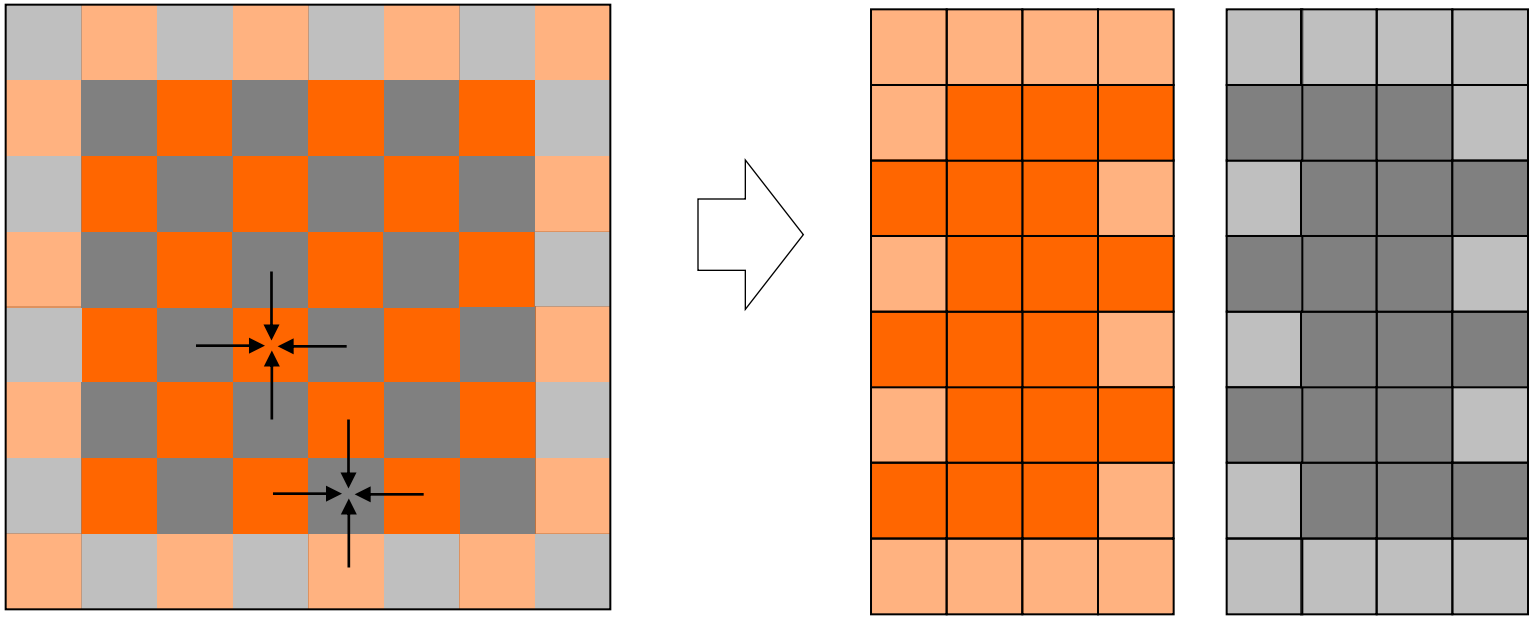
- Red＆Black SOR
  - ストライドアクセスはデータ転送レートの低下を招く
  - Red格子点に対する配列とBlack格子点に対する配列に分割(2)



(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

Poisson方程式の計算

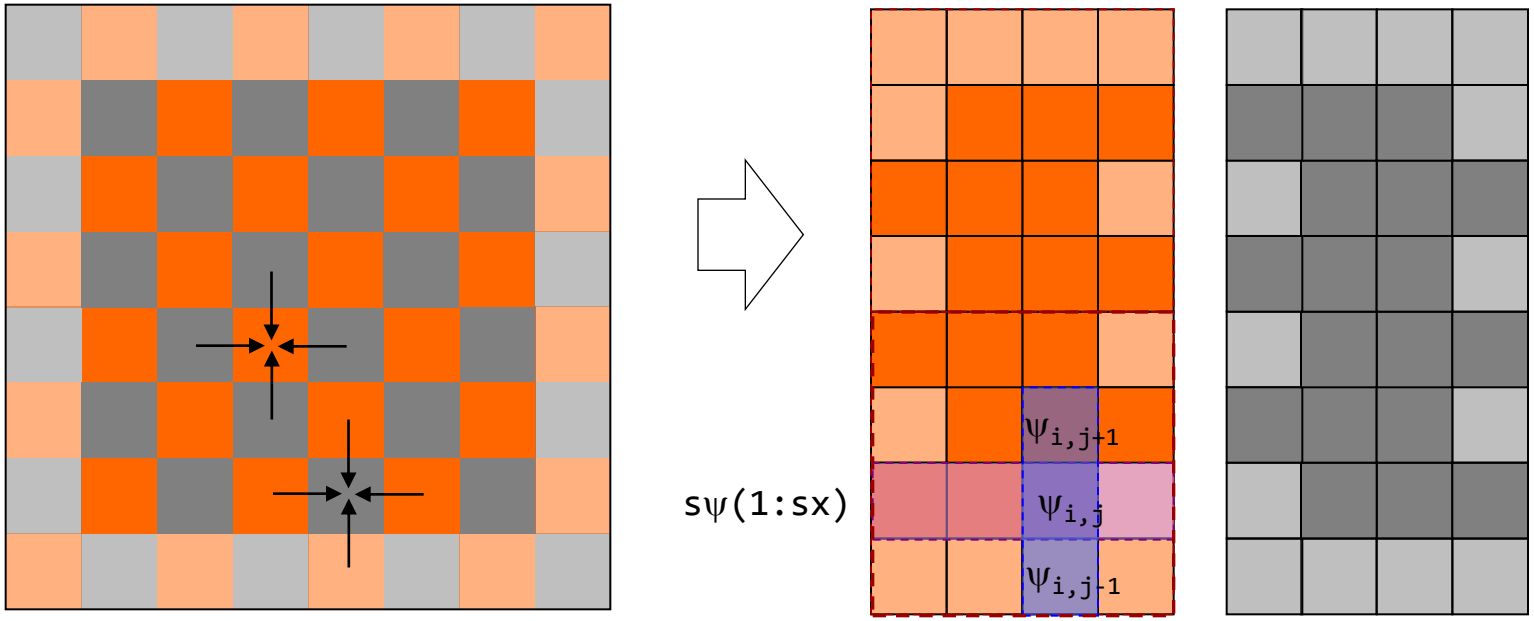
- Red＆Black SOR
  - ストライドアクセスはデータ転送レートの低下を招く
  - Red格子点に対する配列とBlack格子点に対する配列に分割(2)



(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

Poisson方程式の計算

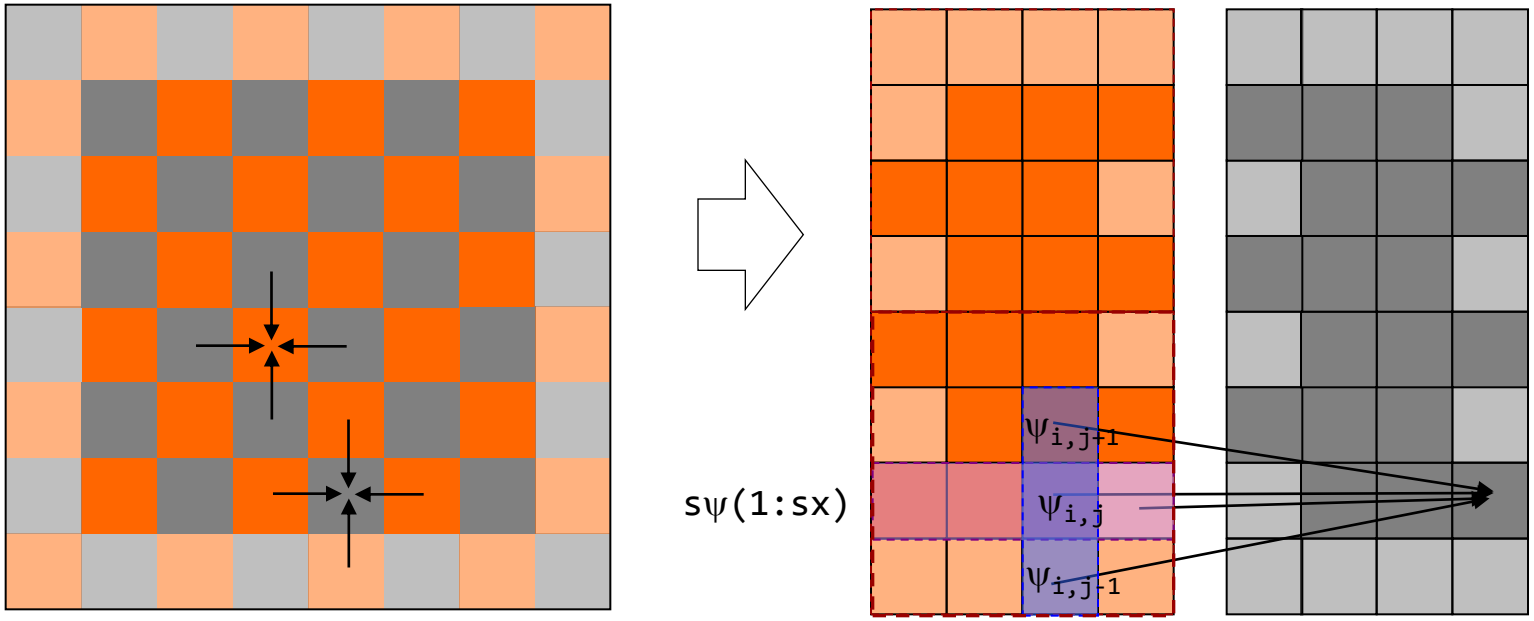
- Red＆Black SOR
  - ストライドアクセスはデータ転送レートの低下を招く
  - Red格子点に対する配列とBlack格子点に対する配列に分割(2)



(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

Poisson方程式の計算

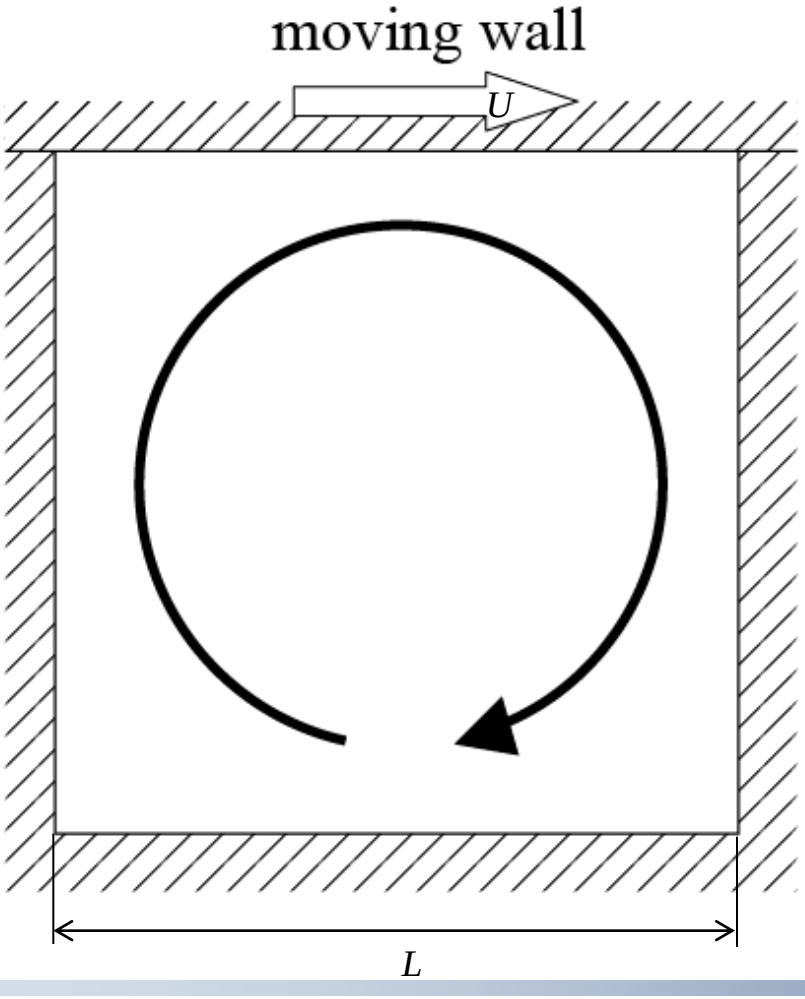
- Red＆Black SOR
  - ストライドアクセスはデータ転送レートの低下を招く
  - Red格子点に対する配列とBlack格子点に対する配列に分割(2)



(2)小川，青木，日本計算工学会論文集, Vol2009,No.20090021, 8pages

計算対象

- 正方キャビティ流れ



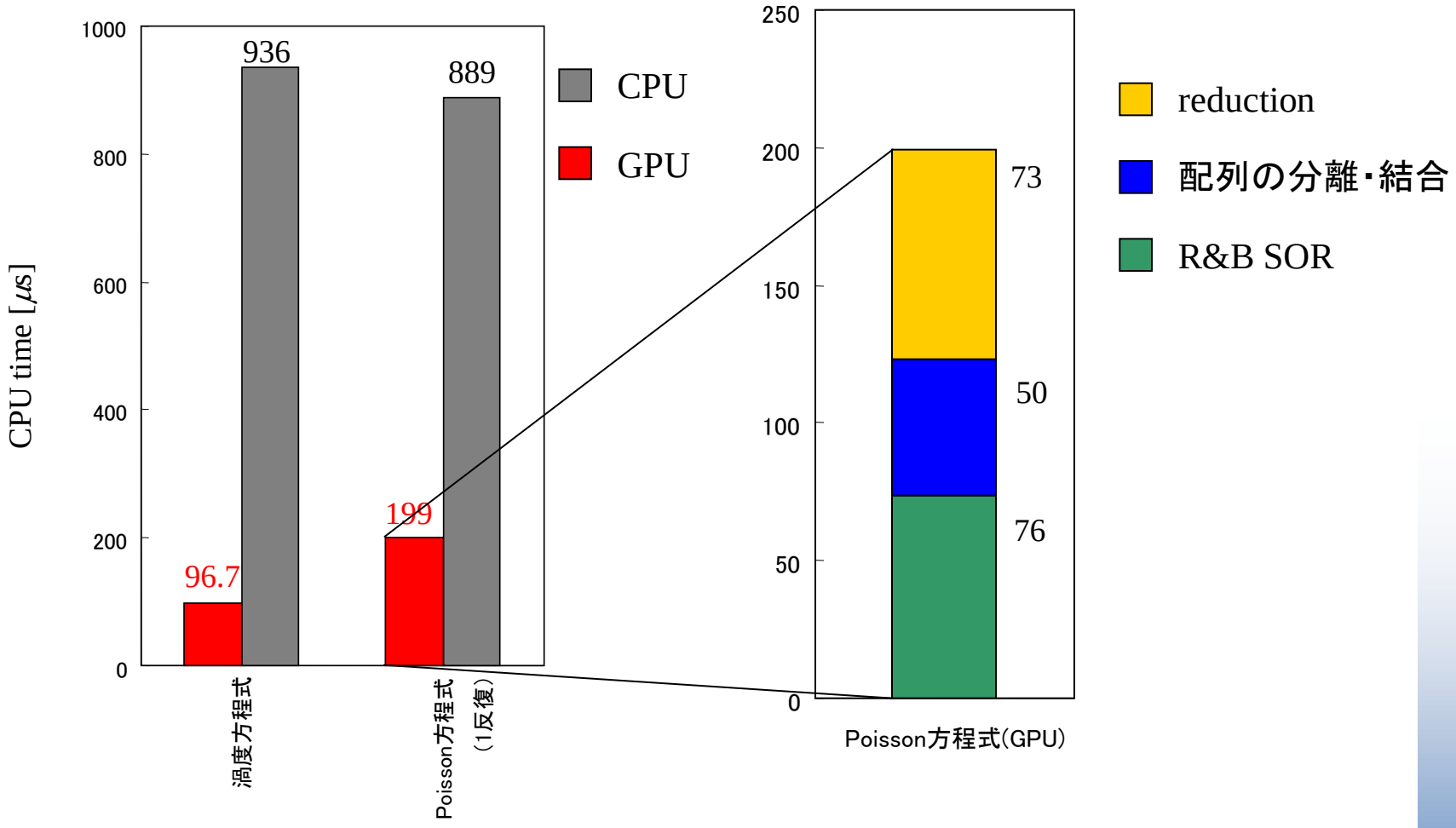
レイノルズ数: $\mathrm{Re}=UL/\nu=1000$

格子分割数: 256×256

計算時間間隔: $U\Delta t/\Delta x=0.1$

- このページは都合により空白です

各方程式の計算時間


CPU time [μs]
1000
800
600
400
200
0
936
889
96.7
199
CPU
GPU
渦度方程式
Poisson方程式
(1反復)
250
200
150
100
50
0
73
50
76
reduction
配列の分離・結合
R&B SOR
Poisson方程式(GPU)

ここまでのまとめ

- 渦度方程式の計算は10倍程度高速化
  - 単純な差分計算だから
- Poisson方程式を解くためのkernelに時間がかかる
  - kernelの内訳を見ると
    - Red&Black SOR法の計算
    - Red&Black SOR法を実行するための配列の分離・結合
    - 収束判定用のreduction

    がほぼ均等な重さ
  - 配列の分離・結合は，反復初回にしか行わない
    - 反復回数が多ければ，影響は小さい
  - 収束判定用reductionは反復ステップごとに実行されるため，高速化が必須
    - そもそも1ステップごとに収束判定？

まとめ

- CUDA Fortranは，普段からFortranを使用してプログラム開発している場合には非常に有用
  - CUDA Cと比較して性能で劣ることはない
  - CPUでプログラムを書くようにGPUプログラムが書ける
    - GPU・CPUを意識せずに，配列の動的確保，データの初期化，parameterの参照が可能
- GPGPUプログラミング自体は概念の理解と慣れが必要
  - 少しの違いで性能が大きく変わる
    - それなりの高速化は難しくないが，超高速を目指すのは難しい
  - 場合によってはアルゴリズム自体を変更する必要がある
    - いわゆるGPGPUとCUDA Fortranの親和性は・・・？
  - 現時点でもハード・ソフトウェア環境の変化が大きい
    - 変化に追従し続けるか，ある程度で妥協するか