正方形ダクト乱流の直接数値計算の GPU 加速

関 本 敦
岡山大学学術研究院 環境生命科学学域

1. はじめに

矩形ダクトなど角を有する流路内の乱流 (図 1(a)) は，熱交換器内部などの熱伝達効率に直結する工学上重要な流れである．正方形ダクト内乱流の平均流には，主流に垂直方向の二次流れがみられる（図 1(c)）．この平均二次流れは，プラントルの第二種二次流れと呼ばれ，その大きさは主流の数%程度であるが，主流の平均的な分布にも影響し，熱や物質輸送に大きく関わる．乱流中の大小さまざまなスケールの渦が側壁の影響を受け, それらの統計的振る舞いに異方性が生まれるためである ([1, 2])．乱流のレイノルズ平均応力などの統計量の定量的な評価には，Navier-Stokes 方程式の直接数値計算 (Direct numerical simulation, DNS) 用いられ，得られた乱流統計量は乱流の基礎研究, 及び乱流モデル開発の良い参照データとして重要であり，データベースとして公開されている．特に，高レイノルズ数の DNS のデータは, 実験では得ることができない 3 次元流れ場の時系列データであり，時々刻々と変化する流れの構造から特徴を抽出し，高精度の乱流モデルを構築するために有用である．これらのビッグデータから有用な情報を抽出するためには，高い情報処理技術が必要である．

乱流の DNS には，少ない格子点数でも高精度に計算できるスペクトル法がよく用いられる．初期の DNS ではベクトル計算機で高速に実施されていたものが，スカラーコンピュータで並列数を上げて計算するものへと変遷し，分散メモリ環境で多くの計算ノードを利用するために，MPI と OpenMP のハイブリッド計算が必須となっている．さらに，近年は，機械学習研究の後押しによって，GPU を搭載した計算機がスーパコンピューティングセンターで導入されることが多くなってきた．計算機の性能向上がヘテロジーニアス環境によって実現されており，研究で用いるプログラムもそれに追随する必要に迫られている．そこで，本稿では，正方形ダクト乱流の DNS コードのボトルネックとなる Helmholtz 方程式の解法を GPU 化して，性能評価を行った結果を紹介する．

2. なぜ正方形ダクト乱流の DNS か？

世界最大規模の正方形ダクト乱流の直接数値シミュレーション (DNS) は現在は $Re_\tau \approx 1000$ 程度 [3, 4] であるが，平行平板間チャネル乱流や境界層乱流データベースでは摩擦速度に基づくレイノルズ数 Re_τ は 8000 程度である [5, 6]．また，平行平板間チャネル乱流や円管内乱流については，GPU を用いた直接数値計算も実施されつつあるが ([7, 8])，GPU を用いた流体の大規模計算の例は少ない．計算機の性能向上がヘテロジーニアス環境 (GPU などのアクセラレータを利用した環境) で維持されているという現状があり, その計算機環境に合わせた乱流の高精度 DNS コードの開発には人的・時間的コストがかかるためであると考えられる.

計算機の性能は約 7 年で 100 倍になっていることから, 正方形ダクト乱流の DNS において断面の 1 辺の格子点数 N を 2 倍にするためには約 4 年かかると見積もることができる. 現在では, 高レイノルズ数 ($Re_\tau = 2000$ 程度, $N \approx 1000$) を実現できると期待できる．つまり，既往の研究 [4] の 2 倍程度のレイノルズ数，また，著者の以前の DNS($Re_\tau = 200$–300, $N = 256$) [2] の約 10 倍のレイノルズ数の乱流を実現することに相当する．

チャネル乱流や境界層乱流では，流れ方向やスパン方向に周期性を課して，フーリエ級数展開を適用する．壁面に垂直な方向が 1 方向のみで，スペクトル法や高精度なコンパクト差分法 [9] を用いた大規模数値計算では各時間ステップで解くべき Helmholtz 方程式や Poisson 方程式は 1 次元となる． Helmholtz 方程式の解法部分は，その他の計算部分 (FFT や非線形項の計算の為の微分操作など) と同程度であり，並列性能を高く保ったまま計算規模を大きくできる．しかしながら正方形ダクトでは，壁面に垂直な方向が 2 方向あるため, 解くべき Helmholtz 方程式が 2 次元となり計算コストが大きい [10] (数値計算の詳細は以前の記事 [11] の式 (7)(8) 参照)．過去に行った正方形ダクト乱流の DNS のスペクトル法 [2] では, 行列積の計算を用いる為, 計算コストは，ダクト断面の 1 辺の格子数を N とすると $O(N^3)$ で増大する．後の研究 [3, 4] でも，同様に 2 次元の Helmholtz ソルバーのコストが問題となる．スペクトル法を用いないという選択もあるが，差分法で高精度を維持するには多くの格子数が必要であり，大規模な計算機環境が必須となり，保存するデータ容量も膨大である．そんな中，最近は，機械学習の GPU 利用に見られるように，行列積の演算を高速に行えるようになってきた．そこで，本稿では, スペクトル法の精度を維持しながら計算コストのボトルネックとなる 2 次元ヘルムホルツ方程式の解法を GPU で加速させた方法の詳細を報告する．3 章では， 2 次元の Helmholtz ソルバーの計算アルゴリズムを説明し，続く 4 章で，BLAS を利用した行列積の計算を GPGPU ライブラリ cuBlas に置き換える方法を解説する．科学計算でよく用いられる Fortran を用いて実装しており，コンパイラにも依存しない実装とした．性能評価を行い，GPU が P100 (Reedbush) と A100 (Wisteria Aquarius) での結果も掲載する．

3. 計算手法

スペクトル法による正方形ダクトの直接数値計算 (DNS) のアルゴリズムと並列化の方法については，[11] を参照して頂きたい．乱流渦を十分に解像するために壁面近傍に格子を細かく切る必要があり，刻み時間ステップの制約を緩和するために，粘性項は時間に対して陰的に取り扱うため，流れの時間発展の反復ごとに速度と圧力のポアソン方程式を解く必要がある．ある時刻での 3 次元の速度と圧力場については，管軸方向にフーリエ級数展開をすることで，速度と圧力の各フーリエ係数について， 2 次元の Helmholtz 方程式 ([11] の (7)(8) 式) を得る．これを適切な境界条件の下で解くが，この解法部分で行列積の演算を必要とする．x 方向速度 u の波数 k_x に対するフーリエ

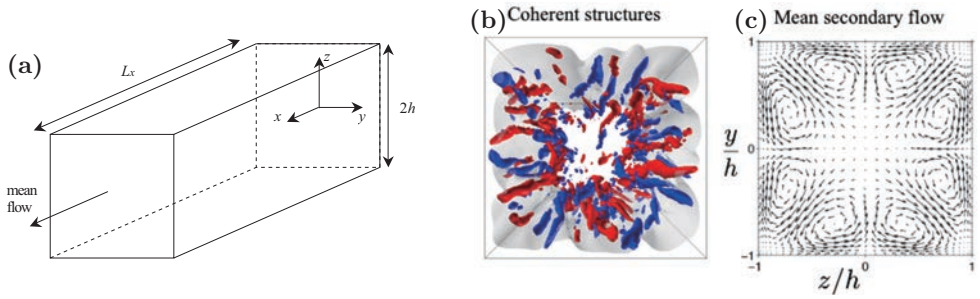

図 1　(a) 正方形ダクトの外観．(b) 乱流中の瞬時渦の様子．流れ方向の等値面を灰色で，時計回り・反時計回りの渦構造を赤と青で示す．流れは手前から奥．(c) 断面内平均二次流れ．

係数 $\hat{u}$ を例に，2 次元の高速対角化解法 [10] を説明する．解くべき Helmholtz 方程式は，

$$\nabla^2_{2D}\hat{u}(y,z) - a\hat{u}(y,z) = F \tag{1}$$

ここで，$\nabla^2_{2\mathrm{D}} \equiv \frac{\partial}{\partial^2 y} + \frac{\partial}{\partial^2 z}$, で, a は流れ方向の波数 k_x に依存する定数，F は Navier-Stoke 方程式の非線形項と圧力勾配と境界条件から求まる．$U = (U_{j,k}) = \hat{u}(y_j, z_k)$, y 方向と z 方向の格子点数を同じ N として，境界条件も考慮した微分係数行列を D として，

$$D^2U + U(D^2)^T - aU = F \tag{2}$$

と表現する．本研究では，チェビシェフ・ガウス・ロバット配置された点 ($y_j = \cos(\pi j/N)$, $z_k = \cos(\pi k/N)$) を考えるため $D(= D_y = D_z)$ は密な実対称行列である（フーリエ係数の実部と虚部は別々に計算する）．対角化 $D^2 = P\Lambda P^{-1}$ を施し，$\tilde{U} = P^{-1}UP^{-T}$, $\tilde{F} = P^{-1}FP^{-T}$ とおくと，$(P\Lambda P^{-1})^T = P^{-T}\Lambda P^T$ に注意して，

$$(P\Lambda P^{-1})U + U(P^{-T}\Lambda P^T) - aU = F \tag{3}$$

$$\Lambda\tilde{U} + \tilde{U}\Lambda - a\tilde{U} = \tilde{F} \tag{4}$$

$$\tilde{U}_{j,k} = \frac{\tilde{F}_{j,k}}{\lambda_j + \lambda_k - a} \tag{5}$$

である．したがって，解は $U = P\tilde{U}P^T$ と求まる．P^{-1} とその転置行列 P^{-T}, 及び固有値ベクトル Λ を予め計算しておき，以下の 3 つの手順

(Step 1) $\tilde{F} = P^{-1}FP^{-T}$

(Step 2) 式 (5)

(Step 3) $U = P\tilde{U}P^T$

を実装すれば良い．Step 1 と 3 では，2 回ずつ行列積の倍精度演算を行うためボトルネックとなる．行列積の計算は，Level 3 BLAS の DGEMM を用い，GPU の場合は CUBLAS を適用する．ここで，N が大きい場合に行列積 $C = AB$ を高速に実行するための工夫をする．メモリへのアクセスが連続的になるように，行列 A の転置操作を先にしておいてから，DGEMM ルーチンに渡し，$C = (A^T)^T \times B$ を計算させると非常に高速である．また，Step 2 の分母の計算は固有値の和を要素に持つ行列 $E_{j,k} = \lambda_j + \lambda_k$ を用意しておく．最終的に，以下のような実装となる．

$$\text{(Step 1)}\quad \tilde{F}\ (= P^{-1}FP^{-T}) = \underbrace{\underbrace{\left\{\underbrace{\left[\underbrace{(P^{-T})^T \times F}_{\text{(i) dgemm}}\right]^T}_{\text{(ii) transpose}}\right\}^T}_{\text{(iii) dgemm}} \times P^{-T}}$$

$$\text{(Step 2)}\quad \tilde{U}_{j,k} = \frac{\tilde{F}_{j,k}}{E_{j,k} - a}$$

$$\text{(Step 3)}\quad U\ (= P\tilde{U}P^T) = \underbrace{\underbrace{\left\{\underbrace{\left[\underbrace{(P^{T})^T \times \tilde{U}}_{\text{(i) dgemm}}\right]^T}_{\text{(ii) transpose}}\right\}^T}_{\text{(iii) dgemm}} \times P^{T}}$$

4. GPU 化の詳細な実装方法

現行の DNS プログラムは Fortran で書かれており，今回は過去の遺産を有効利用しながら GPU での高速化を目指す．OpenACC では十分に高速化できない場合もあり，人類の宝である Blas ライブラリの利用を心がける．また，MPI 並列計算のように明示的に GPU デバイスへのデータ転送の命令を書き，Intel Fortran から cuBlas を呼び出す仕様とした．これには，CUDA のインストールフォルダ (CUDA_INSTALL_DIR)/src/にある wrapper と interface を利用する (fortran.c, fortran.h, fortran_common.h)．微分係数行列が疎な場合は, cuSparse ライブラリ用の cusparse_fortran.c, cusparse_fortran.h, cusparse_fortran_common.h が利用できる．cuBlas のバージョン v8 から v11 までは同じもので動作するのを確認したが, cuSparse を呼び出す場合は v10 から v11 で仕様が変わっているので注意する．cuSparse ではバッチ化された処理も利用できる．cuBlas のライブラリで対応できない式 (5) の計算と転置行列については, CUDA でソースコードを C interface と合わせて用意する必要がある．以下に行列の転置の CUDA コードを示す (Coalesced Transpose, NVIDIA ブログ “An Efficient Matrix Transpose in CUDA C/C++” (https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/より)．

Listing 1　GPU メモリ上での行列転置関数 (my_transpose.cu).

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#if defined(__GNUC__)
#include <stdint.h>
#endif /* __GNUC__ */
#include "cuda.h"
#include "cuda_runtime.h"
//#include <cublas.h>
#include <cublas_v2.h>

#define TILE_DIM 32
#define BLOCK_ROWS 8

typedef size_t ptr_t;

__global__ void transposeCoalesced(double *odata,
                                   double *idata, int width, int height)
{
  __shared__ double tile[TILE_DIM][TILE_DIM];
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;
  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;
```

```
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] =
      idata[index_in+i*width];
  }
  __syncthreads();
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i*height] =
      tile[threadIdx.x][threadIdx.y+i];
  }
}
// interfaces
extern "C" {
  int transpose_gpu_(ptr_t *handle, const int *Nx, const int *Ny,
                const double *alpha, const ptr_t *devPtrA, ptr_t *devPtrC)
  {
    double *A = (double *)(*devPtrA);
    double *C = (double *)(*devPtrC);
    const int size_x = *Nx;
    const int size_y = *Ny;
    // execution configuration parameters
    dim3 grid(size_x/TILE_DIM, size_y/TILE_DIM), threads(TILE_DIM,BLOCK_ROWS);
    transposeCoalesced<<<grid, threads>>>(C, A, size_x, size_y);
    return 0;
  }
}
```

以下のようにオブジェクトファイルを生成する．

Listing 2　fotran.c のコンパイル

```
CUDAINC = /$(CUDA_INSTALL_DIR)/$(CUDA_VER)/include/
CUDALIB = /$(CUDA_INSTALL_DIR)/$(CUDA_VER)/lib64/
CCFLAGS = -DARCH_64=1 -DCUBLAS_GFORTRAN -I$(CUDAINC)
icc -c $(CCFLAGS) fortran.c
nvcc -c $(CCFLAGS) my_transpose.cu
```

Fortran ソースコードとリンクさせるには，以下のオプションを付けておく．

```
-I$(CUDAINC) -L$(CUDALIB) -lcudart -lcublas -lcublas_device \
-lcudadevrt [-lcusparse]
```

ここまでで, cuBlas を利用する準備ができた．次に, 実際の Fortran コードの書き換えの例を Step 1-(i) dgemm, $(P^{-T})^T \times F$ で示す．オリジナルの CPU 版コードの dgemm に関わる allocate と call 部分である．予め $N \times N$ の行列 P^{-T} が計算済みで PM1T に格納してある．

Listing 3　Fortran オリジナルコード例

```
allocate(PM1T(N,N),F(N,N),FT(N,N))
... ! pre-process of P, PT, PM1, PM1T, E !
call dgemm('t','n',N,N,N,1.d0,PM1T,N,F,N,0.d0,FT,N)
```

はじめに，cublas の初期化をしておく．自作の transpose 関数ではデバイスの handle が必要なのでここで作成しておく．

Listing 4　Fortran から cuBlas を初期化

```
external cublas_init, cusparse_create
integer*8 handle
...
call cublas_init
istat = cusparse_create(handle)
```

次に，GPU メモリ上に int8 型のデバイスポインタを定義し，cublas_allocate を用いて GPU デバイス上で配列領域を確保する．以下は，プリプロセスで計算した double 型 $N \times N$ 配列 PM1T を転送する例である．GPU メモリにデータを送るには，このポインタと行列のサイズを指定して，cublas_set_matrix を用いる (メモリの確保に失敗したときはエラーメッセージを表示させないと気付かずにそのまま実行される).

Listing 5　Fortran から cuBLAS でデバイスメモリにコピーする例

```
external cublas_alloc, cublas_set_matrix
integer*8 dp_PM1T, devPtrF, devPtrFT
...
istat= cublas_alloc(N*N, 8, dp_PNM1T)
if (istat.ne.0) write(*,*) 'cublas alloc. error: PNM1T',istat
istat= cublas_set_matrix(N, N, 8, PM1T, N, dp_PM1T, N)
if (istat.ne.0) write(*,*) 'cublas setmat error: PM1T',istat
```

また，GPU に送る配列 F と GPU メモリ上で一時的に FT を保存しておく領域を確保して初期化しておく．

```
istat= cublas_alloc(N*N, 8, devPtrF)
istat= cublas_set_matrix(N,N,8,0.d0,0,devPtrF,1) ! zero
istat= cublas_alloc(N*N, 8, devPtrFT)
istat= cublas_set_matrix(mm2+1,mm2+1,nk,0.d0,0,devPtrFT,1) ! zero
```

cuBlas の実行と転置操作は以下のようになる．

```
external cublas_set_matrix, cublas_dgemm, transpose_gpu
...
istat = cublas_set_matrix(N,N,8,F,N,devPtrF,N)
istat = cublas_dgemm('t','n',N,N,N,1.d0,dp_PM1T,N,devPtrF,N.0.d0,devPtrFT,N)
istat = transpose_gpu(handle,int(N,kind=8),int(N,kind=8),1.d0, &
                      devPtrFT,dp_tmp2d)
...
```

最終的に Helmholtz ソルバーの結果 U をデバイスメモリから取り出すには，cublas_get_matrix を用いる．

```
external cublas_get_matrix
...
istat = cublas_get_matrix(N,N,8,devPtrU,N,U,N)
```

プログラムの最後で `cublas_free` と `cublas_shutdown` で GPU メモリを開放しておく．

5. 性能評価

2 次元 Helmholtz 方程式の CPU 版の結果と GPU 版の性能比較を図 2 に示す．(現在試行利用中の Wisteria Aquarius (A100) での性能評価も合わせて示す．) CPU での実行時間は N^3 で増えていることが確認できる．それに対して，GPU での実行時間（ホストメモリからデバイスメモリのデータ移動の時間を含む）から，$N \approx 1000$ で 10 倍程度の高速化が達成できており，N が大きいほど高速化率も高い．Wisteria では小さいデータ ($N < 512$) を送ると逆に性能が落ちる傾向が見られたが，$N > 1000$ では P100 よりも早い．これは GPU デバイスへのデータの送受信の速度向上が要因である．(PCI express Gen3 と Gen4 では，転送速度は 2 倍の性能差がある．)

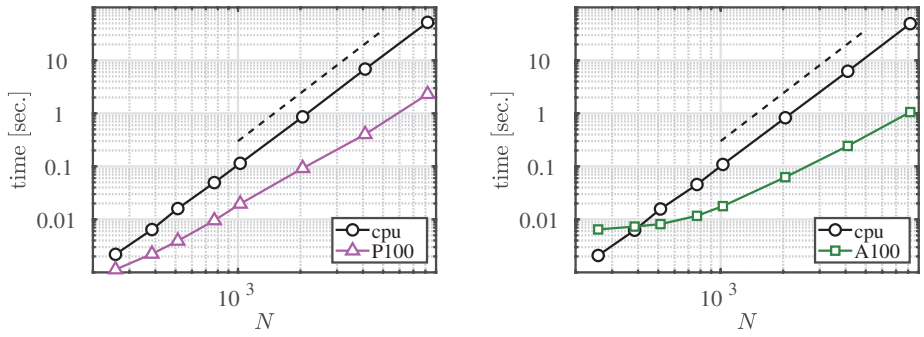

図 2　Helmholtz ソルバーの性能評価．実行時間計測にはホストメモリから GPU デバイスメモリへのデータ転送を含む．CPU での実行時間（黒）と GPU での実行時間 (マゼンタが Reedbush-H, P100，緑は Wisteria-A, A100) を縦軸に，横軸 N は，1 辺の格子点数．破線は N^3 のスケーリングの傾きを示す．

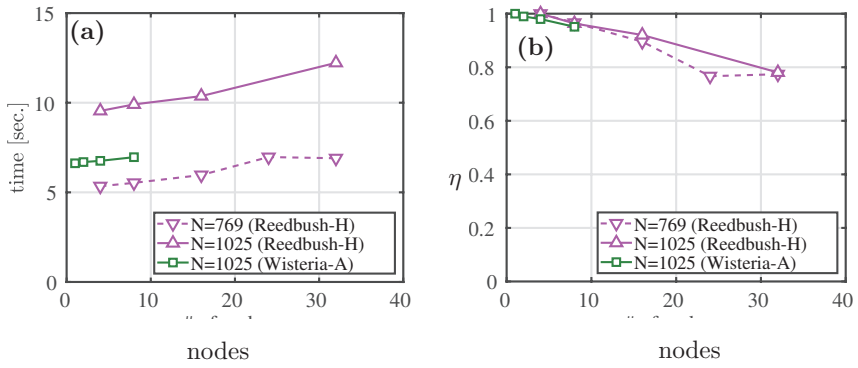

図 3　Weak scaling の結果．(a) DNS コード全体の実行時間計測結果. 格子点数は $N_x = 96 \times (node), N_y = N_z = N$. (b) 並列化効率 (マゼンタが Reedbush-H (32 process/node), 緑は Wisteria-A (32 process/node) の結果.)

コード全体の Weak scaling の結果を図 3 に示す．同じ規模の計算でも Wisteria-A の方が高速に計算できるが，InfiniBand EDR と InfiniBand HDR の違いによって，通信性能が大きく向上したのが主な要因である．

6. まとめ

今回実装した 2 次元 Helmholtz ソルバーの GPU 加速によって，ダクトの DNS 計算でボトルネックは取り除けた．CPU 環境では GPU を使えば恐れずに計算規模を大きくすることができる．今回は，複数 GPU 利用では 1GPU の場合と比べて 10〜20% 程度しか高速化できず，2GPU 以上の有効利用の課題が残った．現状の CPU/GPU ハイブリッドコードでは，1CPU あたり 1GPU で加速するのが効率がよく，Reedbush-H の構成で大量の計算ノードを用いるのが良さそうである．Reedbush-H では，$3072(x) \times 1025(y) \times 1025$ の計算規模で 32 ノードを利用すると，1step の計算時間が 12 秒程度であった（図 3(a)）．プロダクトランの十分な統計量が得られる 100 万ステップの計算にはおよそ 10 万ノード時間程度で Reedbush-H (P100 × 2) で 3 ヶ月程度である．

情報基盤センターで導入されている複数 GPU を利用した流体計算の為には，結局，フル GPU バージョンのコード開発が必要である．GPU プログラミングも MPI 並列計算と同様，まだ学んでいない人にとっては非常にとっつきにくいイメージがあるが，少し頑張って最初の障壁を乗り越えれば，得られるものは大きい．MPI 並列に馴染みがある人が GPU 利用する場合は，cuBlas ライブラリでも OpenACC であっても，はじめから CPU/GPU 間のデータ転送については明示的にコーディングする方が良い．最新の A100 では GPU 上のメモリが 40GB もあるので，少ない格子点数で高精度の結果が得られるスペクトル法を用いると，1GPU だけでも中規模計算を高速に実施できる．1 ケースの超大規模計算を目指すだけではなく，パラメターを振りながら小中規模の計算を多く行う研究での利用価値も高いと思われる．

謝辞

若手利用推薦課題「世界最高レイノルズ数乱流データベース構築のための GPU-DNS コードの作成」(2019 年度) 及び「GPU 加速 DNS コードを用いた正方形ダクト乱流の直接数値計算」(2020 年度) の援助を受けた．また，結果の一部は Wisteria/BDEC-01 での試行利用を活用した．

参考文献

[1] S. Gavrilakis, *J. Fluid Mech.* **244**, 101 (1992).

[2] A. Pinelli, M. Uhlmann, A. Sekimoto, G. Kawahara, *J. Fluid Mech.* **644**, 107 (2010).

[3] 河原源太, 中辻竜也, 清水雅樹, ウルマン・マルクス, ピネリ・アルフレド, 日本機械学会年次大会 **2012**, J054023 (2012).

[4] 森下誠, 河原源太, 清水雅樹, 流体工学部門講演会講演論文集 **2016**, 1008 (2016).

[5] M. Lee, R. D. Moser, *J. Fluid Mech.* **774**, 395–415 (2015).

[6] Y. Yamamoto, Y. Tsuji, *Phys. Rev. Fluids* **3**, 012602 (2018).

[7] A. Vela-Martín, M. P. Encinar, A. García-Gutiérrez, J. Jiménez, *arXiv:1808.06461* (2019).

[8] J. M. López, *et al.*, *SoftwareX* **11**, 100395 (2020).

[9] S. K. Lele, *J. Comput. Phys.* **103**, 16 (1992).

[10] P. Haldenwang, G. Labrosse, S. Abboudi, *J. Comput. Physics* **55**, 115 (1984).

[11] 関本敦, 河原源太, M. Uhlmann, P. Alfredo, 東京大学情報基盤センター, スーパーコンピューティングニュース **10 (5)** (2008).