# PARALLEL COMPUTATION OF SOME INCOMPRESSIBLE VISCOUS FLOW PROBLEM BY BLOCK DECOMPOSITION

Sizil Krishna

Indian Institute of Technology Guwahati

May 6, 2014

# Fluid Dynamics

## Introduction

Generally the steps to tackle Computer Simulation is as follows.

- Fetch governing equations.
- Discritize equations to apply computation.
- Compute / Simulate problem using relevant algorithms.

We'll be computing fluid flow in some standard problem domains following the above mentioned steps.

Governing equations are,

**Navier-Stokes** equation is given as,

$$\rho\left(\frac{\partial V}{\partial t} + V.\bigtriangledown V\right) = -\bigtriangledown p + \mu\bigtriangledown^2 V \tag{1}$$

**Continuity equation** is given as,

$$\frac{\partial \rho}{\partial t} + \nabla(\rho V) = 0 \tag{2}$$

where, $\rho$ is density, $V = u\hat{i}, v\hat{j}, w\hat{k}$ is velocity vector, $\mu$ is the (constant) dynamic viscosity, $p$ is pressure and $\bigtriangledown = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$.

For incompressible fluid flow in 2-D, the continuity equation can be rewritten as,

$$\rho = constant$$
$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{3}$$

where $u$ and $v$ are the velocities in the x and y coordinate directions, respectively. Now we can introduce Stream function $\psi$ for a two dimensional flow as ,

$$u = \frac{\partial \psi}{\partial y}, \qquad v = -\frac{\partial \psi}{\partial x} \tag{4}$$

Where Stream function represents trajectories of particles in a steady flow.

The vorticity of a flow is a pseudovector field $\omega$, equal to the curl (rotational) of its velocity field $V$.
It describes the local spinning motion of a fluid near some point. Thus vorticity can be written as,

$$\begin{aligned}
\omega &= \nabla \times V = \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \\ u & v \end{bmatrix} \\
&= \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \\
&= -\frac{\partial^2 \psi}{\partial x^2} - \frac{\partial^2 \psi}{\partial y^2} \\
&= -\nabla^2 \psi
\end{aligned}$$

Or,

$$\nabla^2 \psi = -\omega \tag{5}$$

## Discritization of equations

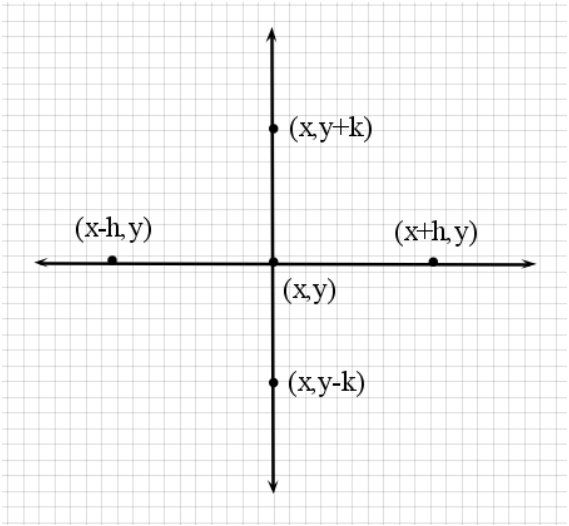
**Finite-difference methods** are numerical methods for approximating the solutions to differential equations using finite difference equations to approximate derivatives.
Discritization makes it easy to solve a equation using advanced computational iterators and solvers.

$$Forward - difference \quad : \quad f'(a) \approx \frac{f(a+h) - f(a)}{h}$$

$$Backward - difference \quad : \quad f'(a) \approx \frac{f(a) - f(a-h)}{h}$$

$$Central - difference \quad : \quad f'(a) \approx \frac{f(a+h) - f(a-h)}{2h}$$

where $h$ being the gap between two consecutive points.




Finite-difference scheme of a second order derivative can be given as,

$$f_{xy}(x,y) \approx \frac{f(x+h,y+k) - f(x+h,y-k) \\ -f(x-h,y+k) + f(x-h,y-k)}{4hk} \tag{6}$$

where $h$ being the gap between two consecutive points in x direction and $k$ in y direction.

**Steady State Flow**

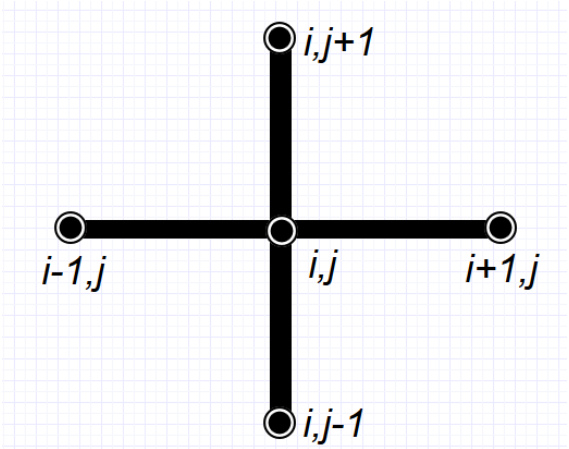
Steady state flow means that no flow parameters (velocity, pressure etc ) depend on time. The time dependant terms in Navier-Stokes and continuity equations drop off. The governing equations are,

$$\frac{\partial \psi}{\partial y}.\frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x}.\frac{\partial \omega}{\partial y} = \frac{1}{Re}\left[\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}\right] \tag{7}$$

where $Re$ is Reynolds number (constant introduced to make equations dimensionless). The discretized form of above equation is,

$$\begin{aligned} \psi_{i,j} &= \frac{\psi_{i+1,j} + \psi_{i-1,j} + \beta^2(\psi_{i,j+1} + \psi_{i,j-1}) + (\Delta x)^2 \omega_{i,j}}{2(1+\beta^2)} \\ \beta &= \frac{\Delta x}{\Delta y} \end{aligned} \tag{8}$$

where, $\Delta x$ is grid size in $X$ direction and $\Delta y$ is grid size in $Y$ direction.




$$\omega_{i,j} = \frac{\begin{array}{r}\omega_{i+1,j} + \omega_{i-1,j} + \beta^2(\omega_{i,j+1} + \omega_{i,j-1}) - \\ \frac{1}{4}\beta Re(\omega_{i+1,j} - \omega_{i-1,j})(\psi_{i,j+1} - \psi_{i,j-1}) + \\ \frac{1}{4}\beta Re(\omega_{i,j+1} - \omega_{i,j-1})(\psi_{i+1,j} - \psi_{i-1,j})\end{array}}{2(1+\beta^2)} \tag{9}$$

**Unsteady State Flow**

Unsteady state means that various quantities like pressure, velocity etc depend on time. The Navier-Stokes equation in $\psi$-$\omega$ form is,

$$\frac{\partial \omega}{\partial t} + \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right) \tag{10}$$

Discretized forms are,

$$\begin{aligned} \psi_{i,j} &= \frac{\psi_{i+1,j} + \psi_{i-1,j} + \beta^2(\psi_{i,j+1} + \psi_{i,j-1}) + (\Delta x)^2 \omega_{i,j}}{2(1+\beta^2)} \\ \beta &= \frac{\Delta x}{\Delta y} \end{aligned} \tag{11}$$

where, $\Delta x$ is grid size in $X$ direction and $\Delta y$ is grid size in $Y$ direction.

*XSweep* :

$$\frac{\omega_{i,j}^{n+\frac{1}{2}} - \omega_{i,j}^{n}}{\frac{\Delta t}{2}} + v_{i,j}^{n}\left[\frac{\omega_{i,j+1}^{n} - \omega_{i,j-1}^{n}}{2h}\right] + v_{i,j}^{n+\frac{1}{2}}\left[\frac{\omega_{i+1,j}^{n+\frac{1}{2}} - \omega_{i-1,j}^{n+\frac{1}{2}}}{2h}\right] =$$

$$\frac{1}{Re}\left[\left[\frac{\omega_{i+1,j}^{n+\frac{1}{2}} + \omega_{i-1,j}^{n+\frac{1}{2}} - 2\omega_{i,j}^{n+\frac{1}{2}}}{h^2}\right] + \left[\frac{\omega_{i,j+1}^{n} + \omega_{i,j-1}^{n} - 2\omega_{i,j}^{n}}{h^2}\right]\right] \quad (12)$$

*YSweep* :

$$\frac{\omega_{i,j}^{n+1} - \omega_{i,j}^{n+\frac{1}{2}}}{\frac{\Delta t}{2}} + v_{i,j}^{n+1}\left[\frac{\omega_{i,j+1}^{n+1} - \omega_{i,j-1}^{n+1}}{2h}\right] + v_{i,j}^{n+\frac{1}{2}}\left[\frac{\omega_{i+1,j}^{n+\frac{1}{2}} - \omega_{i-1,j}^{n+\frac{1}{2}}}{2h}\right] =$$

$$\frac{1}{Re}\left[\left[\frac{\omega_{i+1,j}^{n+\frac{1}{2}} + \omega_{i-1,j}^{n+\frac{1}{2}} - 2\omega_{i,j}^{n+\frac{1}{2}}}{h^2}\right] + \left[\frac{\omega_{i,j+1}^{n+1} + \omega_{i,j-1}^{n+1} - 2\omega_{i,j}^{n+1}}{h^2}\right]\right] \quad (13)$$

**Gauss-Seidel method**

The Gauss Seidel method is an iterative technique for solving a square system of n linear equations with unknown x.

$$Ax = b$$

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{14}$$

where $A \epsilon R^{n,n}$ , $b \epsilon R^n$ and $x \epsilon R^n$. Then the decomposition of A into its lower triangular component and its strictly upper triangular component is given by,

$$A = L_* + U$$

$$L_* = \begin{bmatrix} a_{1,1} & 0 & \dots & 0 \\ a_{2,1} & a_{2,2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix}, U = \begin{bmatrix} 0 & a_{1,2} & \dots & a_{1,n} \\ 0 & 0 & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \quad (15)$$

$$L_* x = b - Ux$$

$$x^{k+1} = L_*^{-1}(b - Ux)$$

However, by taking advantage of the triangular form , the elements of x(k+1) can be computed sequentially using forward substitution,

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>1} a_{ij} x_j^{(k)} \right), i, j = 1, 2, \dots, n \quad (16)$$

The procedure is generally continued until the changes made by an iteration are below some tolerance, such as a sufficiently small residual.

Currently we have used Gauss-Seidel method to solve,

- Steady State's Stream function (8)
- Steady State's Vorticity (9)
- Unsteady State's Stream function (11)

**Alternating direction implicit method**

Alternating Direction Implicit (ADI) method is a finite difference method for solving parabolic, hyperbolic and elliptic partial differential equations.

The idea behind the ADI method is to split the finite difference equations into two, one with the x-derivative taken **implicitly** and the next with the y-derivative taken **implicitly**, as we have done in solving Unsteady State's Vorticity, (12) and (13).

The advantage of the ADI method is that the equations that have to be solved in each step have a simpler structure and can be solved efficiently with the **tridiagonal matrix algorithm**.

## Tridiagonal matrix algorithm

In numerical linear algebra, the tridiagonal matrix algorithm, also known as the Thomas algorithm (named after Llewellyn Thomas), is a simplified form of Gaussian elimination that can be used to solve tridiagonal systems of equations. A tridiagonal system for n unknowns may be written as,

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$$

where $a_1 = 0$ and $c_n = 0$. In matrix form,

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & 0 \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix} \tag{17}$$

A first sweep eliminates the $a_i$'s, and then an backward substitution produces the solution. The forward sweep consists of modifying the coefficients as follows, denoting the new modified coefficients with primes,

$$c_i' = \begin{cases} \frac{c_i}{b_i} & : \quad i = 1 \\ \frac{c_i}{b_i - c_{i-1}' a_i} & : \quad i = 2, 3, 4, ..., n-1 \end{cases} \tag{18}$$

$$d_i' = \begin{cases} \frac{d_i}{b_i} & : \quad i = 1 \\ \frac{d_i - d_{i-1}' a_i}{b_i - c_{i-1}' a_i} & : \quad i = 2, 3, 4, ..., n \end{cases} \tag{19}$$

The solution is then obtained by back substitution,

$$x_n = d_n' \tag{20}$$

$$x_i = d_i' - c_i' x_{i+1}, \quad i = n-1, n-2, ..., 1 \tag{21}$$

# CUDA

- CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce.
- The CUDA platform is accessible to software developers through CUDA-accelerated libraries and standard programming languages, like C/C++.
- Unlike CPUs, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly.

# CUDA

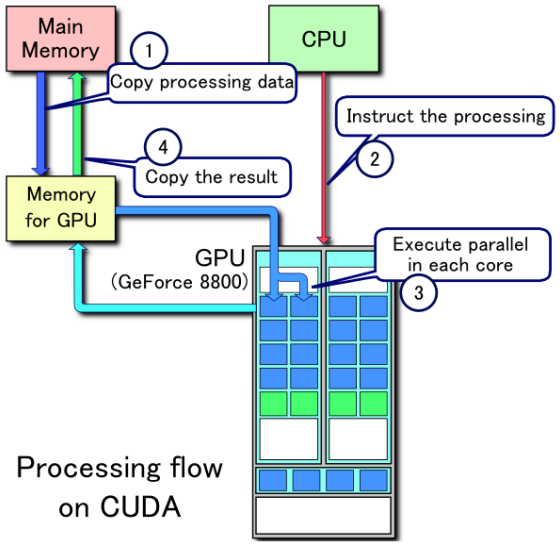
**Why CUDA?**

- CUDA provides ability to use high-level languages such as C to develop application that can take advantage of high level of performance and scalability that GPUs architecture offer.
- GPUs contain much larger number of dedicated ALUs then CPUs.
- GPUs allow creation of very large number of concurrently executed threads at very low system resource cost.
- CUDA also exposes fast shared memory that can be shared between threads.
- Compiled code will run directly on GPU.

**CUDA limitations**

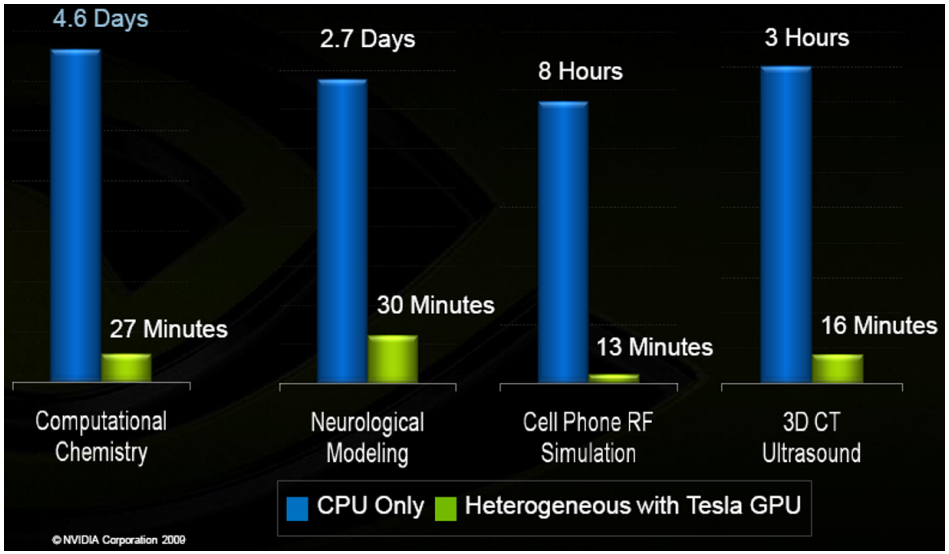
- No support of recursive function. Any recursive function must be converted into loops.
- Only Nvidia GPU's are supported.

## CUDA process flow




1. Copy data from main memory to GPU memory.
2. CPU instructs the process to GPU.
3. GPU execute parallel in each core.
4. Copy the result from GPU memory to main memory.

# CUDA




Comparison released by Nvidia for different simulations fields.

## Sample C implementation for CUDA

```
__global__ void matrix_mul(...){
        ...
        [GPU (Parallel) code]
        ...
}

void main(){
        ...
        [CPU (serial) code]
        [Copy CPU variables to GPU]
        matrix_mul<<<dimGrid, dimBlock>>>(...)
        [Copy GPU variables to CPU ]
        [CPU (serial) code]
        ...
}
```

## Sample CUDA Kernel

```
__global__ void kernelFunc(...)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x
    if(idx < size)
        {
            ...
        }
}
```

- CUDA uses kernel functions. These functions are called from device (GPU) and are executed on it simultaneously by many threads in parallel.
- There are several built in variables that are available to kernel call:
  blockIdx - block index within grid.
  threadIdx - thread index within block.
  blockDim - number of threads in a block.

## CUDA execution flow

- At application start of execution CUDA's compiled code runs like any other application. Its primary execution is happening in CPU.
- When kernel call is made, application continue execution of non-kernel function on CPU. In the same time, kernel function does its execution on GPU. This way we get parallel processing between CPU and GPU.
- Memory move between host and device is primary bottleneck in application execution. Execution on both is halted until this operation completes.

**Results**

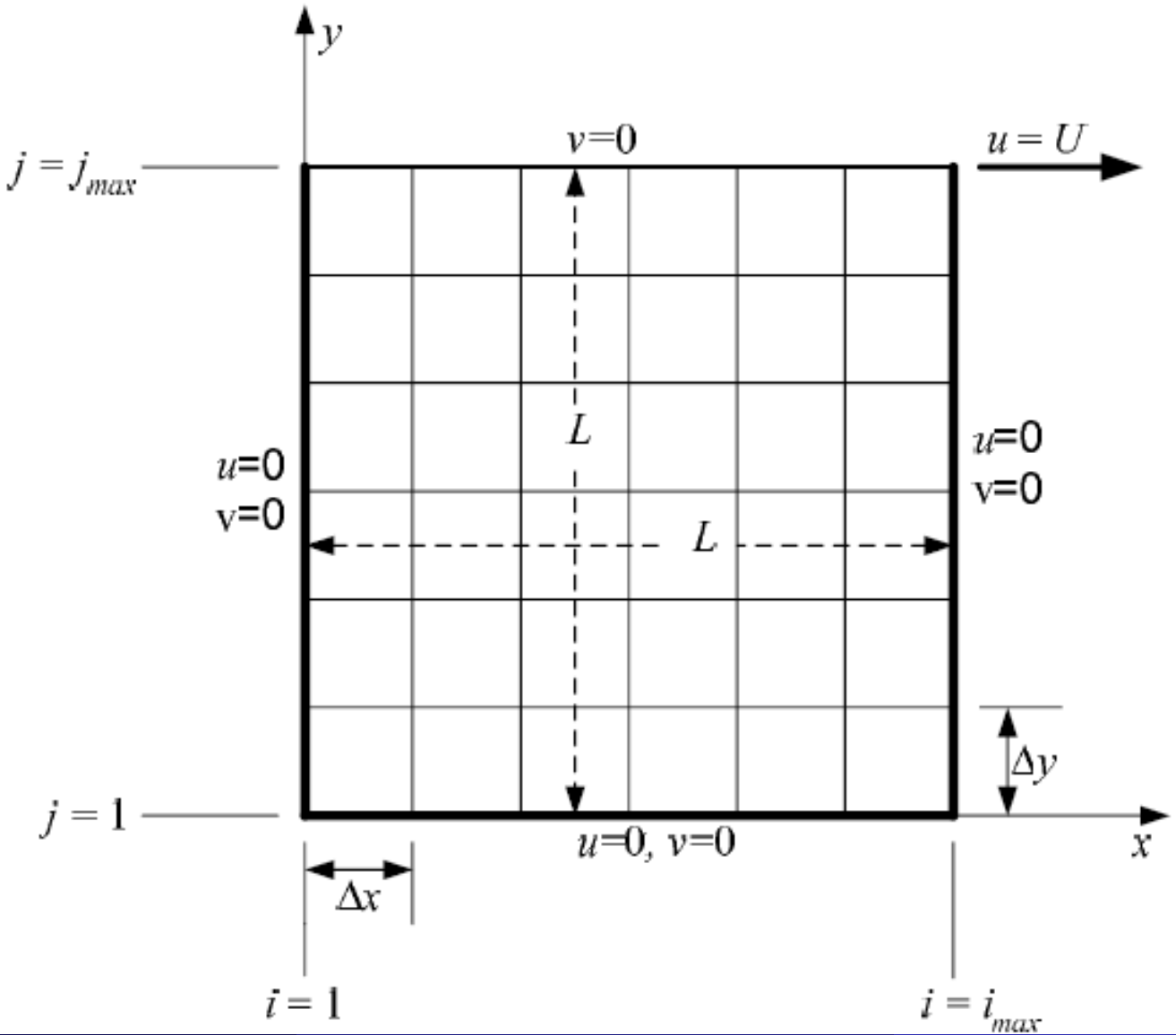
**Steady State - Lid Driven Cavity**

The problem geometry is simple and two-dimensional. Problem domain consists of a square block filled with some incompressible fluid, where top lid is constantly moving with constant velocity in fixed direction while other three walls are solid.

The flow pattern generated by fluid particles in this given circumstances can be simulated using Gauss-Seidel iterator on previously mentioned discritized form of Steady State Navier-Stokes (8) and Continuity equation (9).

# Results

## Steady State



The Boundary conditions used in our problem are,

$$\psi = 0, \quad \forall i, j$$

$$LeftWall \quad : \quad \omega_{i,j} = -\frac{2\psi_{2,j}}{(\Delta x)^2}, \quad for 2 \leq j \leq j_{max-1}$$

$$RightWall \quad : \quad \omega_{imax,j} = -\frac{2\psi_{imax-1,j}}{(\Delta x)^2}, \quad for 2 \leq j \leq j_{max-1}$$

$$BottomWall \quad : \quad \omega_{i,1} = -\frac{2\psi_{i,2}}{(\Delta y)^2}, \quad for 2 \leq i \leq i_{max-1}$$

$$TopWall \quad : \quad \omega_{i,jmax} = -\frac{2\psi_{i,jmax-1} + 2\Delta y U}{(\Delta y)^2}, for 2 \leq i \leq i_{max-1}$$

# Results

## Steady State

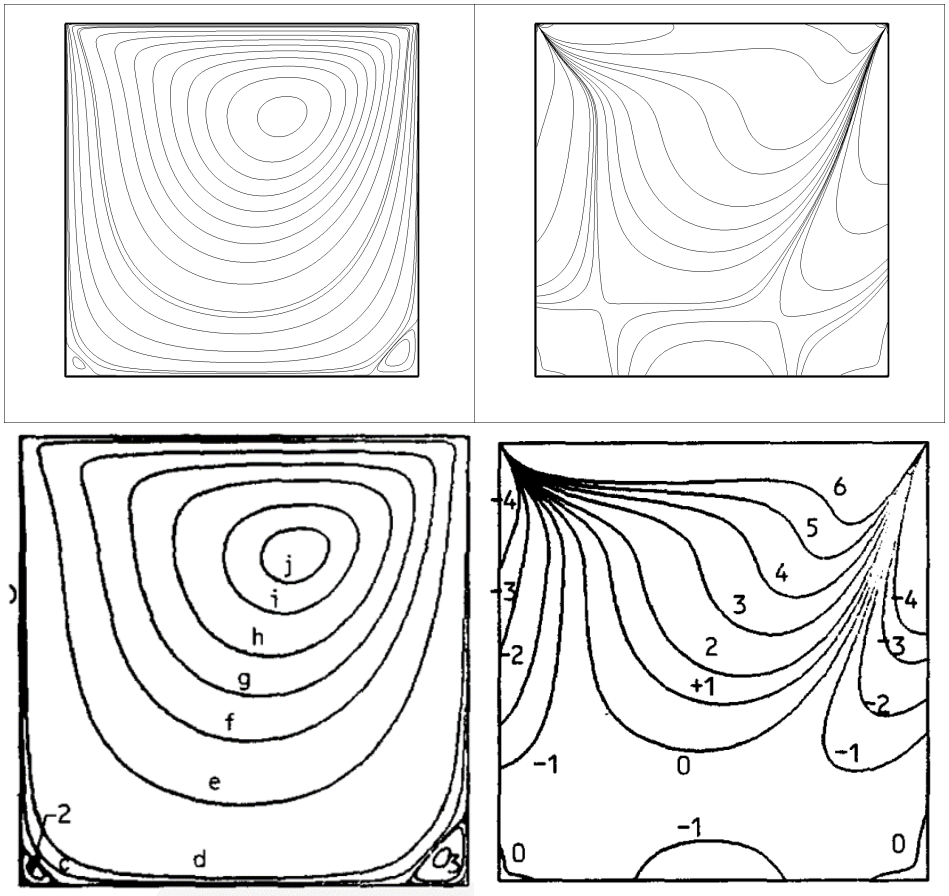



Figure: Streamlines and Vorticity Plot. Our Results vs Ghia, et al [1] at Re 100

# Results

## Steady State

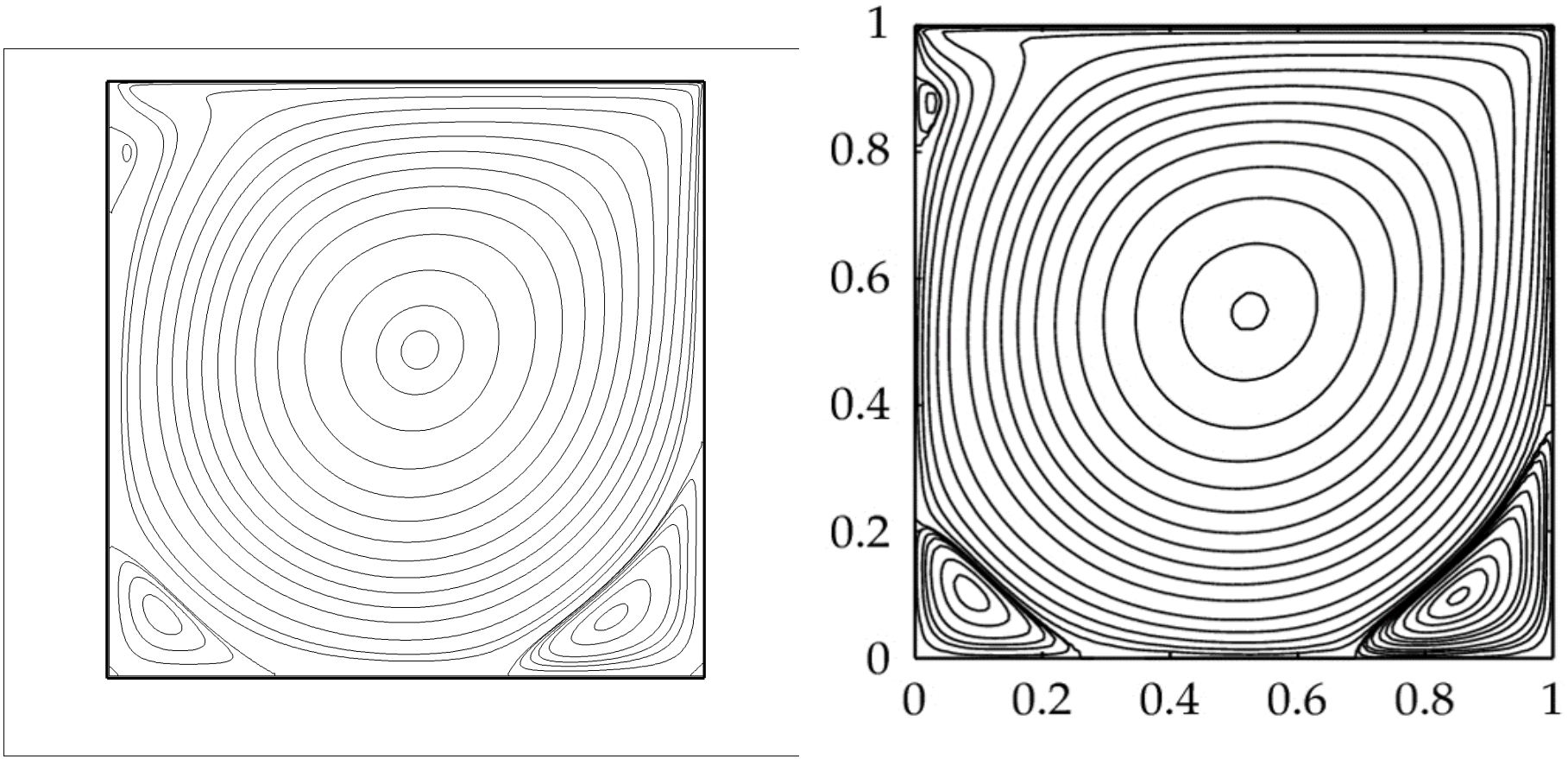
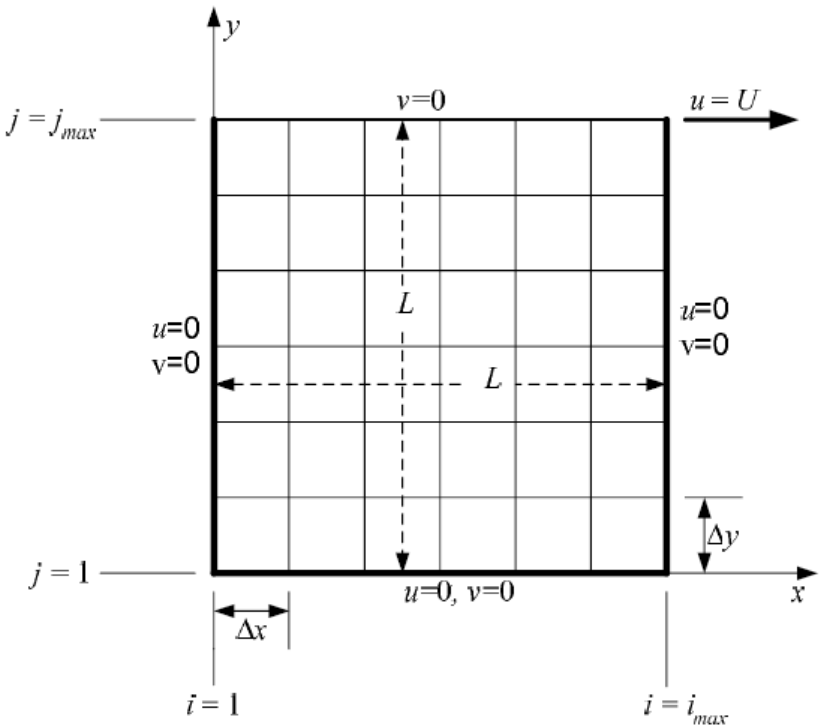



Figure: Streamlines Plot. Our Results vs K. Poochinapan [2] at Re 2000

## Unsteady State

The problem is to find the evolution of flow of a incompressible viscous fluid in a square domain where the top lid is moving with velocity U. The problem domain and boundary conditions are same as Steady State.




Lid-driven square cavity with grid, $Re = LU/\nu$

# Results

## Unsteady State

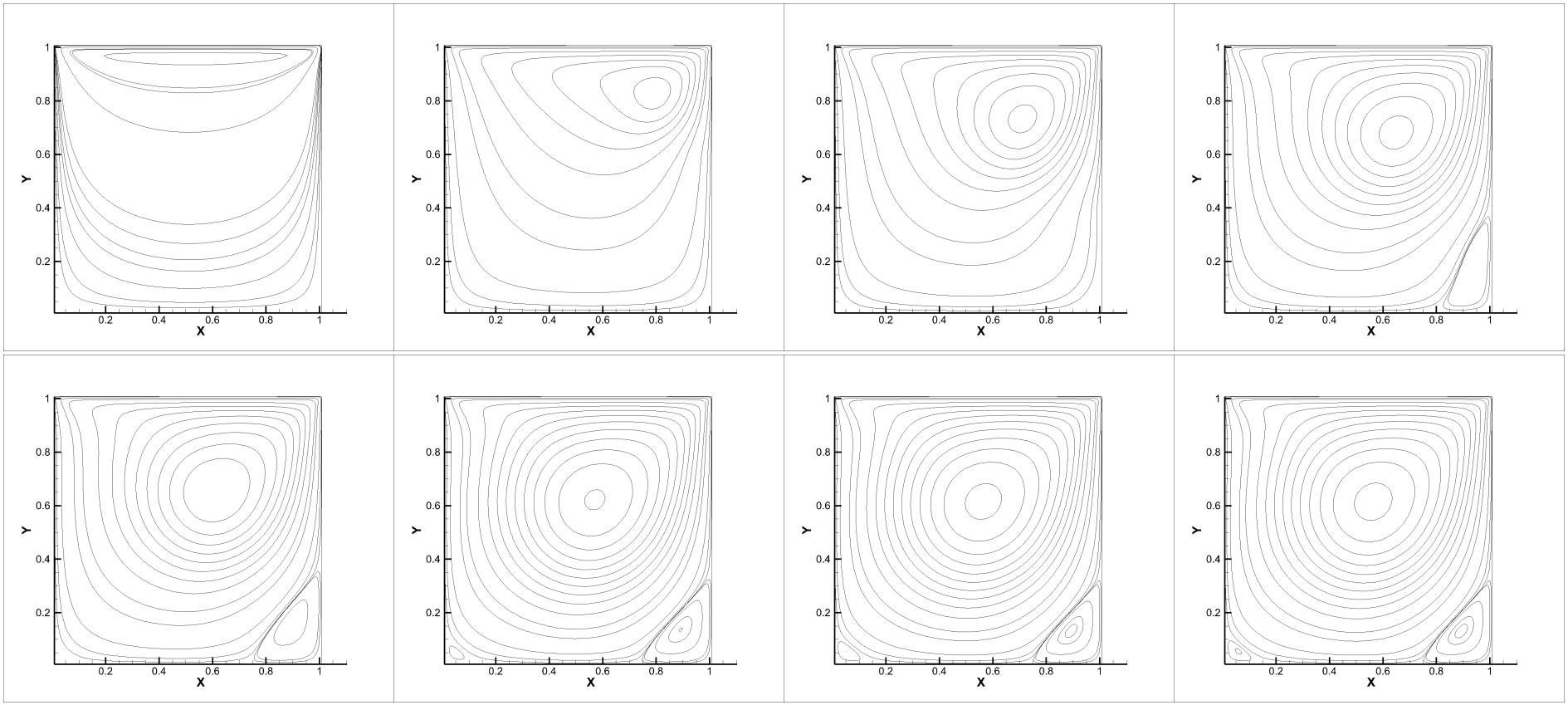



Figure: The evolution of stream function in unsteady state Lid-Driven cavity for Reynolds number 400.

## Staggered Cavity

The boundary conditions are same as Steady State's.

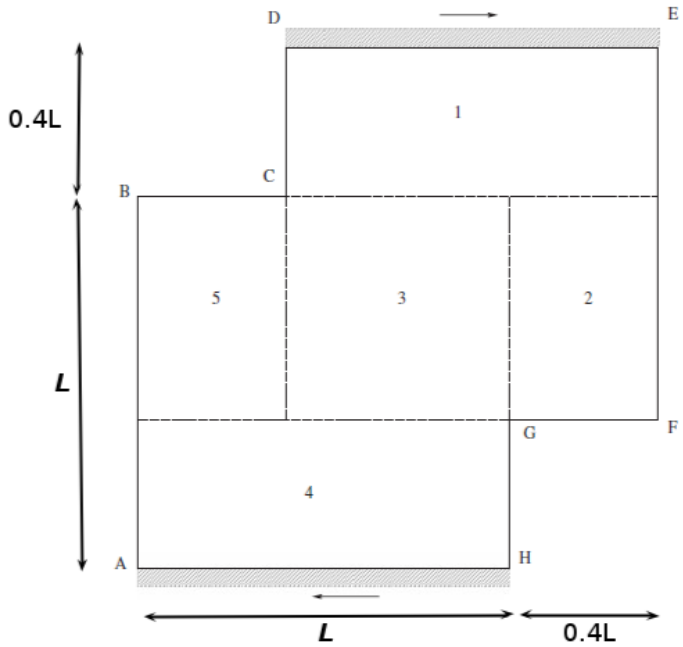


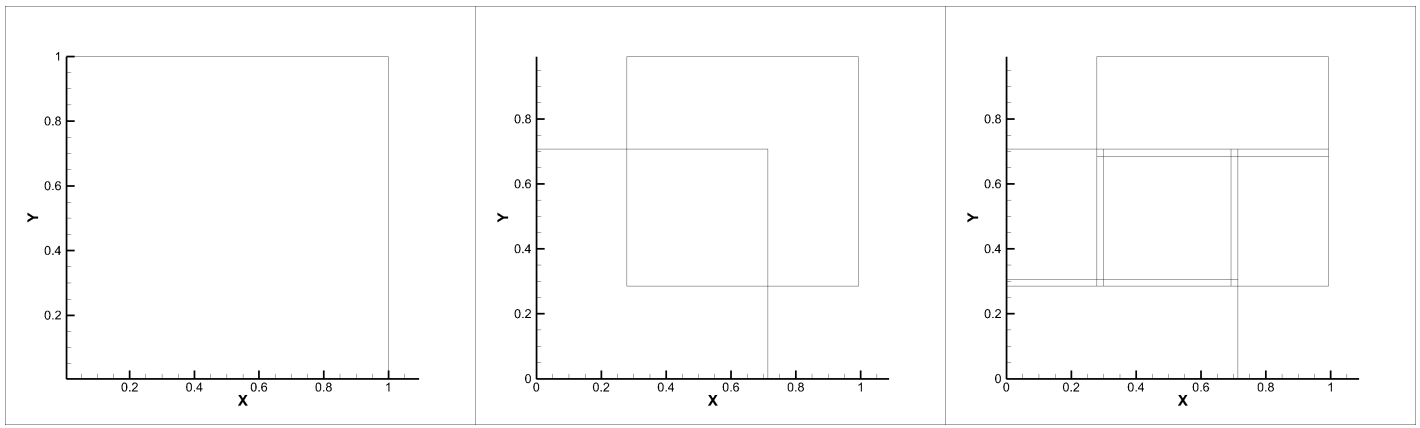

Figure: The problem domain for staggered cavity.

# Results

## Staggered Cavity

This problem is difficult to solve because of the blocked domains at the upper left and lower right corner. To overcome this problem we had three options.

- Force zero entries in the omega and psi at block domains after every iteration.
- Divide the domain into two smaller sub-domains and solve these domains individually.
- Divide the domain into five smaller sub-domains and solve these domains individually.



# Results

## Staggered Cavity

Pseudo-Code for solving Staggered Cavity,

```
function Solve(BLOCK){
        Iterate psi using Gauss-Seidel
        Solve omega using TDMA
        Calculate u and v
}
do{
        SetBoundaryConditions
        Solve(BLOCK1)
        Solve(BLOCK2)
        Solve(BLOCK3)
        Solve(BLOCK4)
        Solve(BLOCK5)
        Calculate error
}while error>10^-5
```

# Results

## Staggered Cavity

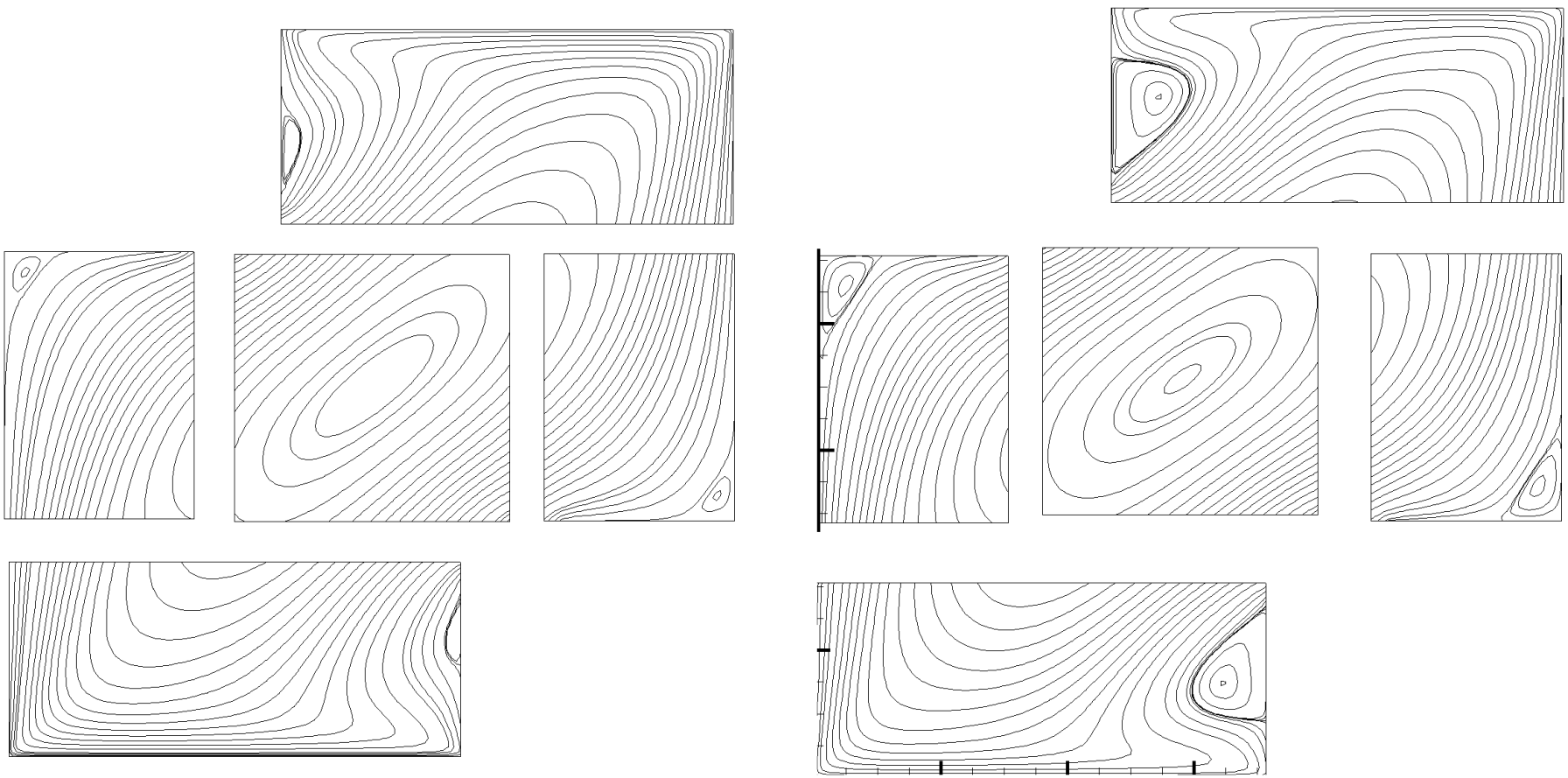


Figure: Streamlines calculated by diving domains into 5 subdomains at Re 400 and Re 800

# Results

## Staggered Cavity

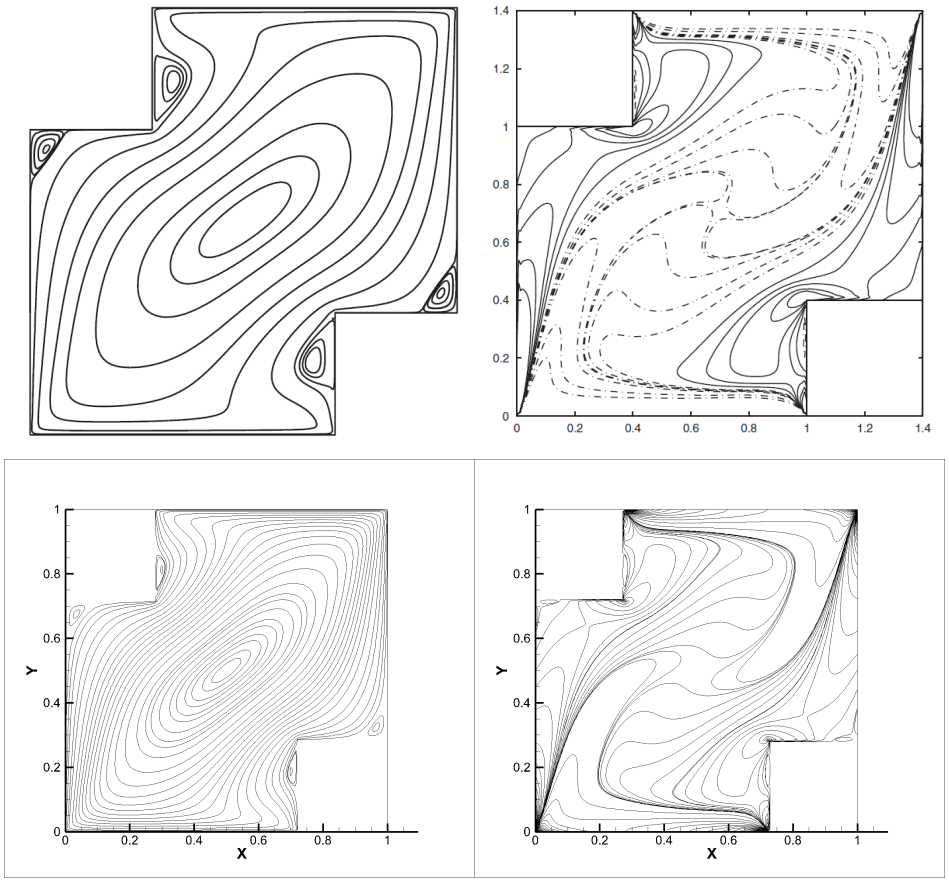



Figure: Streamlines and Vorticity plot in Staggered Cavity for Re 400. Results from Results from Zhou, et al [4] and Our results

# Results

## Staggered Cavity

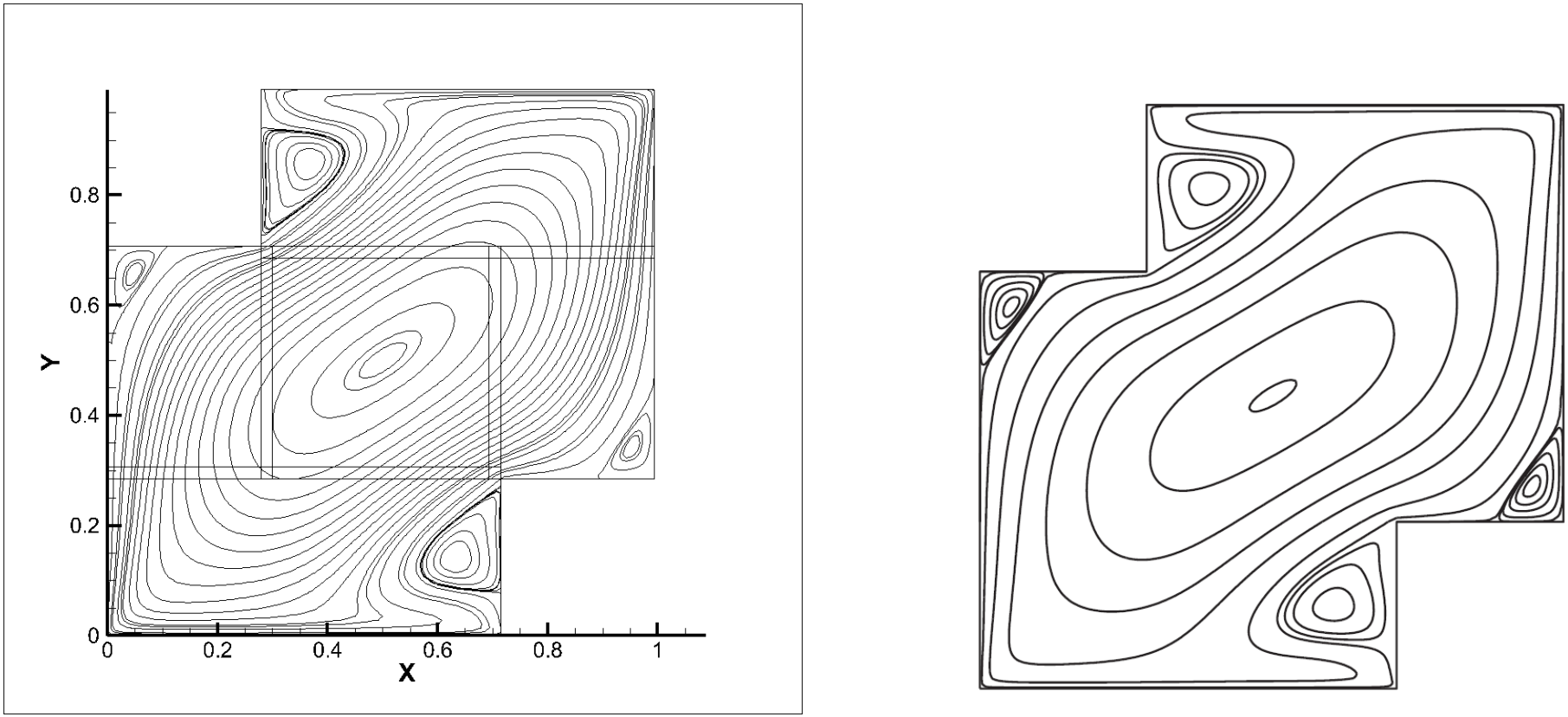



Figure: Streamlines plot at Re 1000. Our results vs Results from Zhou et al [4]

## Cross Cavity

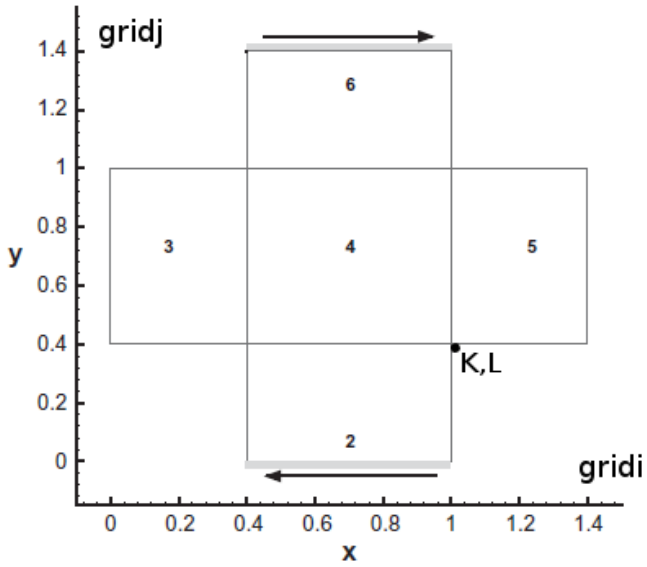



Figure: Problem domain of Cross Cavity.

Boundary conditions used to solve this domain is as follows,

$$LeftWall: \quad \omega_{0,j} = \frac{v_{0+1,j} - v_{0,j}}{\Delta x}$$

$$\omega_{gridi-k,j} = \frac{v_{gridi-k+1,j} - v_{gridi-k,j}}{\Delta x}$$

$$RightWall: \quad \omega_{k-1,j} = \frac{v_{k-1,j} - v_{k-2,j}}{\Delta x}$$

$$\omega_{gridi-1,j} = \frac{v_{gridi-1+1,j} - v_{gridi-2,j}}{\Delta x}$$

$$BottomWall: \quad \omega_{i,0} = -\frac{u_{i,1} - u_{i,0}}{\Delta y}$$

$$\omega_{i,l} = -\frac{u_{i,l+1} - u_{i,l}}{\Delta y}$$

$$TopWall: \quad \omega_{i,gridj-l-1} = -\frac{u_{i,gridj-l-1} - u_{i,gridj-l-2}}{\Delta y}$$

$$\omega_{i,gridj-1} = -\frac{u_{i,gridj-1} - u_{i,gridj-2}}{\Delta y}$$

# Results

## Cross Cavity

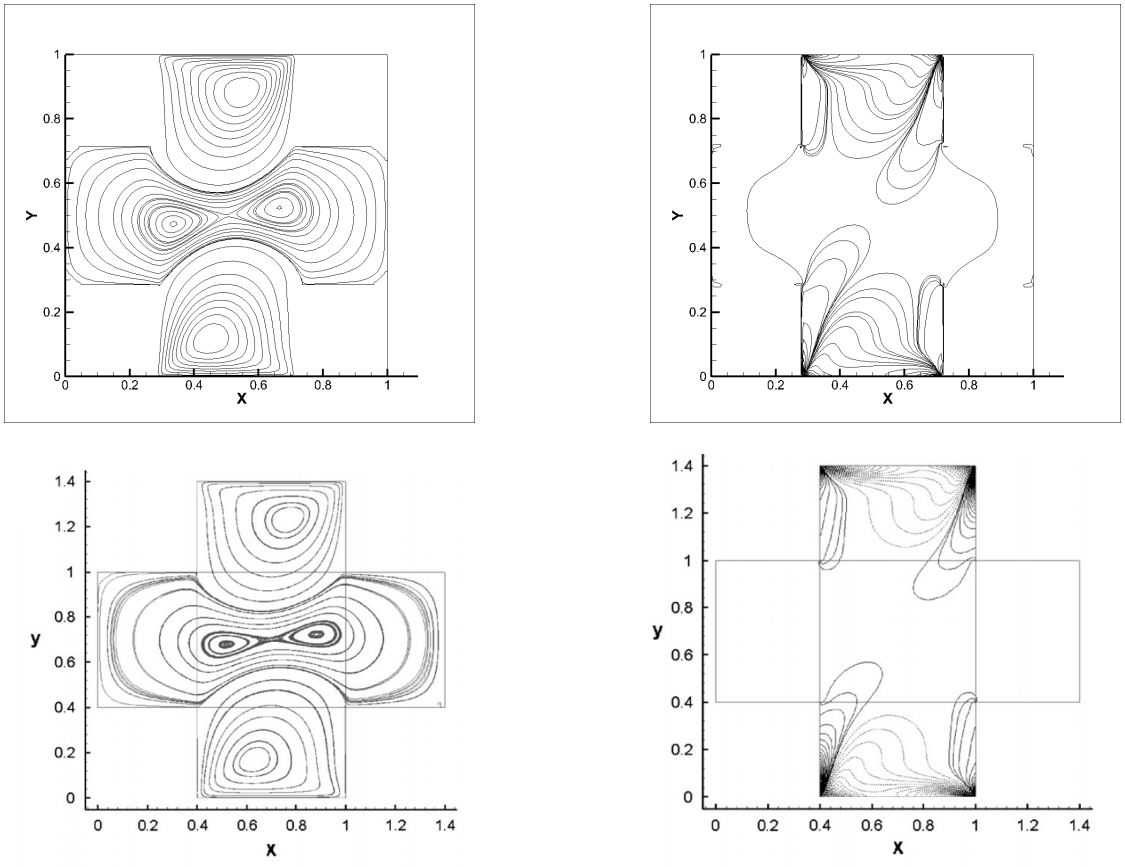



Figure: Stream Lines and Vorticity plots in Cross Cavity. Our Results vs Results from De Vicente et al [5]

## Backward Facing Step Flow

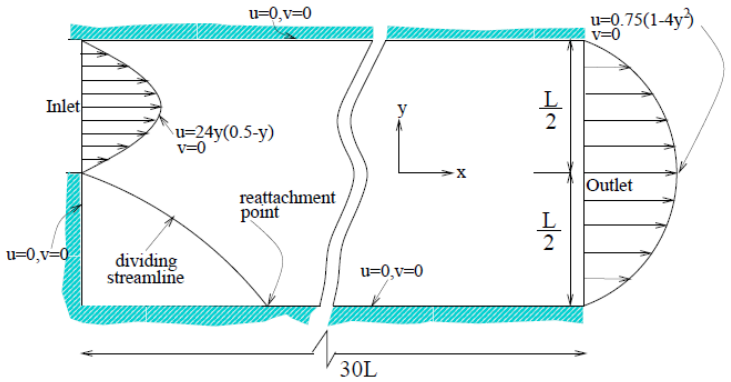



Figure: Problem domain of the Backward step flow.

In this problem fluid enters the domain with the parabolic profile into a larger cross-section area tube. The outlet conditions correspond to fully developed flow.

These Boundary conditions are as follows,

$$
\begin{aligned}
TOP &: u = v = \omega = 0; \psi = 0.5 \\
BOTTOM &: u = v = \psi = \omega = 0 \\
INLET &: u = v = \psi = \omega = 0 \\
&\quad at -0.5 \leq y \leq 0 \\
&\quad u = 12y(1-2y), v = 0, \psi = 2y^2(3-4y), \omega = 12(4y-1) \\
&\quad at 0 \leq y \leq 0.5 \\
OUTLET &: u = \frac{3}{4}(1-4y^2), v = 0, \psi = \frac{1}{4}(1+3y-4y^3), \omega = 6y
\end{aligned}
$$

# Results

## Backward Facing Step Flow

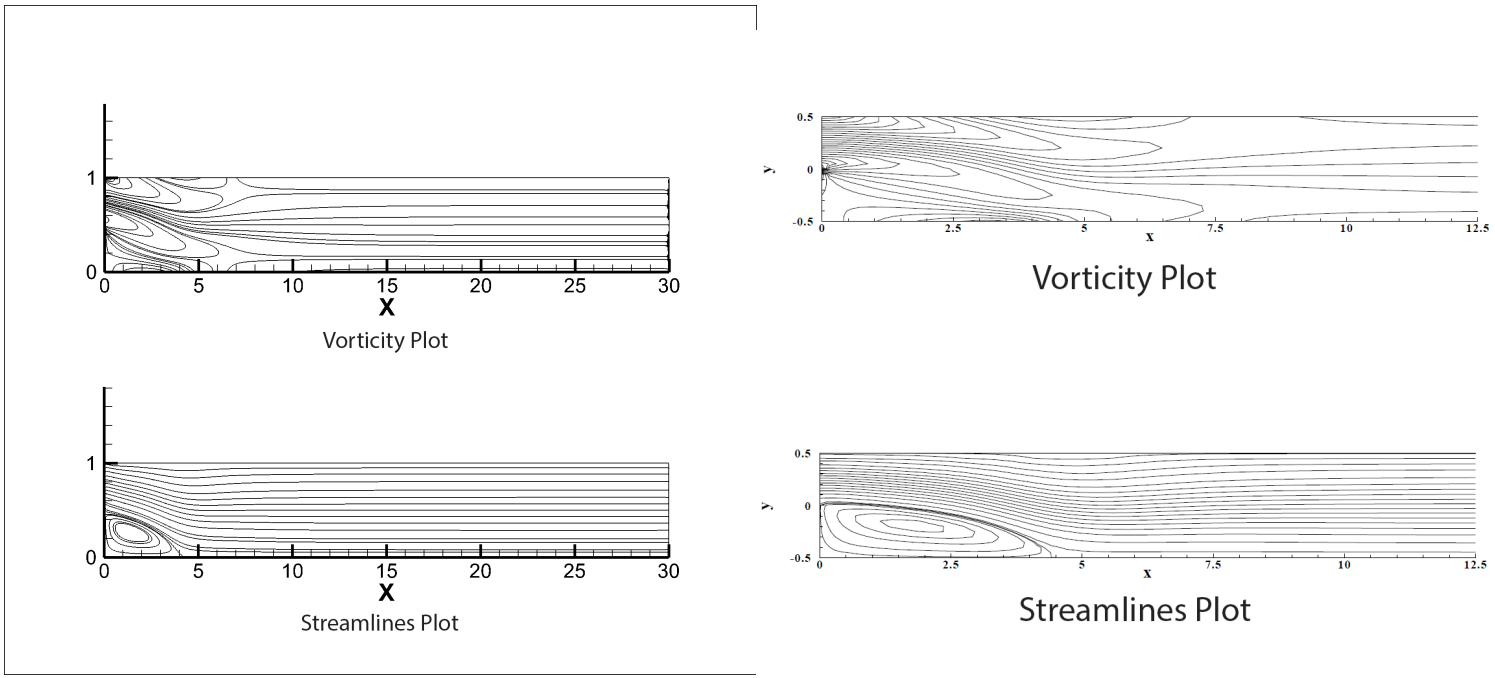



Figure: Streamlines and Vorticity Plot. Our Results vs Dr. Jiten Kalita's thesis [3] at Re 400

# Results

## Backward Facing Step Flow

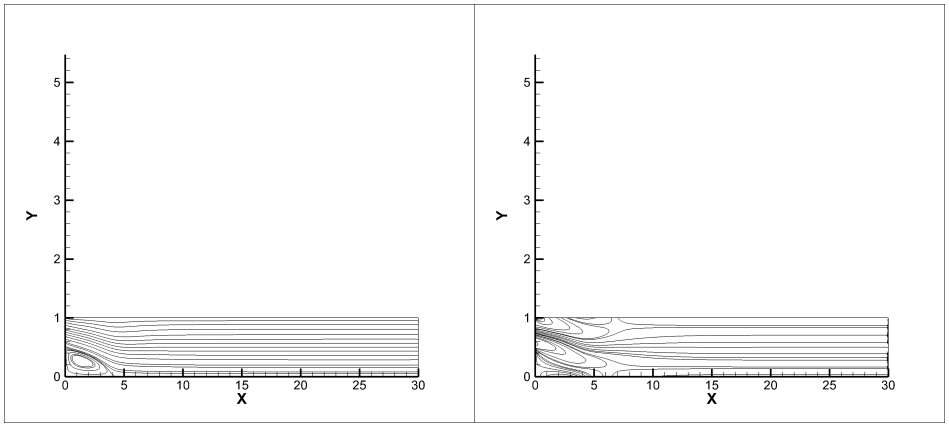



Figure: Plots at Reynolds number 400. Stream Lines and Vorticity plots calculated.

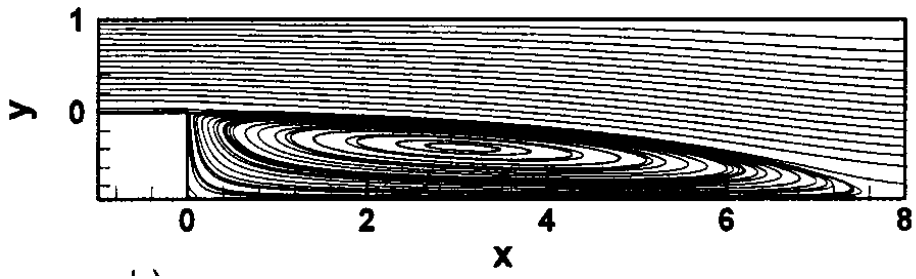



Figure: Stream Lines plot from G.Biswas et. al.

# Results

## Backward Facing Step Flow

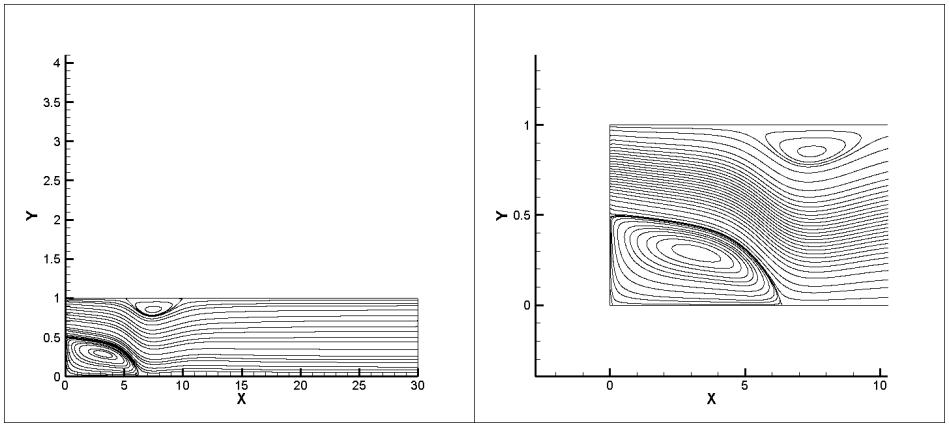



Figure: Stream Lines and Vorticity plots at Reynolds number 800 on grid size 150x4500. Expansion ratio H/h = 0.15.

The above results were obtained at Reynolds number 800 using grid size 150x4500. The code was simulated till error 0.001 on CUDA program.

# Results

## Backward Facing Step Flow

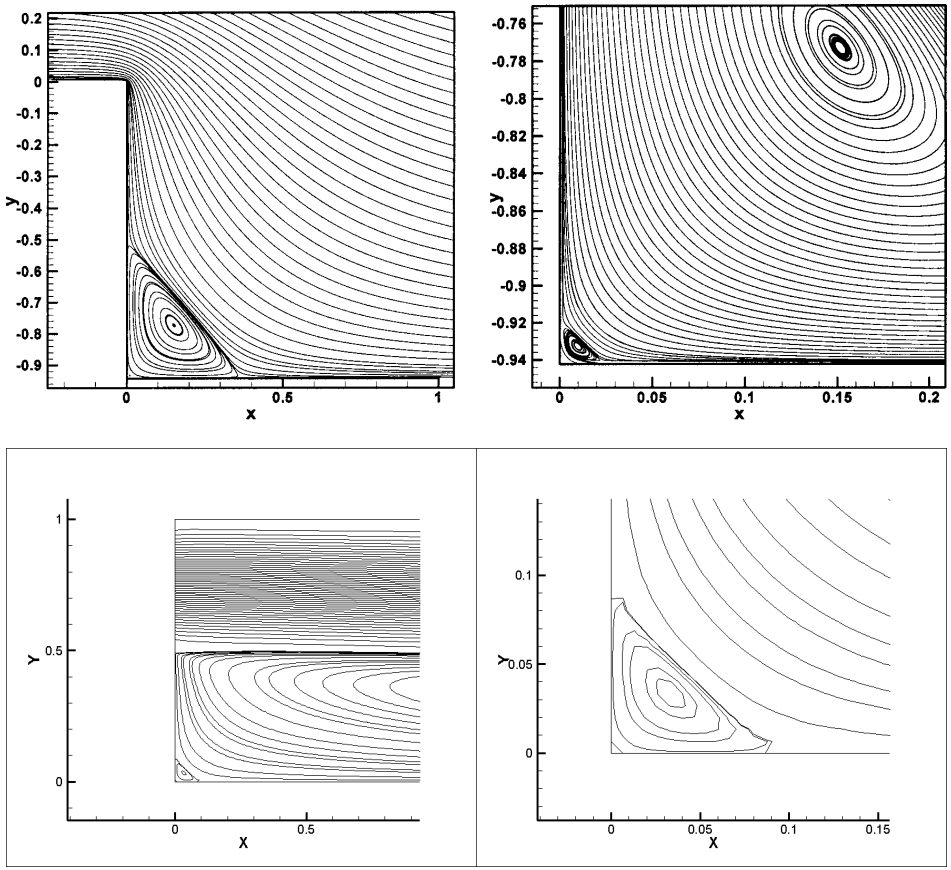



Figure: a).Stream Lines and its further zoom. Plots at Re=1 from G. Biswas et.al. b).Plots calculated at Re=800.

# Conclusion

- CUDA allowed to go to higher grid size which was previously not possible with serial code.
- CUDA computation is alot faster and efficient than average serial codes for simulation and other renderings hence it would be very useful in Direct Numerical Simulation.

# Bibliography

[1] Ghia, Urmila, Kirti N. Ghia, and C. T. Shin. "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method." Journal of computational physics 48.3 (1982): 387-411.

[2] Poochinapan, Kanyuta. "Numerical Implementations for 2D Lid-Driven Cavity Flow in Stream Function Formulation." ISRN Applied Mathematics 2012 (2012).

[3] Dr. Jiten C. Kalita, "HOC Schemes for incompressible viscous flows: Application and Development". Department of Mathematics, Indian Institute of Technology Guwahati (2002).

[4] Zhou, Y. C., et al. "DSC solution for flow in a staggered double lid driven cavity." International Journal for Numerical Methods in Engineering 57.2 (2003): 211-234.

# Bibliography

[5] De Vicente, Javier, et al. "Stability analysis in spanwise-periodic double-sided lid- driven cavity flows with complex cross-sectional profiles." Computers & Fluids 43.1 (2011): 143-153.

[6] G Biswas, M Breuer, and F Durst. Backward-facing step flows for various expansion ratios at low and moderate reynolds numbers. Journal of fluids engineering, 126(3):362374, 2004.

[7] Thibault, Julien C., and Inanc Senocak. "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows." Proceedings of the 47th AIAA Aerospace Sciences Meeting. 2009.