Fluid simulation with SIMPLE method using graphic processors

Ivan Dević
Supervisor: dr. sc. Dejan Vinković

Split, March 2014
Master Thesis in Physics

Department of Physics
Faculty of Natural Sciences and Mathematics
University of Split

Abstract

In this thesis we explore how application of graphics processors can accelerate calculations in fluid dynamics. We derive semi-implicit pressure linked equations (SIMPLE) and present SIMPLE method (algorithm) which is used with a great success in calculation of steady flows. Motivation for using graphics processors (GPUs) comes from their ability to significantly shorten execution time of fluid flow calculations. Our implementation of the SIMPLE algorithm on GPUs proved to be faster than CPU code by a factor larger then ten. We test our GPU code in two common problems in fluid dynamics: driven lid cavity and backwards facing step. Implementation of external forces on the fluid will also be implemented via plasma actuators, which have been explored as a tool for reducing turbulences and noise made by interaction between airplane wing and fluid. Our numerical results show that plasma actuators pull the fluid streams closer to the surface of the backwards facing step, as the size of the fluid vortex is reduced.

Contents

List of figures and diagrams

1. INTRODUCTION

Fluid dynamics is a branch of physics that studies fluid in motion. While basic equations, Navier-Stokes equation, are know for more then 150 years, new computational methods of solving them are still being developed today. Throughout this thesis we will demonstrate possibilites of *semi-implicit pressure linked equation* method, or simply called SIMPLE method (algorithm). SIMPLE method is model in which we can solve velocity fields for steady flows in incompressible fluids. Main idea of SIMPLE method is to utilize artificial mathematical approach, where we use pressure field as a correction tool in calculation of velocity fields.

Solving Navier-Stokes equation via SIMPLE method needs a lot of computing power on *central processor units* (CPUs), so we will implement SIMPLE method using *graphical processor units* (GPUs), via CUDA, programming extension to C/C++. Possibility of parallel calculation over large number of data elements is being recognized by more scientific researches every day, since some calculation can be accelerated by factor 20. We will apply implemented algorithm on two common problems in fluid dynamics : *driven-lid cavity* and *backwards facing step*, as we will also try to implement calculation of one of newest tecnologies in aerodynamics, plasma actuators.

Throughout this thesis, we will introduce mathematical background to SIMPLE method, introduce CUDA and show implementation used in this thesis. Mathematical background was derived by Andersson[1], so we will summarize and review his work and comments.

2. DERIVATION OF EQUATIONS IN *SIMPLE* METHOD

2.1.Finite control volume

Let us investigate flow in a cube shaped control volume *V*, bounded by control surface *S*, through which there is a finite volume flux of fluid. The control volume may be fixed in space with the fluid moving through it (Eulerian specification of the flow field) or the control volume may be defined as a "tracker" of a particular domain of fluid (Lagrangian specification of the flow field). Furthermore, let us apply fundamental physical principles to the control volume. Because of this, we do not need to observe the fluid as a whole, but with the control volume model we only observe the fluid in the control volume itself and it's interactions with neighboring control volumes. [1] The fluid equations that we obtain by applying the fundamental physical principles to the control volume are in integral form, which can be rewritten in the form of partial differential equations. The equations obtained from the Eulerian specification of the flow field are called the conservation form of the governing equations, while equations obtained from the Lagrangian specification of the flow field, are called the nonconservation form of the governing equations.

Let us imagine an infinitesimally small fluid element in the flow with a differential volume *dV.* We must keep in mind that although this volume is infinitesimally small, it is large enough to contain a huge number of molecules so that it can be viewed as a continuous medium. Inside this theoretical framework, we are in position to define interactions between control volumes, while neglecting intermolecular interactions, which happened on the scale much smaller than our computational resolution.

As already stated, instead observing the whole flow field at once, the fundamental physical principles are applied to just the infinitesimally small fluid element itself. [1]

2.2 The momentum equation

We will apply one of fundamental physical principles to a model of the flow-Newton's second law. The resulting equation is called the *momentum equation*. We will utilize the Lagrangian specification of the flow field, because this model is convinient for derivation of *momentum equation*. We must keep in mind that the *momentum equation* can also be derived from other models of control volume. [1]

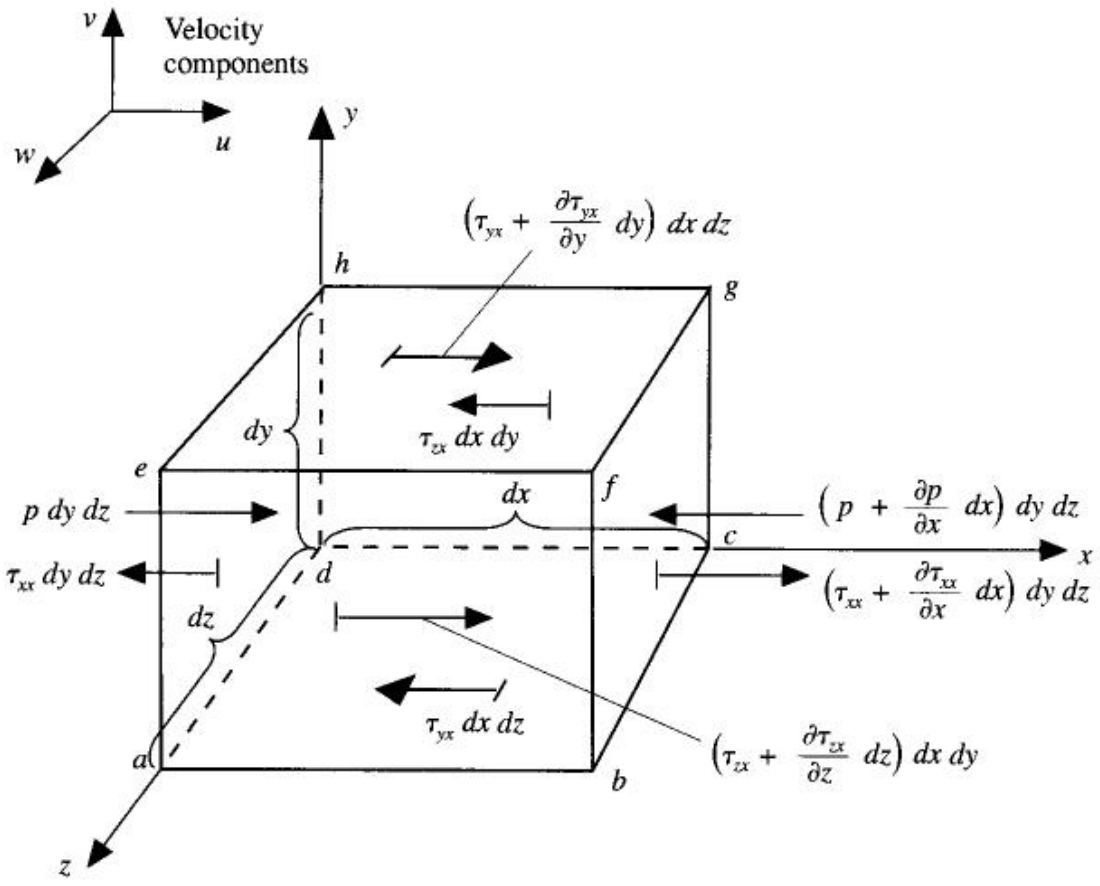

Figure 2.1 Sketch of surface force applied to finite control volume. Taken from [1]

Newton's second law, when applied to the moving fluid element in Fig. 2.1, says that the net force on the fluid element equals its mass times the acceleration of

the element. Since this is a vector relation we can split it into three scalar relations along the *x*, *y*, and *z* axes. Let us consider only the *x* component of Newton's second law, where we will denote *x* component of force as F_X and *x*-component of acceleration as a_X.

First, let's define two types of force, classified by their source:

- *Body forces*; force applied to the content of the control volume. These forces "act at a distance"; examples are gravitational, electric, and magnetic forces. [1]
- *Surface forces*; forces exhibited on the control surface of the fluid element. There are two surface forces we consider; the pressure distribution acting on the surface, imposed by the outside fluid surrounding the fluid element and the shear and normal stress distributions acting on the surface.

Let us denote body force per mass unit acting on fluid as *f*, with its components f_x, f_y and f_z. The surface forces in the x-direction exerted on the fluid element are sketched in Fig. 2.1. The convention will be used in fashion that τ_{ij} denotes a stress in the *j* direction exerted on a plane perpendicular to the *i* axis.

We will denote components of velocity as *u* (x component), *v* (y component) and *w* (z component).

Taking in consideration all mentioned above, we can finally write an equation for total force in *x* direction

$$F_x = \left[-\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z}\right] dxdydz + \rho f_x dxdydz \qquad (2.1)$$

where ρ denotes density of fluid. Density of fluid multiplied by *x*, *y* and *z* differentials equals mass of fluid element, so combining Eq. (2.1) with most common writing of second Newton's law, we obtain

$$\rho \frac{Du}{Dt} = - - \frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \tag{2.2}$$

which is the *x* component of the *momentum equation*. In similar fashion we obtain *y* component and *z* component of the *momentum equation.*

$$\rho \frac{Dv}{Dt} = - \frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + \rho f_y \tag{2.3}$$

$$\rho \frac{Dw}{Dt} = - \frac{\partial p}{\partial z} + \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} + \rho f_z \tag{2.4}$$

Note that inside these partial differential equations, we have defined change of the momentum of the control volume *V*, solely by interactions with other control volumes. Since this fluid element is moving with the flow, Eqs. (2.2) to (2.4) are in nonconservation form. They are scalar equations and are called the Navier-Stokes equations after M. Navier and G. Stokes.

Isaac Newton stated that shear stress in a fluid is proportional to velocity gradients.[1] Such fluids are called Newtonian fluids. Throughout this thesis, the fluid is assumed to be Newtonian. For such fluids, we can write our sheer stresses

$$\tau_{xx} = \lambda(\nabla V) + 2\mu \frac{\partial u}{\partial x} \tag{2.5}$$

$$\tau_{yy} = \lambda(\nabla V) + 2\mu \frac{\partial v}{\partial y} \tag{2.6}$$

$$\tau_{zz} = \lambda(\nabla V) + 2\mu\frac{\partial w}{\partial z} \tag{2.7}$$

$$\tau_{xy} = \tau_{yx} = 2\mu(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}) \tag{2.8}$$

$$\tau_{xz} = \tau_{zx} = 2\mu(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}) \tag{2.9}$$

$$\tau_{yz} = \tau_{yz} = 2\mu(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}) \tag{2.10}$$

where μ is the molecular viscosity coefficient and $\boldsymbol{\lambda}$ is the second viscosity coefficient. $\boldsymbol{V}$ denotes total field of velocity.

2.3 Incompressible Navier-Stokes equation

The incompressible Navier-Stokes equations can be obtained from the compressible form, by setting density equal to a constant. Considering a continuity equation for fluids

$$\frac{D\rho}{Dt} + \rho\nabla\boldsymbol{V} = 0 \tag{2.11}$$

and considering that density is constant, we obtain continuity equation for incompressible fluids

$$\nabla\boldsymbol{V} = 0 \tag{2.12}$$

Using Eqs. (2.2)-(2.10) with Eq. (2.12), we obtain incompressible momentum equation for each componoent: *x*, *y* and *z*.

$$\rho \frac{Du}{Dt} = -\frac{\partial p}{\partial x} + 2\mu \frac{\partial^2 u}{\partial x^2} + \mu \frac{\partial}{\partial y}\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) + \mu \frac{\partial}{\partial z}\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) + \rho f_x \quad (2.13)$$

$$\rho \frac{Dv}{Dt} = -\frac{\partial p}{\partial y} + 2\mu \frac{\partial^2 v}{\partial y^2} + \mu \frac{\partial}{\partial x}\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) + \mu \frac{\partial}{\partial z}\left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right) + \rho f_y \quad (2.14)$$

$$\rho \frac{Dw}{Dt} = -\frac{\partial p}{\partial z} + 2\mu \frac{\partial^2 w}{\partial z^2} + \mu \frac{\partial}{\partial x}\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) + \mu \frac{\partial}{\partial y}\left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right) + \rho f_z \quad (2.15)$$

Using Eq. (2.12) we can further simplify Eqs. (2.13)-(2.15). The idea is to rewrite differential of one component as function of other two components, and after some rearranging, rewrite second term on right side of these equations, we obtain *incompressible Navier stokes* equations

$$\nabla \boldsymbol{V} = 0 \quad (2.16)$$

$$\rho \frac{Du}{Dt} = -\frac{\partial p}{\partial x} + \mu \nabla^2 u + \rho f_x \quad (2.17)$$

$$\rho \frac{Dv}{Dt} = -\frac{\partial p}{\partial y} + \mu \nabla^2 v + \rho f_y \quad (2.18)$$

$$\rho \frac{Dw}{Dt} = -\frac{\partial p}{\partial z} + \mu \nabla^2 w + \rho f_z \quad (2.19)$$

Note that in Eqs. (2.16)-(2.17) we have four variables in four equations, which implies that with these four equations, we close our problem, from mathematical standpoint of view.

Although, incompressible Navier-Stokes equations are derived from the compressible Navier-Stokes equations, we cannot use the same numerical technique for solving both sets of equations. From now on, we deal with the implementation of the *pressure correction* algorithm, which will be explained in great detail.

2.4 Staggered grid

If we write Eq. (2.16) using central spatial differentials, we obtain

$$\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}+\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y}=0 \tag{2.20}$$

There is a big problem solving Eq.(2.20), because it allows checkerboard distribution of velocities. Same problem occurs in rewriting pressure gradient in the same fashion as we did with velocities in Eq.(2.20). In Figure 2.2 we see one of the checkerboard distributions that would satisfy Eq.(2.20), which obviously cannot be steady solution for any of scalar fields.[1]

	50	100	50	100	50	100	50
	10	30	10	30	10	30	10
	50	100	50	100	50	100	50
	10	30	10	30	10	30	10
	50	100	50	100	50	100	50
	10	30	10	30	10	30	10
	50	100	50	100	50	100	50
	10	30	10	30	10	30	10

Figure 2.2 Checker board distribution which would be achievable with non-staggered grid (Source [8])

To avoid this problem, we will use staggered grid, shown in Figure 2.3.

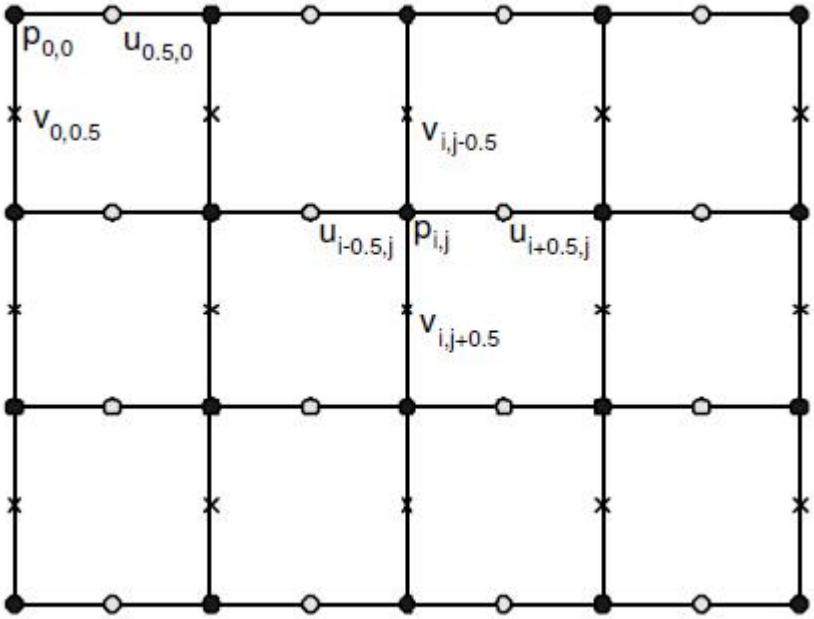

Figure 2.3 Staggered grid used in SIMPLE method. Taken from [4]

In this fashion we will avoid checkerboard distribution, due to using values of other fields, which are not in same grid points (namely velocity and pressure field). In compressible fluids, this problem is avoided by calculating densities, while in incompressible fluids density is set to a constant. Eq.(2.20) rewritten for grid shown in Figure 2.3 is

$$\frac{u_{i+1/2,j}-u_{i-1/2,j}}{\Delta x}+\frac{v_{i,j+1/2}-v_{i,j-1/2}}{\Delta y}=0 \qquad (2.21)$$

2.5 Pressure correction method

Pressure correction method is iterative process (without time interpetation), that follows next few steps to obtain results.

- Guess the initial pressure field, which we will denote as p^*.
- With use of momentum equations, obtain values of u^*, v^* and w^*.

- Since values of u^*, v^* and w^* were calculated from guessed pressure field, we must obtain new pressure field, which we will denote as p'. This field is *pressure correction* field. Once we add this field to initially guessed field, we solve the continuity equation

$$p = p' + p^* \tag{2.22}$$

- In the same fashion as for pressure, we obtain the velocity correction field and sum them as (2.22)

$$u = u' + u' \tag{2.23}$$

$$v = v' + v^* \tag{2.24}$$

$$w = w' + w^* \tag{2.25}$$

- In next iteration, we use values of *p*, *u*, *v* and *w* as *p**, *u**, *v** and *w**.

From now on, we will derive expressions for two-dimensional problem. We will derive equations for *x*-component, since *y* component equations are obtained in the same fashion and are completely the same when we change spatial differentials. All derivations untill now were easier using noncoservation equations, but from now on we will use conservation form of the *momentum equations*, because they are easier to implement in numerics. They are obtained from nonconservation equations once we rewrite time differential (left side of the *momentum equation*) as substantial derivative. Those equations in incompressible form (without external force) are

$$\frac{\partial \rho u}{\partial t} + \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} = -\frac{\partial p}{\partial x} + \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{2.26}$$

$$\frac{\partial \rho v}{\partial t} + \frac{\partial \rho v^2}{\partial y} + \frac{\partial \rho uv}{\partial x} = -\frac{\partial p}{\partial y} + \mu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{2.27}$$

Now, let us consider how can we solve Eq. (2.26), when we apply forward differential for time, and central differential for spatial derivatives. If we recall Figure 2.3 and focus at grid point (*i+1/2,j*), we can see that for central differential in third term on left side of Eq. (2.26), we must somehow obtain values of *v* in middle grid boxes. We will use simple linear interpolation to do that. The same fashion applies to the neccesity of having values of *u* when solving Eq. (2.27). We can now rewrite Eq. (2.26)

$$(\rho u)_{i+1/2.j}^{n+1} = (\rho u)_{i+1/2.j}^{n} + A\Delta t + \frac{\Delta t}{\Delta x}\left(p_{i+1,j}^{n} - p_{i,j}^{n}\right) \qquad (2.28)$$

where A is given by expression

$$A = -\left[\frac{(\rho u^2)_{i+3/2,j}^{n} - (\rho u^2)_{i-1/2,j}^{n}}{2\Delta x} + \frac{(\rho u\bar{v})_{i+1/2,j+1}^{n} - (\rho u\bar{v})_{i+1/2,j-1}^{n}}{2\Delta y}\right] + \mu\left[\frac{u_{i+3/2,j}^{n} - 2u_{i+1/2,j}^{n} + u_{i-1/2,j}^{n}}{(\Delta x)^2} + \frac{u_{i+1/2,j+1}^{n} - 2u_{i+1/2,j}^{n} + u_{i-1/2,j-1}^{n}}{(\Delta y)^2}\right]$$

$\bar{v}$ represents the interpolated value of *v* inside box.

As mentioned, at the end of each iteration, we set results of velocity and pressure field, as guessed result in next time iteration. That means we can rewrite Eq.(2.28)

$$(\rho u^*)_{i+1/2.j}^{n+1} = (\rho u^*)_{i+1/2.j}^{n} + A^*\Delta t + \frac{\Delta t}{\Delta x}\left(p^{*n}_{i+1,j} - p^{*n}_{i,j}\right) \qquad (2.29)$$

If we subtract Eq. (2.29) from Eq. (2.28), we obtain

$$(\rho u')_{i+1/2.j}^{n+1} = (\rho u')_{i+1/2.j}^{n} + A'\Delta t + \frac{\Delta t}{\Delta x}\left(p'^{n}_{i+1,j} - p'^{n}_{i,j}\right) \qquad (2.30)$$

using Eqs.(2.22)-(2.24).

We will obtain a formula for the pressure correction *p'* by demanding that all velocity fields must satisfy the incompressible continuity equation. The pressure correction method is an iterative approach, but results during the iterative process do not have physical meaning until the last step, as the iteration is not performed along time. Values for *p'* provide us with converged solution once *d* is reasonably small and it gives us corrected pressure, which will ultimately give us velocity fields which satisfy the incompressible continuity equation. To summarize, values of *p'* are a numerical artifact which helps us obtain the steady solution for the flow field.

With that in mind, we set *A'* and $(\rho u)^n$ zero and rewrite Eq. (2.30) as

$$(\rho u')_{i+1/2.j}^{n+1} = \frac{\Delta t}{\Delta x}\left(p'^{n}_{i+1,j} - p'^{n}_{i,j}\right) \tag{2.31}$$

If we combine Eq. (2.23) and Eq.(2.31), we obtain

$$(\rho u)_{i+1/2.j}^{n+1} = (\rho u^*)_{i+1/2.j}^{n+1} + \frac{\Delta t}{\Delta x}\left(p'^{n}_{i+1,j} - p'^{n}_{i,j}\right) \tag{2.32}$$

Equation for *y* component can be obtained in same fashion.

$$(\rho v)_{i.j+1/2}^{n+1} = (\rho v^*)_{i.j+1/2}^{n+1} + \frac{\Delta t}{\Delta x}\left(p'^{n}_{i,j+1} - p'^{n}_{i,j}\right) \tag{2.33}$$

Now we can finally obtain pressure correction formula, by combining Eqs.(2.32)-(2.33) into Eq.(2.21), we get

$$ap'_{i,j} + bp'_{i+1,j} + bp'_{i-1,j} + cp'_{i,j+1} + cp'_{i,j-1} + d = 0 \tag{2.34}$$

where

$$a = 2\left[\frac{\Delta t}{(\Delta x)^2} + \frac{\Delta t}{(\Delta y)^2}\right]$$

$$b = -\frac{\Delta t}{(\Delta x)^2}$$

$$c = -\frac{\Delta t}{(\Delta y)^2}$$

$$d = \frac{1}{\Delta x}\left((\rho u^*)_{i+1/2,j} - (\rho u^*)_{i-1/2,j}\right) + \frac{1}{\Delta y}\left((\rho v^*)_{i,j+1/2} - (\rho v^*)_{i,j-1/2}\right)$$

Eq. (2.34) is called *pressure correction formula*. During the process of calculation velocity fields *u* and *v* do not satisfy continuity equation. Once velocities sastisfy continuity equation *d* will be equal zero, but it is computationally expensive to reach that degree of convergence, so we aim to get as close to zero value as possible in reasonable computational time. We can also see that Eq. (2.34) is central differential representation of Poisson equation, which explains the elliptic behavior of the *pressure correction* formula.

3. GRAPHICS PROCESSOR UNIT AND CUDA

3.1 Introduction

As science advances, so are calculation used in science getting more complicated and need for higher computing performance is higher then ever. To adress this problem, graphic processor units (GPU) have been used to perform calculation. With multiple cores and very high memory bandwidth, todays GPUs offer strong resources for both graphics and non-graphics prosessing.

As GPUs main objective is graphics rendering, it was designed for highly parallel and intensive computation. Distribution of transistors in CPUs and GPUs is shown in Figure 3.1, where ALU stands for arithmetig-logic unit, which are transistors used in calculations.

Figure 3.1 Transistor distribution in CPU and GPU. Taken from [5]

GPUs are well-suited to adress problems that can be expressed as data-parallel computations as same program can be executed over many data elements in parallel. Strong advantage of CPUs over GPUs is memory transfer and *flow* control in programs, so the best way to use both of technologies is via *heterogeneous*

programming or hybrid programming. Idea is to let GPUs handle arithmetic and logic tasks, while minimizing flow control and memory transfer which is done by CPUs.

3.2 CUDA

CUDA [5] stands for *Compute Unified Device Architecture* and is used as software architecture for issuing and managing computations on the GPUs as data-parallel computing device. It was designed as an extension to C and C++ programming language. CUDA also provides ability to read and write data at any location in DRAM (*dynamic random-access memory*), just like on a CPU.

When programmed through CUDA, GPUs are used in identically as CPUs, but due to different transistor distribution, it can handle many operations in the same time. In further text, we will refer to CPUs as a *host*, while we will call GPUs *device.* What we want to achieve is to let *host* deal with the majority of code, while *device* while be used only for calculating arithmetical steps of the code. Both *host* and *device* maintain their own DRAM, referred to as *host memory* and *device memory*. Execution of *kernel* is organized as a grid of thread blocks, as shown in Figure 3.2.

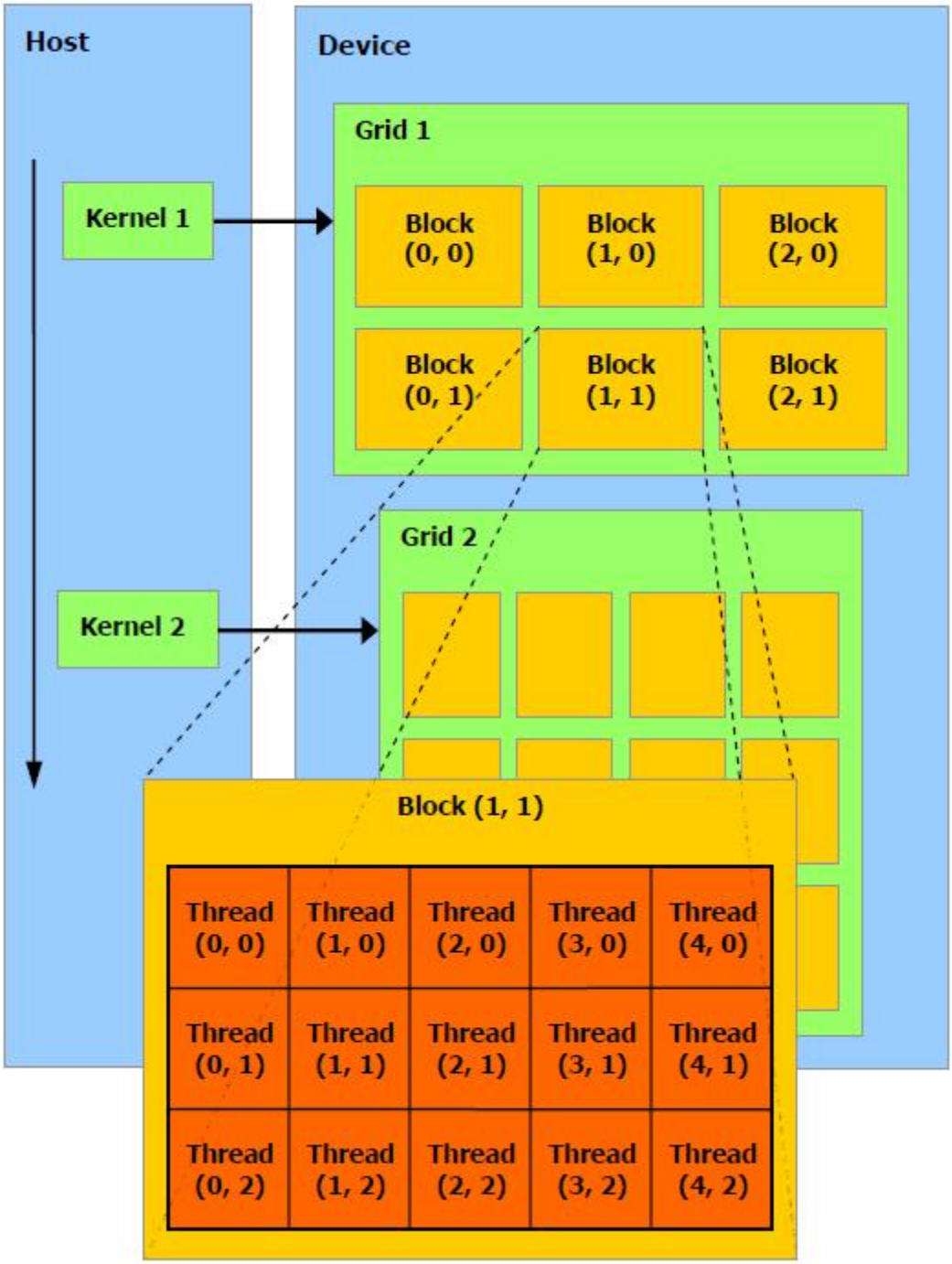

Figure 3.2 Visualization of grid hierarchy in *kernel*. Taken from [5]

A thread block is a collection of threads that can cooperate together by sharing data through some fast shared memory and synchronizing their exectuion to coordinate memory accesses. Each thread is identified by its *thread ID*, which is the thread number inside it's own block. Blocks can be organized as one-, two- or

three-dimensional array of threads, where threads are then identified as one-, two- or three-component index.

There is a limited maximum number of threads that a block can contain, but then *kernel* can be used to execute itself over grid of blocks, so total number of threads in execution can be larger than maximum size of a block. As thread, each block is identified by its *block ID*, which is the block number within the grid. Grids can be organized as blocks, hence grid can have a two-dimensional array of blocks, while only new GPUs can have three-dimensional array of blocks.

3.3 Memory model and memory types

A thread that execute on the *device* has only access to the *device's* DRAM and on-chip memory. GPUs have few memory types and memory model which defines communication between memory of different types. Memory types are classified as [5]:

- **Global memory** – is the largest memory space on the device and is used for storage of data, once they are downloaded from *host*. Speed of accesing global memory is slower then is with shared memory (explained below), so it's optimal to use global memory only for data transaction between *host* and *device*. It's lifetime is application execution time
- **Local memory** – is memory that resides on device, as global memory. Local memory is a part of global memory that is created for a single thread, so it has lifetime of thread execution.
- **Shared memory** – is on-chip memory and because of that is much faster than local or global memory. Shared memory is created for every block, hence

block can only access data from it's respective shared memory. It has a lifetime of kernel execution.

- **Constant memory** – is also on-chip memory and is equally fast as shared memory, but *device* can only read this data, while *host* can change it. Unlike shared memory, constant memory can be accessed by whole grid and has lifetime of application execution time.
- **Registers** – are memory specific for each processor, so that each processor has right to write and read fomt its own registers. Register is accessable by particular thread and it's lifetime is kernel execution time.

With all this in mind, we can already get an idea how to approach memory on GPUs. Idea is to use global memory for transaction of data between *host* and *device*, but execute calculation on shared memory. Figure 3.3 shows summary of memory types and their communication.

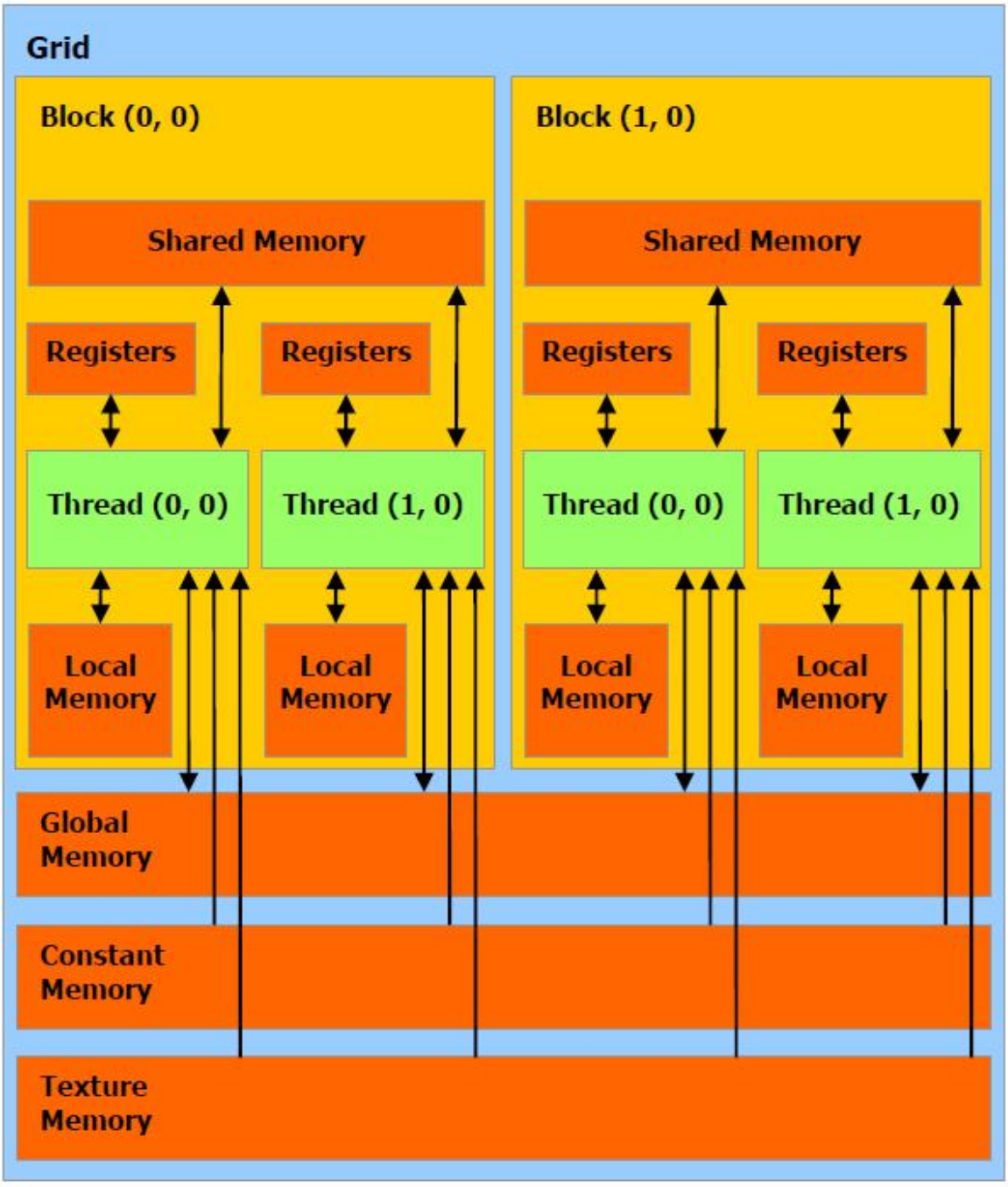

Figure 3.3 Memory model of GPU. Taken from [5]

3.4 Execution configuration

Any invocation of a *kernel* must be specified with the *execution configuration*, which is done with *dim3* variables. *Dim3* variables is three-component vector which components can be accessed as *variable_name.x* (*y or z*). Execution configuration is

specified in form *<<<Dg, Db, Ns>>>* between function name and argument list, where (definitions are copied from [5]):

- **Dg** is of type *dim3* and specifies the dimension and size of the grid, such that **Dg.x * Dg.y** eqauls the number of blocks being lunched
- **Db** is of type *dim3* and specifies the dimension and size of each block such that **Db.x * Db.y * Db.z** equals the number of threads per block
- **Ns** specifies the number of bytes in shared memory that is danymically allocated per block for this call in adition to the statically allocated memory. It is an optional argument which defaults to 0.

Kernels are declared with argument *__global__ void* in front of function name and argument list, so function declared as:

- *__global__ void Function (type of argument* argument)*

must be invoked as

- *Function <<<**Dg**, **Db**, **Ns** >>> (argument)*

4. SIMPLE ALGORITHM

4.1 Summary of equations

In this chapter, we will present numerical procedure for solving equations derived in second chapter of this thesis. We will also introduce *Reynolds number* (Re), with which we will parameterize density ρ and molecular viscosity coefficient μ. *Reynolds number* will be also used for scaling space and time parameters. From now on we will denote partial derivatives as subsribts. Let us summarize equations we are solving in two-dimensional form:

$$u_t = -u_x^2 - (uv)_y - p_x + \frac{1}{Re}\left(u_{xx} + v_{yy}\right) + f_x \tag{4.1}$$

$$u_t = -v_y^2 - (uv)_x - p_y + \frac{1}{Re}\left(u_{xx} + v_{yy}\right) + f_y \tag{4.2}$$

$$u_x + v_y = 0 \tag{4.3}$$

$$\nabla^2 p = \nabla V \tag{4.4}$$

Eq. (4.4) is the same equation as Eq. (2.34). It may be confusing considering the right side should be zero, once aproximation of incompressibility is taken into consideration, but velocities which will be input inside Eq. (4.4) don't satisfy continuity equation, so we are obtaining their corrections while solving Eq. (4.4).

If there are no forces applied to fluid, values of f_x and f_y are zero.

In Table 1. we will summarize all the differentials that were used to solve Eqs. (4.1)-(4.4). If we recall from Chapter 2., we used forward differential for time derivatives and central differentials for spatial derivatives.

Table 1 Summary of partial derivative and it's respective differentials

Partial derivative	Differentials
u_t	$\frac{u^{n+1}-u^n}{\Delta t}$
u_{xx}	$\frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{(\Delta x)^2}$
u_x^2	$\frac{u_{i+1,j}^2-u_{i,j}^2}{2\Delta x}$
p_x	$\frac{p_{i+1,j}-p_{i,j}}{\Delta x}$
p_y	$\frac{p_{i,j+1}-p_{i,j}}{\Delta y}$

Equations which will be iterated and solved using GPUs are:

$$u_{i+0.5.j}^{n+1} = u_{i+0.5.j}^{n} + \Delta t\left(A - (\Delta x)^{-1}\left(p_{i+1,j}^{n} - p_{i,j}^{n}\right)\right) \tag{4.5}$$

$$A = a_1 + Re^{-1}(a_2 + a_3) \tag{4.6}$$

$$a_1 = -\frac{(u^2)_{i+1.5,j}^{n}-(u^2)_{i-0.5,j}^{n}}{2\Delta x} - \frac{(u\bar{v})_{i+0.5,j+1}^{n}-(u\bar{v})_{i+0.5,j-1}^{n}}{2\Delta y} \tag{4.7}$$

$$a_2 = -\frac{u_{i+1.5,j}^{n}-2u_{i+0.5,j}^{n}+u_{i-0.5,j}^{n}}{(\Delta x)^2} \tag{4.8}$$

$$a_3 = -\frac{u_{i+0.5,j+1}^{n}-2u_{i+0.5,j}^{n}+u_{i+0.5,j-1}^{n}}{(\Delta y)^2} \tag{4.9}$$

$$v_{i.j+0.5}^{n+1} = v_{i.j+0.5}^{n} + \Delta t\left(B - (\Delta y)^{-1}\left(p_{i,j+1}^{n} - p_{i,j}^{n}\right)\right) \tag{4.10}$$

$$B = b_1 + Re^{-1}(b_2 + b_3) \tag{4.11}$$

$$b_1 = -\frac{(v^2)^n_{i,j+1.5} - (u^2)^n_{i,j-0.5}}{2\Delta x} - \frac{(v\overline{u})^n_{i+1,j+0.5} - (v\overline{u})^n_{i-1,j+0.5}}{2\Delta y} \quad (4.12)$$

$$b_2 = -\frac{v^n_{i,j+1.5} - 2v^n_{i,j+0.5} + v^n_{i,j-0.5}}{(\Delta y)^2} \quad (4.13)$$

$$b_3 = -\frac{v^n_{i+1,j+0.5} - 2v^n_{i,j+0.5} + v^n_{i-1,j+0.5}}{(\Delta x)^2} \quad (4.14)$$

$$p'_{i,j} = -a^{-1}\left(b\left(p'_{i+1,j} + p'_{i-1,j}\right) + c\left(p'_{i,j+1} + p'_{i,j-1}\right) + d\right) \quad (4.15)$$

$$a = 2\Delta t\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right) \quad (4.16)$$

$$b = -\frac{\Delta t}{\Delta x^2} \quad (4.17)$$

$$c = -\frac{\Delta t}{\Delta y^2} \quad (4.18)$$

$$d = \frac{1}{\Delta x^2}\left(u_{i+0.5,j} - u_{i-0.5,j}\right) + \frac{1}{\Delta y^2}\left(v_{i,j+0.5} - v_{i,j-0.5}\right) \quad (4.19)$$

$$u' = -\frac{\Delta t}{\Delta x}\left(p'_{i+1,j} + p'_{i,j}\right) \quad (4.20)$$

$$v' = -\frac{\Delta t}{\Delta y}\left(p'_{i,j+1} + p'_{i,j}\right) \quad (4.21)$$

Alghorithm which will be used is:

1. Solve Eqs. (4.5)-(4.14)
2. Iterate Eqs. (4.15)-(4.19). Number of iterations depends on precision that we want to achieve. Lower number of iterations of this step may result in need for higher

number of iterations of whole algorithm. However, final results, if converged, must be the same. In this thesis, we iterated this step 30 times.

3. Using Eqs (4.20)-(4.21) find the correction of velocities.
4. Add valus of p' to original p
5. Repeat steps 1-4

4.2 Data transfer between global and shared memory on GPU

While discussing memory types CUDA can access, we mentioned that shared memory is fastest, together with constant memory which cannot be changed from *device*. Before solving any of Eqs. (4.5)-(4.22), we will copy all of our data to shared memory, where each member of data will represent one thread. As shared memory is unique to each block inside grid, we must be carefull of possible memory malfunction. Let's imagine we are working with two-dimensional square grid, with dimension 4.

BlockID (0,3)	BlockID (1,3)	BlockID (2,3)	BlockID (3,3)
BlockID (0,2)	BlockID (1,2)	BlockID (2,2)	BlockID (3,2)
BlockID (0,1)	BlockID (1,1)	BlockID (2,1)	BlockID (3,1)
BlockID (0,0)	BlockID (1,0)	BlockID (2,0)	BlockID (3,0)

Figure 4.1 Block identification system in two-dimensional grid

Lets focus on block with BlockID (1,1), and let's assume each block contains 2*2 threads. Considering Eqs.(4.5)-(4.22) we can see that each data member needs all adjacent data (first neighbors) to be calculated. To do so, we need to declare two-

dimensional array in shared memory for each block, but with dimension 4*4, because we need to copy adjacent rows and columns so calculation can be done. In Figure 4.2, grey squares will represent data that shared memory of BlockID(1,1) must contain (data/threads of non-adjacent blocks won't be shown), for calculation to be correct.

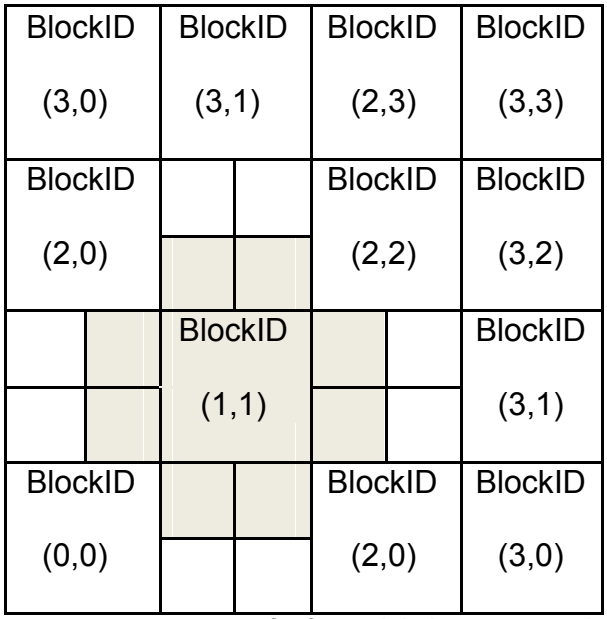

Figure 4.2 Example of necessary memory transfer from global memory to shared memory in SIMPLE algorithm

In this thesis, we are using one-dimensional arrays on *host* that are downloaded to *device* as one-dimensional arrays. With memory transfer to shared memory, we transform them in two-dimensional arrays for less complicated implementation of Eqs. (4.5)-(4.21). Reason for governing data as one-dimensional arrays on *host* and *device* is because of faster download and upload of data from *device*.

4.3 Visualization

For visualization of our data, we will use *Paraview*. *Paraview* is an open-source, multi-platform data analysis and visualization application. We will visualize

magnitudes of velocities and their respective components. One of the techinques of visualization which we will use is *streamline plot technique* and it is incorporated inside *Paraview*. *Streamline plot technique* is used for incompressible fluids, where we calculate *stream* function and with lines (called *streams*) we connect grid positions of same *stream* function value. It is defined as :

$$\Psi = \int_a^b (udy + vdx) \tag{4.22}$$

where shift from *a* to *b* is infinitesimal. *Streams* we get on the plot are good indication of fluid trajectory, vortexes, turbulencies etc. For visualiazing our data in *Paraview*, we must calculate, with linear interpolation, values of velocities in grid points (where we calculated pressure field) and then write them in *.vtk* file (this is shown in Appendix A in function *PRINT_VECT_VTK*).

4.4 Fluid mechanics problems

In this thesis, we are going to apply SIMPLE algorithm to two common fluid mechanics problems.

Driven-lid cavity is one of the most known problems in fluid mechanics and is widely used as test for new solutions in computational fluid dynamics. *Driven lid cavity* is two-dimensional closed box with one of lids moving (in this thesis it will be bottom lid), causing fluid inside the box to move. We will also use *driven-lid cavity* as test for our calculation and also for test of speed-up we achieved using CUDA and GPUs over CPUs.

Boundary conditions for *driven lid cavity* with dimensions [A*B]are:

- No-slip condition, hence velocities perpendicular to the normal vector of boundaries must be zero, with exeption on moving lid

  u(x,0)=1

  u(x,B)=0

  v(0,y)=0

  v(A,y)=0

- Directional derivatives of pressure on normal vectors of boundary wall must be zero:

  p(x,0)=p(x,1)

  p(x,B)=p(x,B-1)

  p(0,y)=p(1,y)

  p(B,y)=p(B-1,y)

- There is no momentum loss in fluid while reflecting from boundary. Considering staggered grid that is used, we can see that for reflection of fluid there is no true boundary condition, because last grid points of velocities perpendicular to the boundaries (only velocities that could reflect due to no-slip condition) are not on the wall. We obtain this boundary condition with linear interpolation, by setting coefficent of proportion between last grid point and his adjacent grid point to -2/3

  u(0.5,y)=-2*u(1.5,y)/3

  u(A-0.5,y)=-2*u(A-1.5,y)/3

  v(x,0.5)=-2*v(x,1.5)/3

  v(x,B-0.5)=-2*v(x,B-1.5)/3

Backwards facing step is another common problem in computational fluid dynamics. It consists of inflow on left side, outflows on top and right side and

boundary wall on the bottom side with backwards facing step. We will also implement calculation of body force in *backwards facing step*, which will be described later.

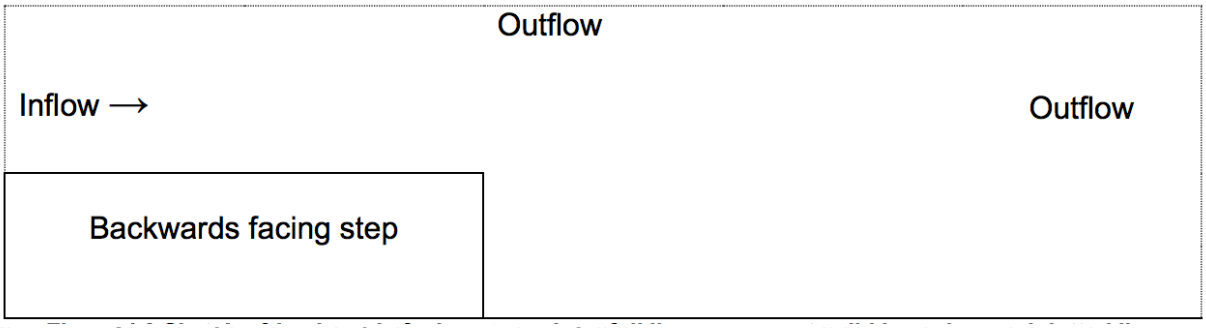

Figure 4.3 Sketch of *backwards facing step*, where full lines represent solid boundary and dotted line represents open boundary (no reflection of fluid on this boundary)

Boundary conditions for *backwards facing step* are same on the boundary wall, while inflow and outflows have own set of boundary conditions:

- We will set u=1, v=0 and p=0 on inflow boundary
- Outflow boundary conditions is zero value of directive derivatives of u, v perpendicular to boundary and value of p is zero.

Body force that we will apply in *backwards facing step* is force excelled by plasma actuator. Plasma actuator is technology used on airplanes to reduce turbulencies and noise made by plane [2]. Force components f_x and f_y are defined by next equations

$$f_x = A * \frac{F_x}{\sqrt{F_x^2+F_y^2}} exp\left(-\left(\frac{x-x_0-y+y_0}{y-y_0+y_b}\right)^2 - \beta_x(y-y_0)^2\right) \quad (4.23)$$

$$f_y = A * \frac{F_y}{\sqrt{F_x^2+F_y^2}} exp\left(-\left(\frac{x-x_0}{y-y_0+y_b}\right)^2 - \beta_y(y-y_0)^2\right) \quad (4.24)$$

where F_x=2.0, F_y=2.6, y_b=0.00333, β_x=72000 and β_y=900000 [2]. We will define strength of this repulsive force via parametar A.

Force described by Eqs. (4.24)-(4.25) is repulsive low range force and its magnitude is shown in Diagram 1.

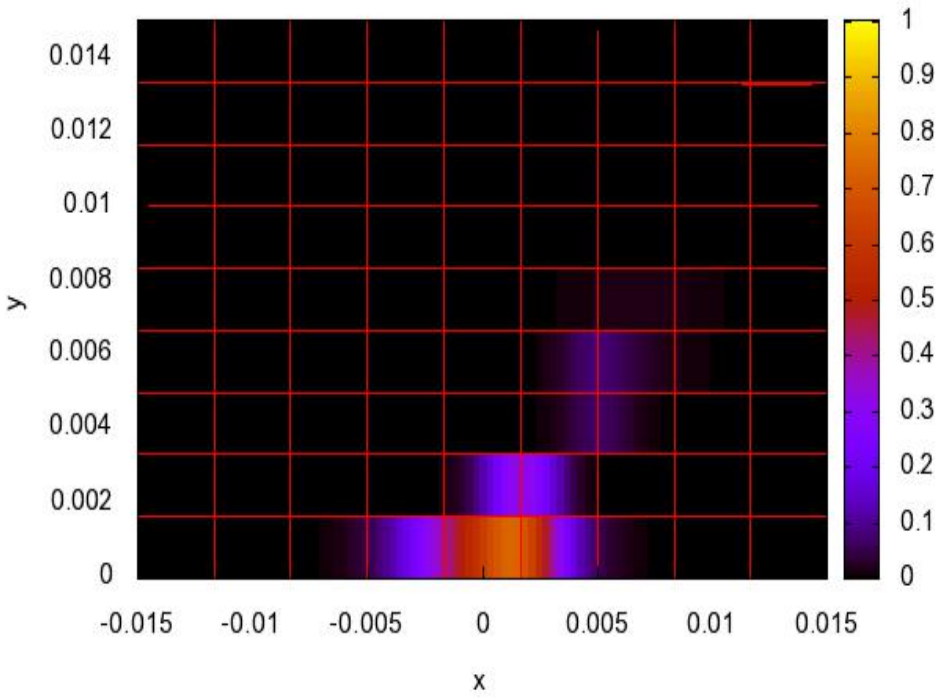

Diagram 1 Magnitude of plasma actuator body force, when plasma actuator is in the centre of coordinate sytem

5. RESULTS

5.1 Driven-lid cavity

Solutions for *driven –lid cavity* (velocity fields) are shown in Figure 5.1, 5.2 and 5.3, while Figure 5.4 represents results for *streamline plot technique.* Calculation was done in resolution 100*100 with Reynolds number set to 400.

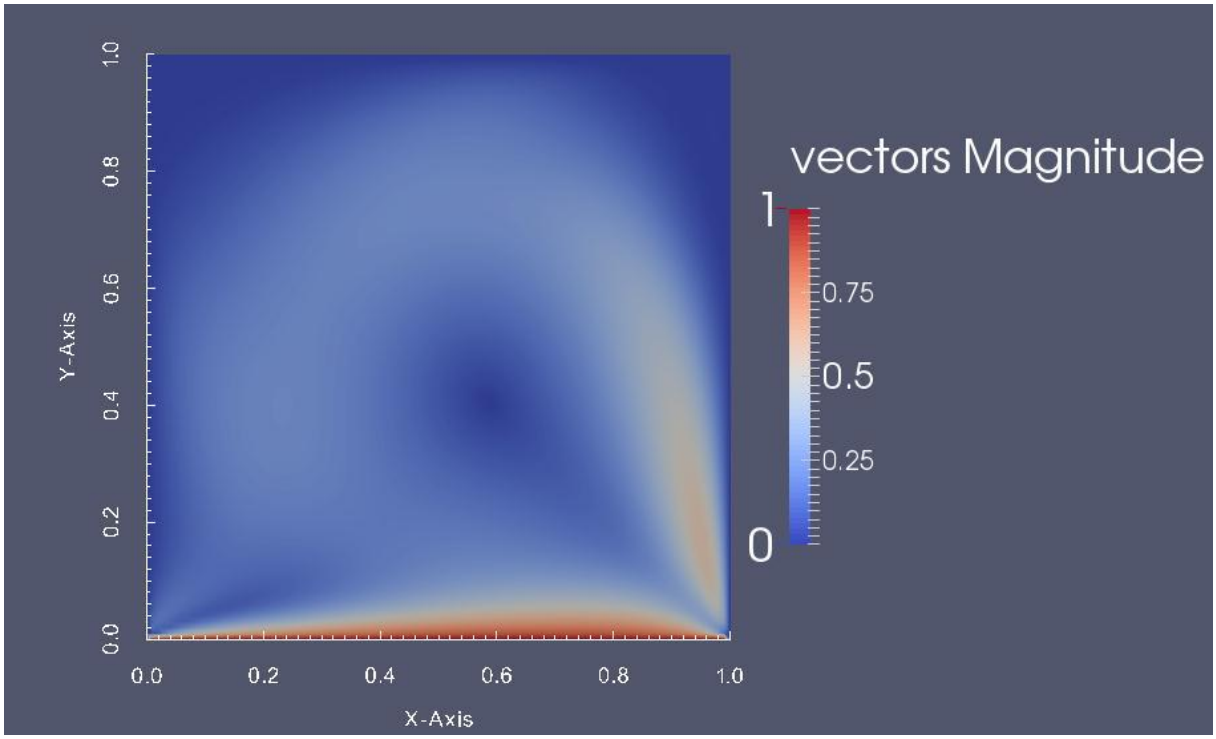

Figure 5.1 Solution of total velocity magnitudes in *Driven-lid cavity*

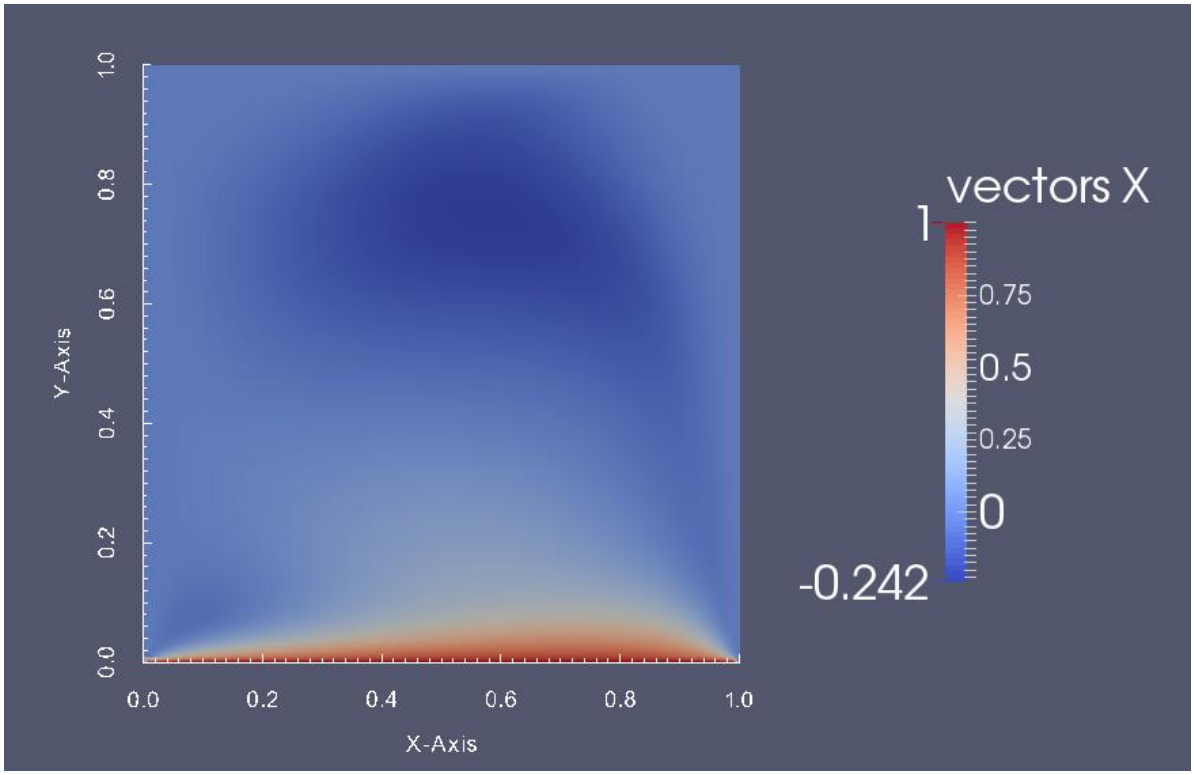

Figure 5.2 Solution of x-component velocitiy magnitudes in *Driven-lid cavity*

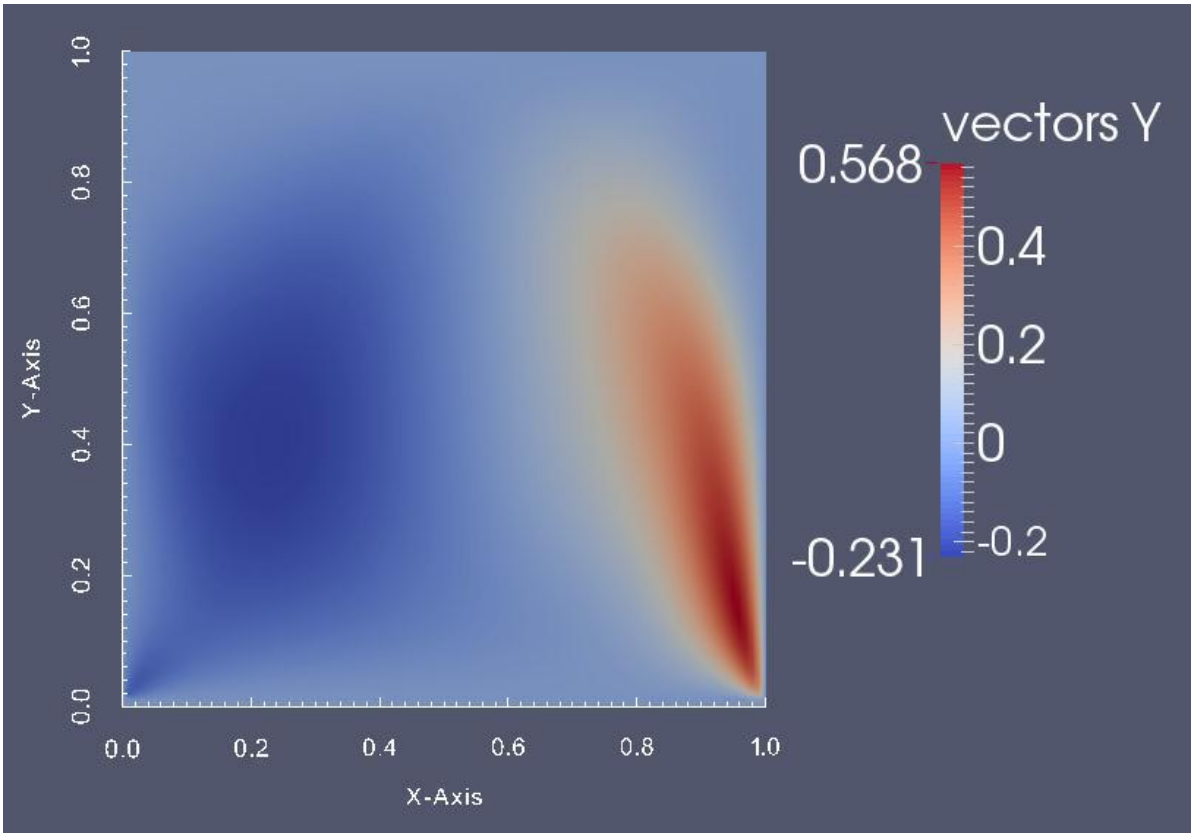

Figure 5.3 Solution of y-component velocitiy magnitudes in *Driven-lid cavity*

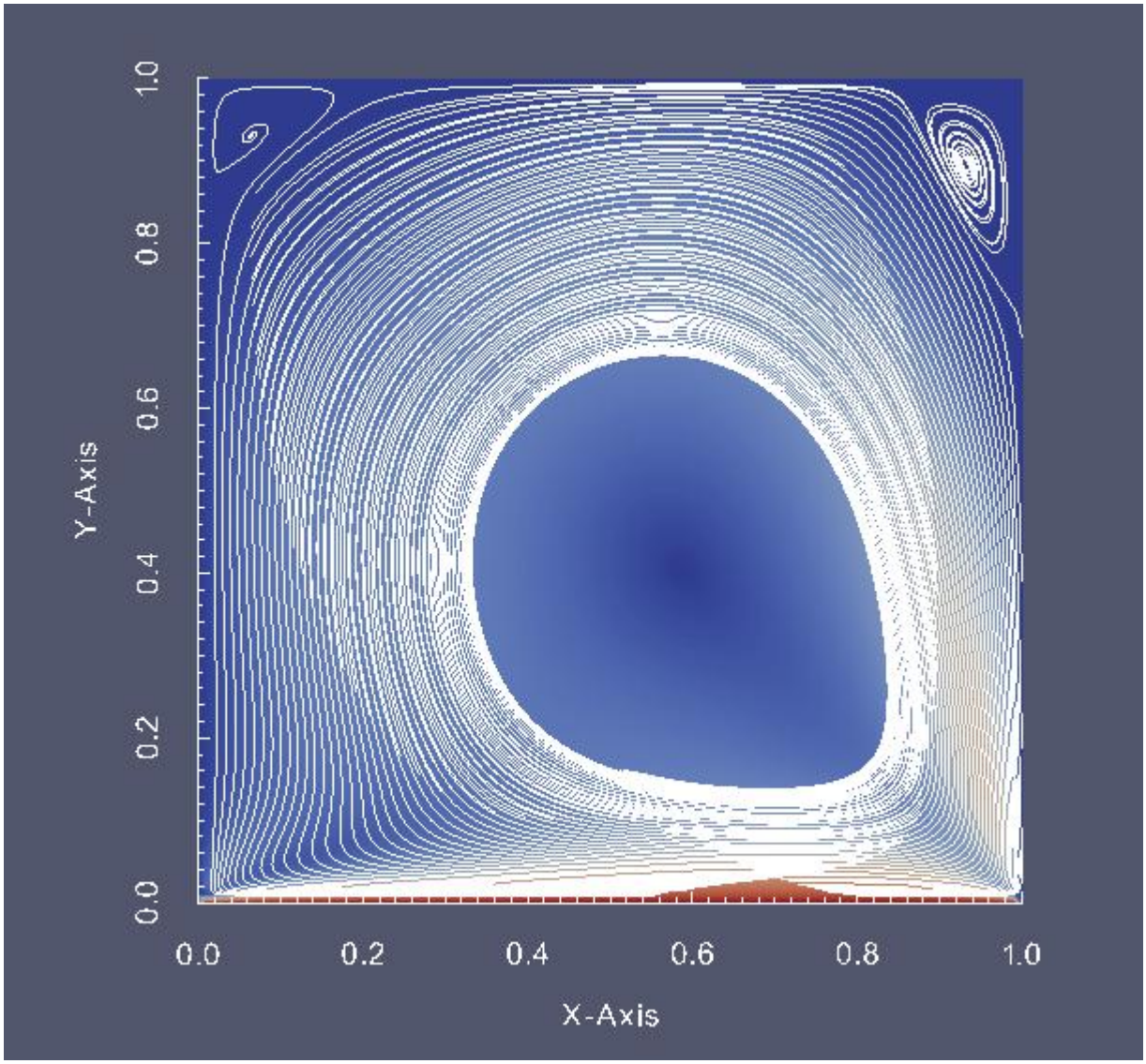

Figure 5.4 Visualization via *streamline plot technique* for *Driven-lid cavity*

This results confirm SIMPLE algorithm as valid computational method for solving incompressible Navier-Stokes equation. Also *stremline plot technique* shows two vortexes that are characteristic for square *driven-lid cavity* problem. One issue with using *Paraview* as *streamline plot technique* is that it calculates stream function in direction of one line, so full display with streams is hardly achievable. Now that our results are postivie, lets see the acceleration that CUDA has provided us and what size of block is most optimal for this computation. We will test accelartion on 5000

iterations, since number of iterations doesn't affect acceleration. We will define accelartion is result of dividing speeds achieved by CPU and GPU.

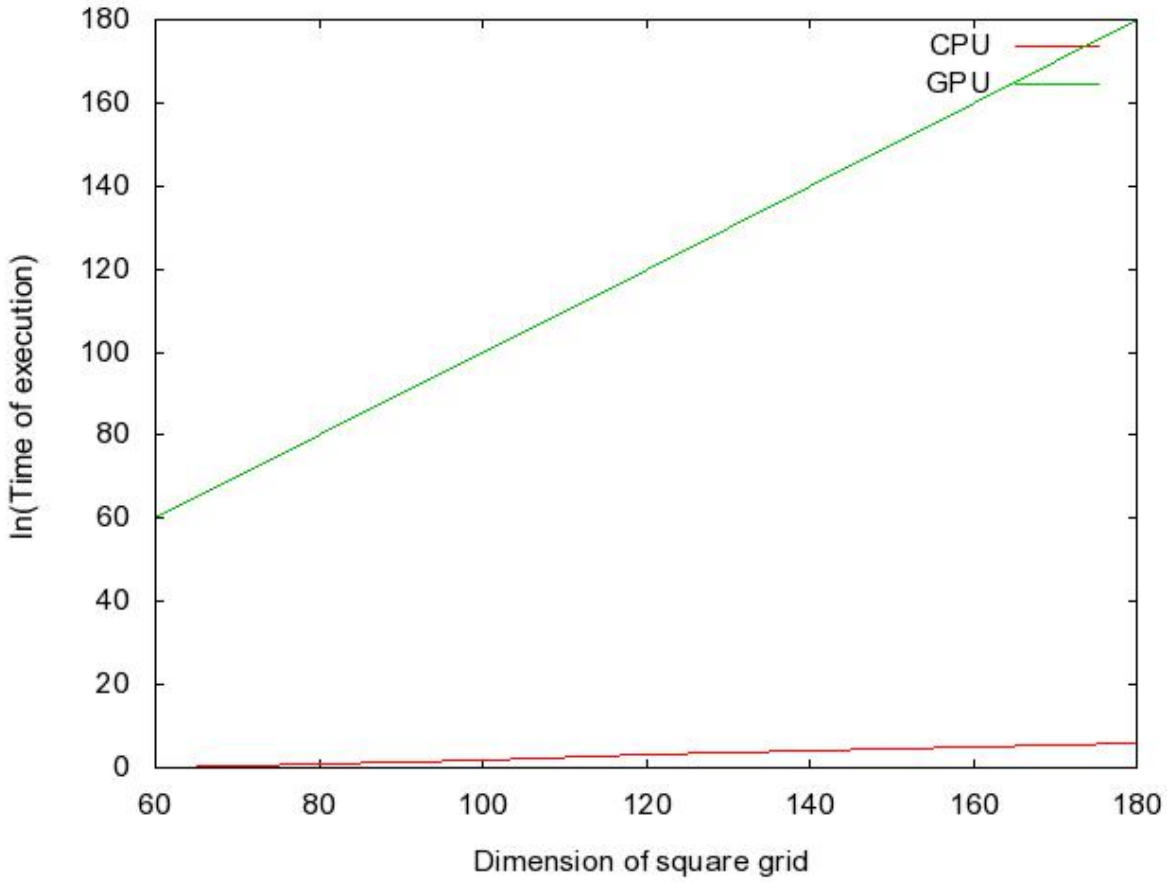

Diagram 2 Time of execution of SIMPLE algorithm depending on size of a square grid

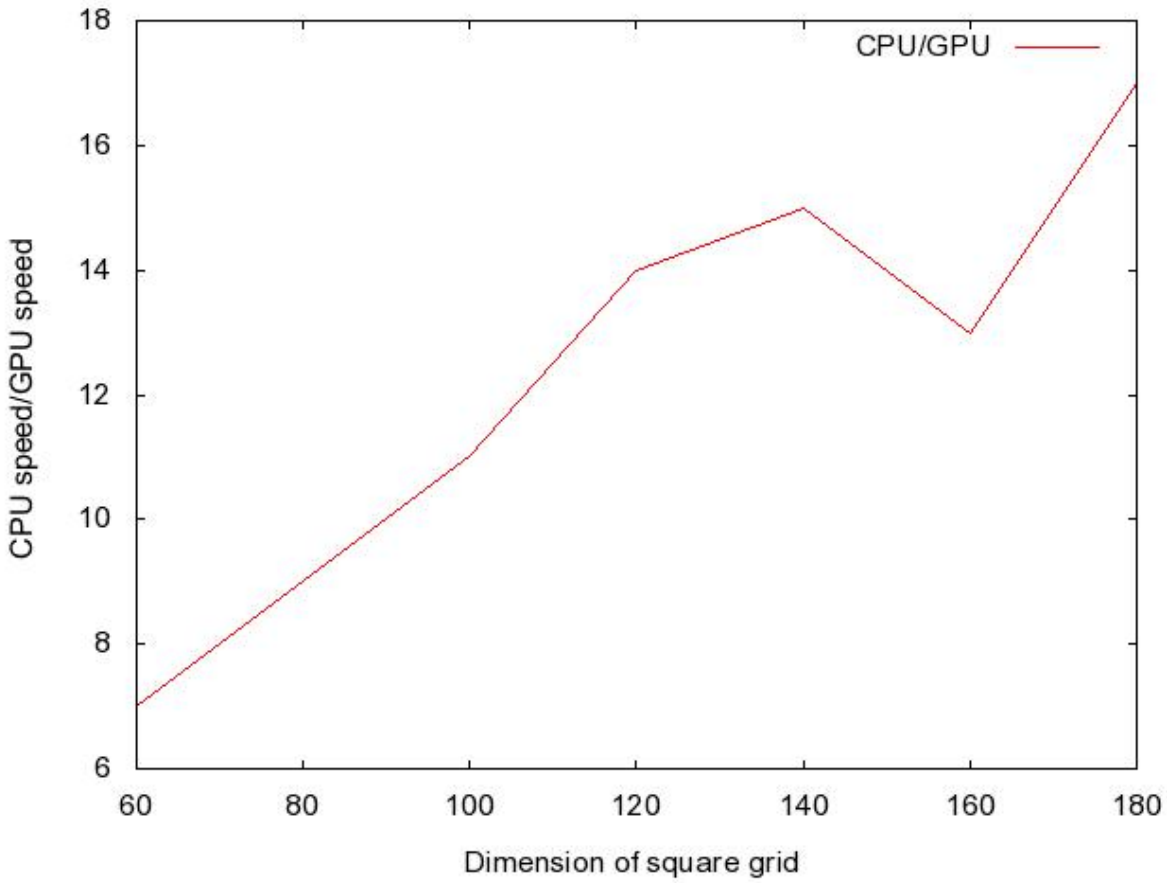

Diagram 3 Acceleration of GPU computation over CPU computation

CPU that was used is Intel(R) Core(TM) i5-2430M @ 2.40 GHz.

GPU that was used is GT200 @ 1.3 GHz, (nVidia Tesla S1070 card)

We can see from Diagram 3 that at lowest resolution GPU is already outperforming CPU by minimum factor of 7. There is no explanation for lower accelaration of program execution at resolution 160*160, as CPU somehow managed to do it faster then expected (Diagram 2).

5.2 Backwards facing step

Backwards facing step was calculated over grid of dimensions 1000*200, with value of Reynolds number 400. Results are shown in Figures (5.5)-(5.8). It's impossible to change font size of axis numbers in *Paraview* at the moment, so dimension on y-axis will be hard to read (dimension is 0.2).

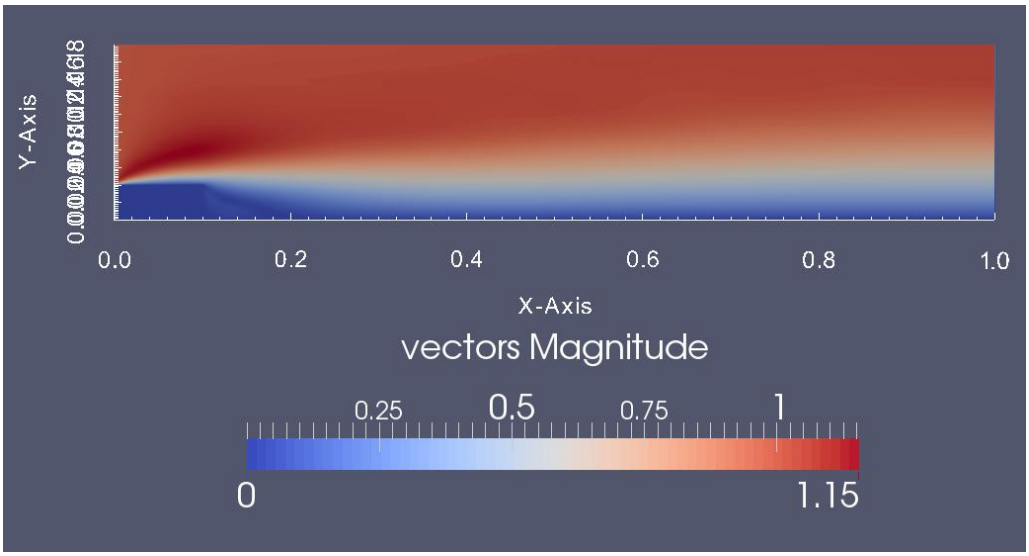

Figure 5.5 Solution of total velocity magnitudes in *backwards facing step*

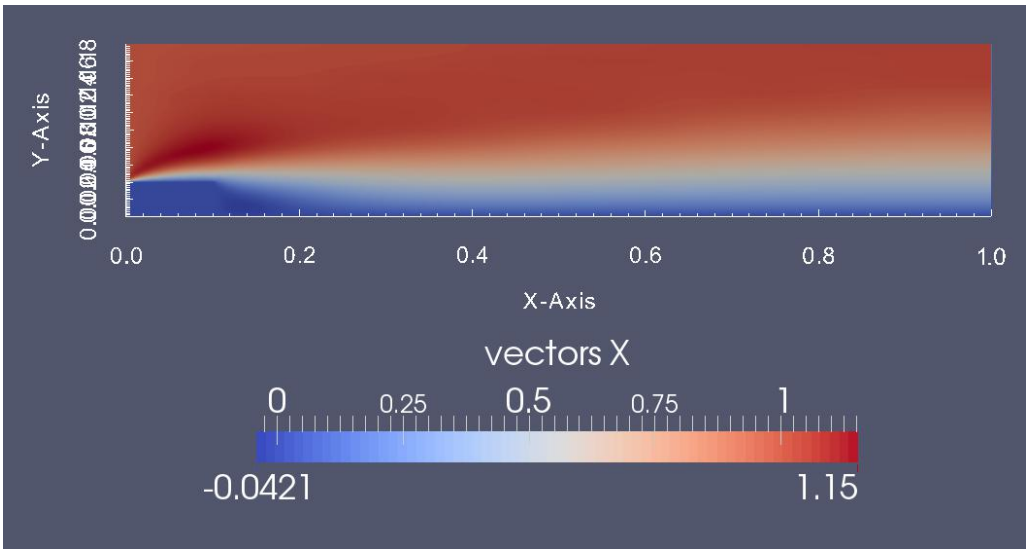

Figure 5.6 Solution of x-component velocity magnitudes in *backwards facing step*

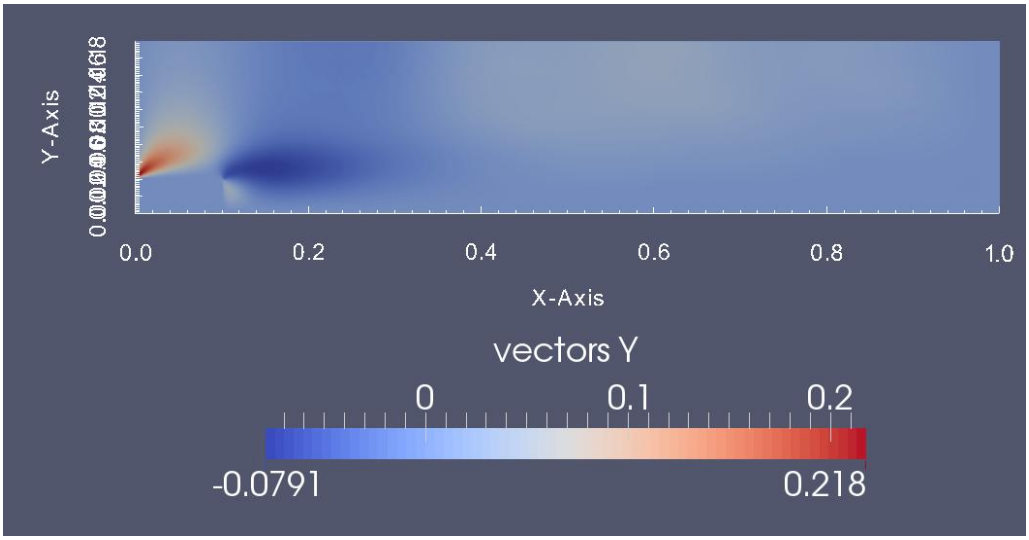

Figure 5.7 Solution of y-component velocity magnitudes in *backwards facing step*

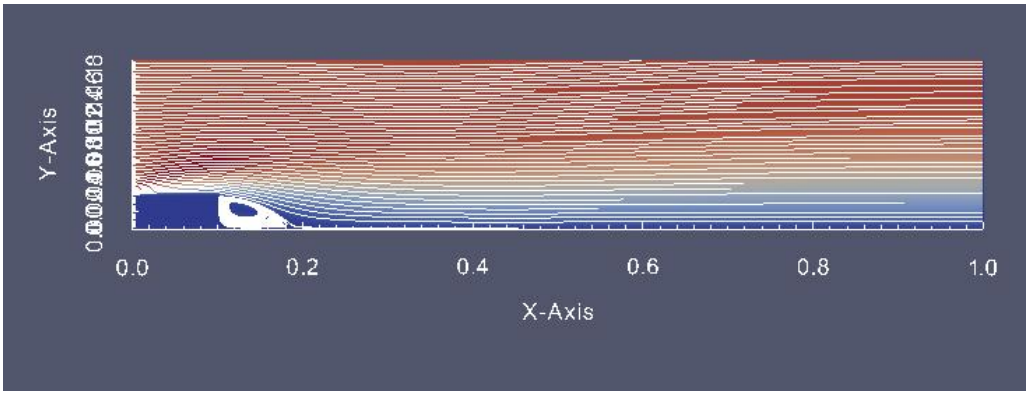

Figure 5.8 Visualization via *streamline plot technique* for *backwards facing step*

For simulation of plasma actuator we will use two values of parameter *A*, *A*=1000 and *A*=2000. Since repulsive force of plasma actuator decays rapidly, we will only add force in the area around actuator as shown in Diagram 1.

Results for *backwards facing step* with plasma actuator located at x=0.08, y=0.04 (using color distribution of Figures (5.5)-(5.8)) are shown in next figures.

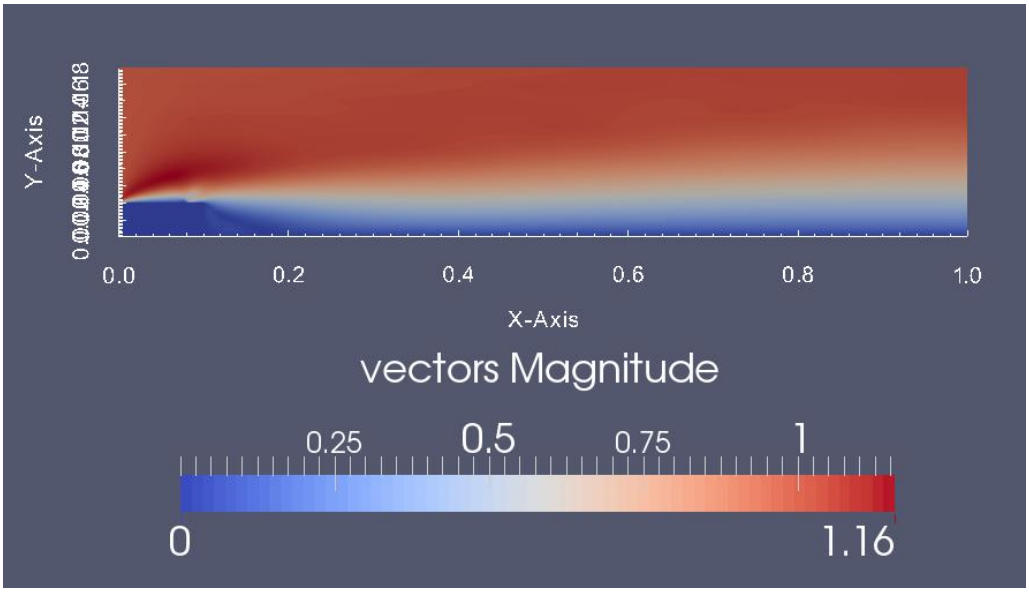

Figure 5.9 Solution of total velocity magnitudes in *backwards facing step* with plasma actuator (A=1000)

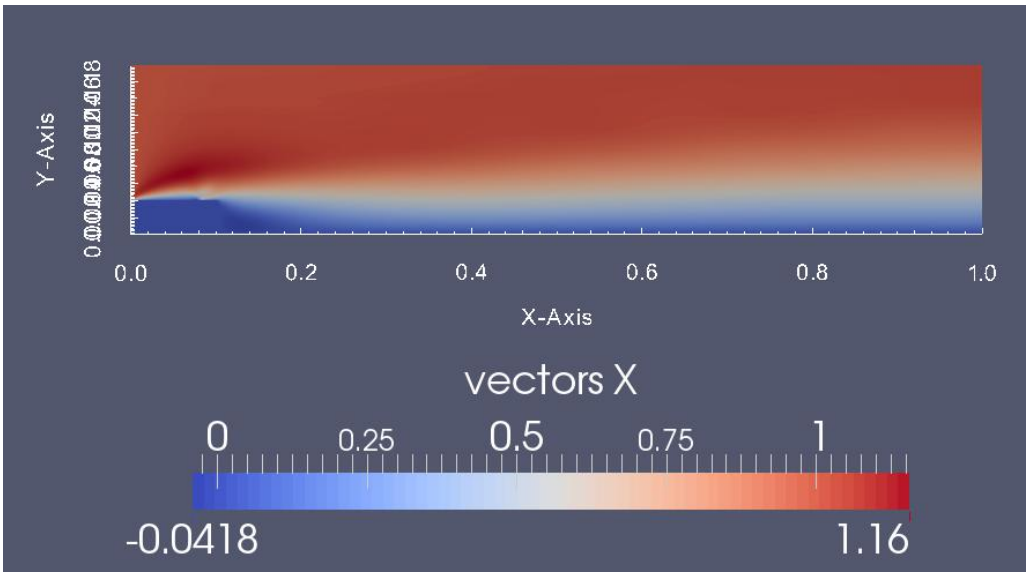

Figure 5.10 Solution of x-component velocity magnitudes in *backwards facing step* with plasma actuator (A=1000)

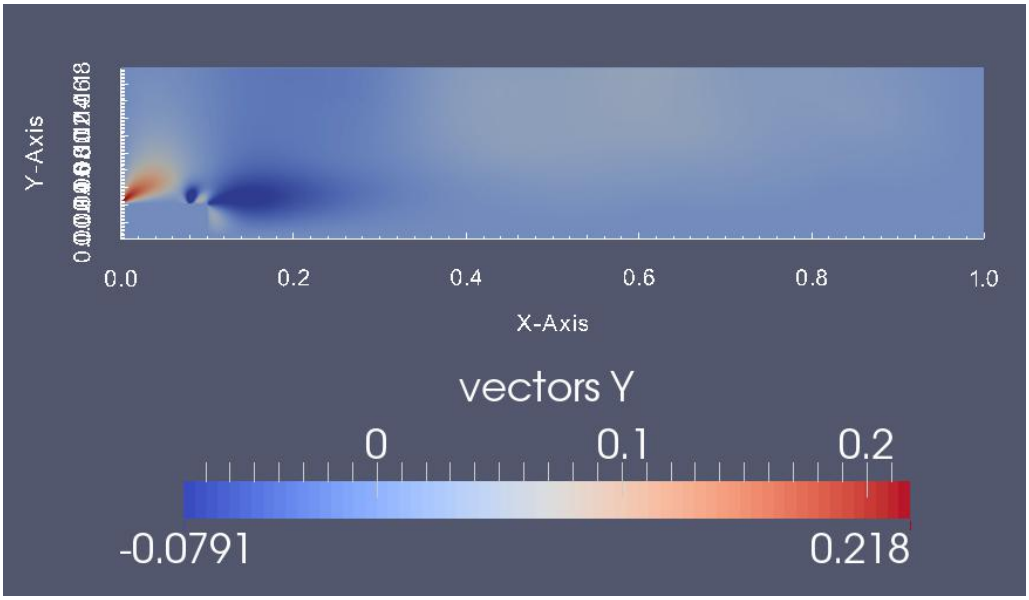

Figure 5.11 Solution of y-component velocity magnitudes in *backwards facing step* with plasma actuator (A=1000)

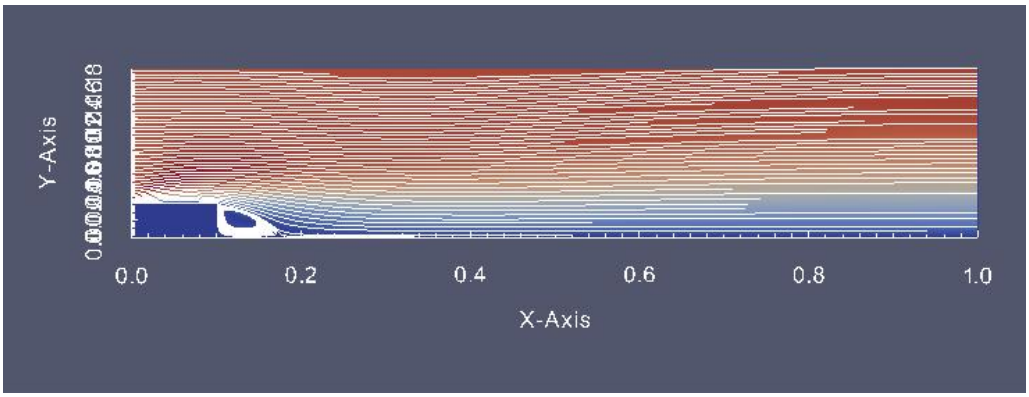

Figure 5.12 Visualization via *streamline plot technique* for *backwards facing step* with plasma actuator (A=1000)

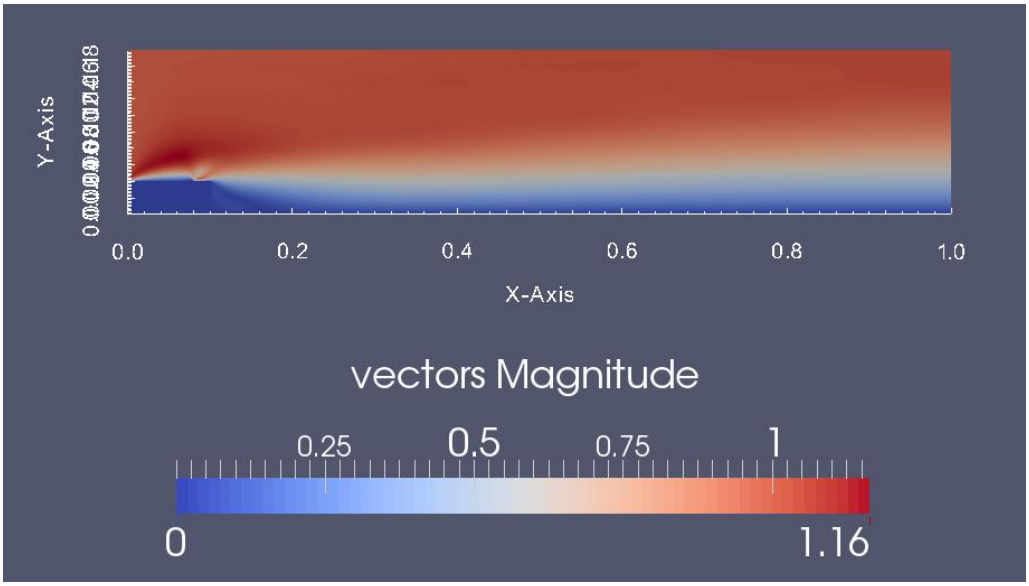

Figure 5.13 Solution of total velocity magnitudes in *backwards facing step* with plasma actuator (A=2000)

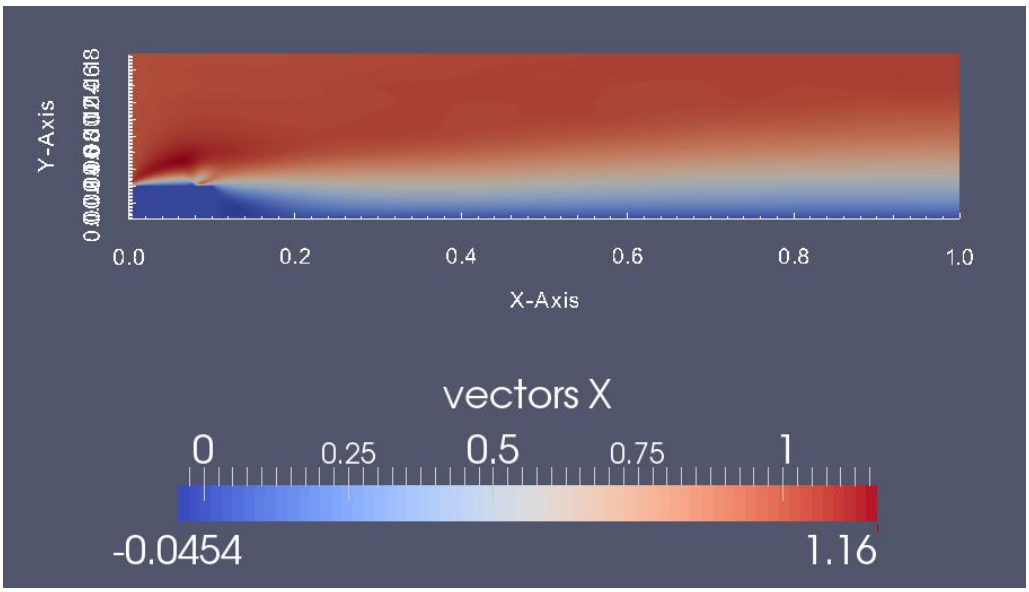

Figure 5.14 Solution of x-component velocity magnitudes in *backwards facing step* with plasma actuator (A=2000)

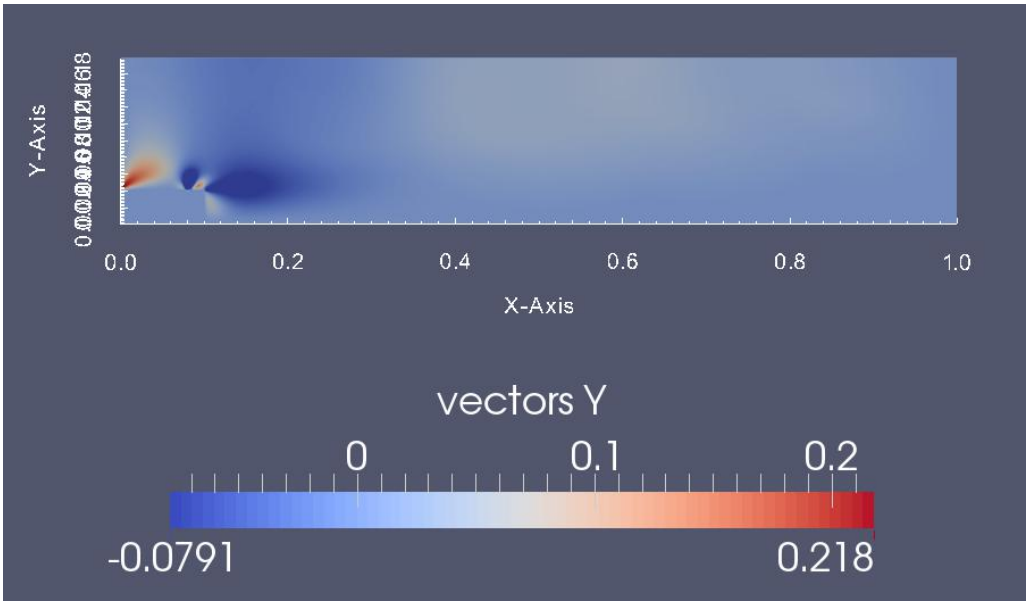

Figure 5.15 Solution of y-component velocity magnitudes in *backwards facing step* with plasma actuator (A=2000)

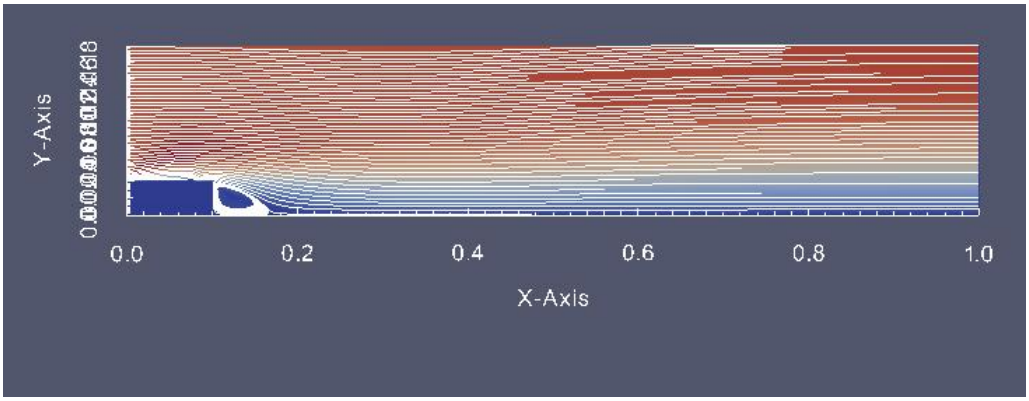

Figure 5.16 Visualization via *streamline plot technique* for *backwards facing step* with plasma actuator (A=2000)

Results shown in Figures (5.9)-(5.16) show some expected results for plasma actuators. When calculating with *A*=1000, we see we have decreased size of vortex that was created behind the step and with *A*=2000 we decreased it even more. We have also achieved that streams get closer to the backwards facing step, which is also expected [2], but what must be noted is that we have achieved closer streams with *A*=1000, then with *A*=2000. These results still need experimental conformation,

since plasma actuators on *backwards facing step* aren't commonly used and scientific publications about this subject are almost nonexistent.

6. DISCUSSION AND CONCLUSION

As stated in introduction, SIMPLE method can only solve steady flows, hence, no time-dependant forces or boundary conditions can be applied in this computation, which narrows greatly use of this kind of calculation. Restriction of incompressibility must also be taken seriously, as it is very unstable aproximation in some calculations, especially at high values of Reynolds number (we used value of 400 for all of our calculation).

One look at Diagram 2., shows that future of computational fluid dynamics is usage of CUDA and GPUs. Even at lowest resolution, acceleration factor is 7 and today's resolution used in computation fluid dynamics are way higher, hence, GPUs could be even more usefull, as they were in this calculation, which would lead to higher precision of calculation in more then even 20 times less time of execution.

Results presented for *driven lid cavity* and *backwards facing step* show that implementation of SIMPLE algorithm was correct, since there is a lot of results available via scientific publications. Result for plasma actuators still need experimental confirmation and simulation of plasma actuators are nowadays greatly discussed in science community, since it is fairly new research area. As SIMPLE method cannnot solve unsteady flows, it is very unlikeable that this approach will be used in future, because most applications of plasma actuators demand time dependent force and not constant force as used in this thesis.

One major flaw of code used in this thesis is that it can only solve problems with flat boundaries and it would require major upgrade to calculate fluid flow around non-flat boundaries. What must be noted also is that acceleration we

obtained using CUDA could be higher, since some parts of code could probably be more memory optimized, but even this version of code is capable of executing calculation more then ten times faster then CPU execution of code.

APPENDIX A – Code

```
#include <stdio.h> #include
<stdlib.h> #include
<string.h> #include
<cuda.h> #include
<cuda_runtime.h> #include
<math.h> #include <time.h>

#define BD 20
#define BX 50
#define BY 10
#define NX 1000
#define NY 200
#define KT 5
#define AI -2./3.
#define DT 0.00005
#define RX 100
#define RY 40
#define FX 0.7926
#define FY 0.6097
#define Bx 72000
#define By 900000
#define yb 0.00333
#define IY 40
#define IX 80
#define AMP 2000


#define Re 400
#define GR 0.0001
#define T 60000
__global__ void inix (float *a)
{
   __shared__ float s[BD][BD];
  int i = threadIdx.x;
  int j = threadIdx.y;
  int I = blockIdx.x;
  int J = blockIdx.y;
  int i0=I*BD;
  int j0=J*BD;
  int iu=i0+i;
  int ju=j0+j;
  int k=iu *NY+ju;
  if(iu<RX && ju>RY)
  s[i][j]=1.;
  else
  s[i][j]=0.;
  a[k]=s[i][j];
}

__global__ void ini (float *a)
{
   __shared__ float s[BD][BD];
  int i = threadIdx.x;
  int j = threadIdx.y;
  int I = blockIdx.x;
  int J = blockIdx.y;
  int i0=I*BD;
  int j0=J*BD;
  int iu=i0+i;
```

```
  int ju=j0+j;
  int k=iu *NY+ju;
  s[i][j]=0.;
  a[k]=s[i][j];
}

__global__ void rubnix (float *a)

{
   __shared__ float s[BD][BD];
  int i = threadIdx.x;
  int j = threadIdx.y;
  int I = blockIdx.x;
  int J = blockIdx.y;
  int i0=I*BD;
  int j0=J*BD;
  int iu=i0+i;
  int ju=j0+j;
  int k=iu *NY+ju;
  s[i][j]=a[k];
  __syncthreads();
  if(iu==0 &&
  ju>RY) s[i][j]=1.;
  if(iu==NX-2)
  s[i][j]=s[i-1][j];
  __syncthreads();
  if(ju==0)
  s[i][j]=0.; if(ju==NY-
  1) s[i][j]=s[i][j-1];
  if(ju==RY &&
  iu<RX) s[i][j]=0.;

  if(iu==RX && ju<=RY)
  s[i][j]=s[i+1][j]*AI;
  __syncthreads();
  a[k]=s[i][j];

}

__global__ void rubniy (float *a)

{
  __shared__ float s[BD][BD];
  int i = threadIdx.x;
  int j = threadIdx.y;
  int I = blockIdx.x;
  int J = blockIdx.y;
  int i0=I*BD;
  int j0=J*BD;
  int iu=i0+i;
  int ju=j0+j;
  int k=iu *NY+ju;
  s[i][j]=a[k];
  __syncthreads();
  if(ju==0)
  s[i][j]=AI*s[i][j+1];
  if(ju==NY-2)
  s[i][j]=s[i][j-1];
  __syncthreads();
  if(iu==0)
```

```
  s[i][j]=0.; if(iu==NX-1)
  s[i][j]=s[i-1][j];
  if(ju==RY &&
  iu<=RX)
  s[i][j]=AI*s[i][j+1];
  if(iu==RX && ju<RY)
  s[i][j]=0.;
  __syncthreads();
  a[k]=s[i][j];

}

__global__ void rubnip (float *a)

{
  __shared__ float s[BD][BD];
  int i = threadIdx.x;
  int j = threadIdx.y;
  int I = blockIdx.x;
  int J = blockIdx.y;
  int i0=I*BD;
  int j0=J*BD;
  int iu=i0+i;
  int ju=j0+j;
  int k=iu *NY+ju;
  s[i][j]=a[k];
  __syncthreads();
  if(ju==0 && iu!=0 && iu!=NX-1)
  s[i][j]=s[i][j+1];
  if(ju==NY-1 && iu!=0 && iu!=NX-1)
  s[i][j]=0.0;
  if(iu==0 && ju!=0 && ju!=NY-1)
  s[i][j]=0;
  if(iu==NX-1 && ju!=0 && ju!=NY-1)
  s[i][j]=0.;
  if(iu==RX && ju<=RY)
  s[i][j]=s[i+1][j];
  if(iu<=RX && ju==RY)
  s[i][j]=s[i][j+1];
  __syncthreads();
  a[k]=s[i][j];

}

__global__ void US (float *a, float *b, float *c, float *d)

{
  __shared__ float u[BD+2][BD+2], v[BD+2][BD+2], p[BD+2][BD+2], u1[BD+2][BD+2];
  int i = threadIdx.x;
  int j = threadIdx.y;
  int I = blockIdx.x;
  int J = blockIdx.y;
  int i0=I*BD;
  int j0=J*BD;
  int iu=i0+i;
  int ju=j0+j;
  int k=iu *NY+ju;

  u[i+1][j+1]=a[k];
  if(i==0 && iu!=0)
```

```
  u[0][j+1]=a[k-NY];
  if(i==BD-1 && iu!=NX-1)
  u[BD+1][j+1]=a[k+NY];
  if(j==0 && ju!=0)
  u[i+1][0]=a[k-1];
  if(j==BD-1 && ju!=NY-1)
  u[i+1][BD+1]=a[k+1];
  __syncthreads();

  v[i+1][j+1]=b[k];
  if(i==0 && iu!=0)
  v[0][j+1]=b[k-NY];
  if(i==BD-1 && iu!=NX-1)
  v[BD+1][j+1]=b[k+NY];
  if(j==0 && ju!=0)
  v[i+1][0]=b[k-1];
  if(j==BD-1 && ju!=NY-1)
  v[i+1][BD+1]=b[k+1];
  __syncthreads();

  p[i+1][j+1]=c[k];
  if(i==0 && iu!=0)
  p[0][j+1]=c[k-NY];
  if(i==BD-1 && iu!=NX-1)
  p[BD+1][j+1]=c[k+NY];
  if(j==0 && ju!=0)
  p[i+1][0]=c[k-1];
  if(j==BD-1 && ju!=NY-1)
  p[i+1][BD+1]=c[k+1];
  __syncthreads();
  if(iu<NX-2 && iu>0 && ju<NY-1 && ju>0 && ((iu<=RX && ju<=RY)!=1) ){ u1[i+1][j+1]=-
  ((pow(u[i+2][j+1],2)-pow(u[i][j+1],2))*NX/2.0f+(u[i+1][j+2]*0.5*(v[i+1][j+1] + v[i+2][j+1])-
u[i+1][j]*0.5*(v[i+1][j] + v[i+2][j]))*NY*KT/2.0); u1[i+1][j+1]=u1[i+1][j+1]+(1.0/Re)*((u[i+2][j+1]-
  2.0*u[i+1][j+1]+u[i][j+1])*NX*NX+(u[i+1][j+2]-
2.0*u[i+1][j+1]+u[i+1][j])*NY*NY*KT*KT);
  u1[i+1][j+1]=u[i+1][j+1]+DT*(u1[i+1][j+1]-1.0*NX*(p[i+2][j+1]-p[i+1][j+1]));
  if(ju-IY<15)
  {
      if((iu-IX)<15 && (iu-IX)>=0) u1[i+1][j+1]=u1[i+1][j+1]+AMP*DT*(FX*exp(0-((iu-IX+0.5-
         ju+IY)*0.001)*((iu-IX+0.5-
ju+IY)*0.001)/(((ju-IY)*0.001+yb)*((ju-IY)*0.001+yb))-Bx*0.001*0.001*(ju-IY)*(ju-IY)));
      if((iu-IX)>-15 && (iu-IX)<0)
         u1[i+1][j+1]=u1[i+1][j+1]-AMP*DT*(FX*exp(0-((iu-IX+0.5-ju+IY)*0.001)*((iu-IX+0.5-
ju+IY)*0.001)/(((ju-IY)*0.001+yb)*((ju-IY)*0.001+yb))-Bx*0.001*0.001*(ju-IY)*(ju-IY)));

  }
  d[k]=u1[i+1][j+1];}
  else
  d[k]=a[k];
  __syncthreads();

}

__global__ void VS (float *a, float *b, float *c, float *d)

{
  __shared__ float u[BD+2][BD+2], v[BD+2][BD+2], p[BD+2][BD+2],
  v1[BD+2][BD+2]; int i = threadIdx.x;
  int j = threadIdx.y;
  int I = blockIdx.x;
```

```
  int J = blockIdx.y;
  int i0=I*BD;
  int j0=J*BD;
  int iu=i0+i;
  int ju=j0+j;
  int k=iu *NY+ju;

  u[i+1][j+1]=a[k];
  if(i==0 && iu!=0)
  u[0][j+1]=a[k-NY];
  if(i==BD-1 && iu!=NX-1)
  u[BD+1][j+1]=a[k+NY];
  if(j==0 && ju!=0)
  u[i+1][0]=a[k-1];
  if(j==BD-1 && ju!=NY-1)
  u[i+1][BD+1]=a[k+1];
  __syncthreads();

  v[i+1][j+1]=b[k];
  if(i==0 && iu!=0)
  v[0][j+1]=b[k-NY];
  if(i==BD-1 && iu!=NX-1)
  v[BD+1][j+1]=b[k+NY];
  if(j==0 && ju!=0)
  v[i+1][0]=b[k-1];
  if(j==BD-1 && ju!=NY-1)
  v[i+1][BD+1]=b[k+1];
  __syncthreads();

  p[i+1][j+1]=c[k];
  if(i==0 && iu!=0)
  p[0][j+1]=c[k-NY];
  if(i==BD-1 && iu!=NX-1)
  p[BD+1][j+1]=c[k+NY];
  if(j==0 && ju!=0)
  p[i+1][0]=c[k-1];
  if(j==BD-1 && ju!=NY-1)
  p[i+1][BD+1]=c[k+1];
  __syncthreads();

  if(iu<NX-1 && iu>0 && ju<NY-2 && ju>0 && ((iu<=RX && ju<=RY)!=1)){
  v1[i+1][j+1]=-((v[i+2][j+1]*0.5*(u[i+1][j+1] + u[i+1][j+2])-v[i][j+1]*0.5*(u[i][j+1] +
u[i][j+2]))*NX/2.0+(v[i+1][j+2]*v[i+1][j+2]-v[i+1][j]*v[i+1][j])*KT*NY/2.0);
  v1[i+1][j+1]=v1[i+1][j+1]+(1.0f/Re)*((v[i+2][j+1]-2.0f*v[i+1][j+1]+v[i][j+1])*NX*NX+(v[i+1][j+2]-
2.0f*v[i+1][j+1]+v[i+1][j])*NY*NY*KT*KT);
  v1[i+1][j+1]=v[i+1][j+1]+DT*(v1[i+1][j+1]-(1.0f*NY*KT)*(p[i+1][j+2]-
  p[i+1][j+1])); if(ju-IY<15)
  {
      if((iu-IX)<15 && (iu-IX)>-15) v1[i+1][j+1]=v1[i+1][j+1]+AMP*DT*(FY*exp(0-(iu-
          IX)*0.001*0.001*(iu-IX)/(((ju-
IY+0.5)*0.001+yb)*((ju-IY+0.5)*0.001+yb))-By*(ju+0.5-IY)*0.001*0.001*(ju+0.5-IY)));


  }
  d[k]=v1[i+1][j+1];}
  else
  d[k]=b[k];
  __syncthreads();
```

```
}

__global__ void poiss (float *a, float *b, float *c, float *d)
{
    __shared__ float u[BD+2][BD+2], v[BD+2][BD+2], p[BD+2][BD+2], corr[BD][BD];
   int i = threadIdx.x;
   int j = threadIdx.y;
   int I = blockIdx.x;
   int J = blockIdx.y;
   int i0=I*BD;
   int j0=J*BD;
   int iu=i0+i;
   int ju=j0+j;
   int k=iu *NY+ju;

   u[i+1][j+1]=a[k];
   if(i==0 && iu!=0)
   u[0][j+1]=a[k-NY];
   if(i==BD-1 && iu!=NX-1)
   u[BD+1][j+1]=a[k+NY];
   if(j==0 && ju!=0)
   u[i+1][0]=a[k-1];
   if(j==BD-1 && ju!=NY-1)
   u[i+1][BD+1]=a[k+1];
   __syncthreads();

   v[i+1][j+1]=b[k];
   if(i==0 && iu!=0)
   v[0][j+1]=b[k-NY];
   if(i==BD-1 && iu!=NX-1)
   v[BD+1][j+1]=b[k+NY];
   if(j==0 && ju!=0)
   v[i+1][0]=b[k-1];
   if(j==BD-1 && ju!=NY-1)
   v[i+1][BD+1]=b[k+1];
   __syncthreads();

   p[i+1][j+1]=c[k];
   if(i==0 && iu!=0)
   p[0][j+1]=c[k-NY];
   if(i==BD-1 && iu!=NX-1)
   p[BD+1][j+1]=c[k+NY];
   if(j==0 && ju!=0)
   p[i+1][0]=c[k-1];
   if(j==BD-1 && ju!=NY-1)
   p[i+1][BD+1]=c[k+1];
   __syncthreads();

   corr[i][j]=c[k];
   __syncthreads();

   if(iu>0 && ju >0 && iu<NX-1 && ju<NY-1 && ((iu<=RX && ju<=RY)!=1)){
   c[k] = -(1.0/(2*(DT*NX*NX+DT*NY*NY*KT*KT)))*((-DT*NX*NX)*p[i+2][j+1]+(-
DT*NX*NX)*p[i][j+1]+(-DT*NY*NY*KT*KT)*p[i+1][j+2]+(-
DT*NY*NY*KT*KT)*p[i+1][j]+(1.0*NX)*(u[i+1][j+1] - u[i][j+1]) + (1.0*NY*KT)*(v[i+1][j+1]-v[i+1][j]));
   }
   corr[i][j]-=c[k];
   if(corr[i][j]<0)
   corr[i][j]*=(-1);
   d[k]=corr[i][j];
```

```
  __syncthreads();


}

__global__ void korek (float *a, float *b, float *c)
{
    __shared__ float u[BD+2][BD+2], v[BD+2][BD+2], p[BD+2][BD+2], corr[BD][BD];
  int i = threadIdx.x;
  int j = threadIdx.y;
  int I = blockIdx.x;
  int J = blockIdx.y;
  int i0=I*BD;
  int j0=J*BD;
  int iu=i0+i;
  int ju=j0+j;
  int k=iu *NY+ju;

  u[i+1][j+1]=a[k];
  if(i==0 && iu!=0)
  u[0][j+1]=a[k-NY];
  if(i==BD-1 && iu!=NX-1)
  u[BD+1][j+1]=a[k+NY];
  if(j==0 && ju!=0)
  u[i+1][0]=a[k-1];
  if(j==BD-1 && ju!=NY-1)
  u[i+1][BD+1]=a[k+1];
  __syncthreads();

  v[i+1][j+1]=b[k];
  if(i==0 && iu!=0)
  v[0][j+1]=b[k-NY];
  if(i==BD-1 && iu!=NX-1)
  v[BD+1][j+1]=b[k+NY];
  if(j==0 && ju!=0)
  v[i+1][0]=b[k-1];
  if(j==BD-1 && ju!=NY-1)
  v[i+1][BD+1]=b[k+1];
  __syncthreads();

  p[i+1][j+1]=c[k];
  if(i==0 && iu!=0)
  p[0][j+1]=c[k-NY];
  if(i==BD-1 && iu!=NX-1)
  p[BD+1][j+1]=c[k+NY];
  if(j==0 && ju!=0)
  p[i+1][0]=c[k-1];
  if(j==BD-1 && ju!=NY-1)
  p[i+1][BD+1]=c[k+1];
  __syncthreads();

  if(iu>0 && iu<NX-2 && ((iu<=RX && ju<=RY)!=1)){
  u[i+1][j+1]-=DT*NX*(p[i+2][j+1]-p[i+1][j+1]);
  a[k]=u[i+1][j+1];}

  if(ju>0 && ju<NY-2 && ((iu<=RX && ju<=RY)!=1)){
  v[i+1][j+1]-=DT*NY*KT*(p[i+1][j+2]-p[i+1][j+1]);
  b[k]=v[i+1][j+1];}
  __syncthreads();
```

```
}

__global__ void copy (float *a, float *b)
{
     __shared__ float u[BD][BD];
   int i = threadIdx.x;
   int j = threadIdx.y;
   int I = blockIdx.x;
   int J = blockIdx.y;
   int i0=I*BD;
   int j0=J*BD;
   int iu=i0+i;
   int ju=j0+j;
   int k=iu *NY+ju;

   u[i][j]=a[k];
   __syncthreads();

   b[k]=u[i][j];
   __syncthreads();

}

__global__ void suma (float *a, float *b)
{
     __shared__ float u[BD][BD];
   int i = threadIdx.x;
   int j = threadIdx.y;
   int I = blockIdx.x;
   int J = blockIdx.y;
   int i0=I*BD;
   int j0=J*BD;
   int iu=i0+i;
   int ju=j0+j;
   int k=iu *NY+ju;

   u[i][j]=a[k];
   __syncthreads();

   b[k]+=u[i][j];
   __syncthreads();

}


void PRINT_VECT_VTK(int imax, int jmax, float U[NX][NY], float V[NX][NY], float *x, float *y)

{ int i,j,k;

int nx,ny,nz;

float z = 0.0;

FILE *vect_field;

nx = imax;

ny = jmax;

nz = 1;
```

```
/* File grid.vtk*/

if((vect_field = fopen("a.vtk", "w")) ==

NULL) printf("The grid file could not be

open\n"); else{

fprintf(vect_field, "# vtk DataFile Version

2.0\n"); fprintf(vect_field, "Sample rectilinear

grid\n"); fprintf(vect_field, "ASCII\n");

fprintf(vect_field, "DATASET RECTILINEAR_GRID\n");

fprintf(vect_field, "DIMENSIONS %d %d %d\n", nx, ny,

nz); fprintf(vect_field, "X_COORDINATES %d float\n", nx);

for(i=0; i<=nx-1; i++)

fprintf(vect_field, "%f\n", x[i]);

fprintf(vect_field, "Y_COORDINATES %d float\n",

ny); for(j=0; j<=ny-1; j++)

fprintf(vect_field, "%f\n", y[j]);

fprintf(vect_field, "Z_COORDINATES %d float\n",

nz); fprintf(vect_field, "%f\n", z);

fprintf(vect_field, "POINT_DATA %d\n", (nx)*(ny)*(nz));

fprintf(vect_field, "VECTORS vectors float\n");

for(i=0; i<=ny-1; i++)

for(j=0; j<=nx-1; j++){

fprintf(vect_field, "%f\t %f\t %f\n", U[j][i], V[j][i], 0.0);

}

}

fclose(vect_field);

}

int main (void)
{

    clock_t t2,t1;
   FILE *fp, *fp1, *fp2;
   float a[NX*NY], *a_d, *a_d1, b[NX*NY], *b_d, *b_d1, c[NX*NY], *c_d, *c_d1, corr[NX*NY], *corr_d,
max;
```

```
  float dxx[NX], dyy[NY], xi[NX][NY], yi[NX][NY], dx1, dy1;
  int j,i,t, N=NX*NY, x, br, k, k1;
  float rop;
  rop=DT*NY*NY;
  dim3 blockDim, gridDim;
  blockDim.x=BD;
  blockDim.y=BD;
  gridDim.x=BX;
  gridDim.y=BY;
  fp=fopen("x.dat", "w");
  fp1=fopen("y.dat", "w");
  fp2=fopen("tlak.dat", "w");
    t1=clock();
  cudaMalloc((void **)&a_d, sizeof(float)*NX*NY);
  cudaMalloc((void **)&a_d1, sizeof(float)*NX*NY);

  cudaMalloc((void **)&b_d, sizeof(float)*NX*NY);
  cudaMalloc((void **)&b_d1, sizeof(float)*NX*NY);

  cudaMalloc((void **)&c_d, sizeof(float)*NX*NY);
  cudaMalloc((void **)&c_d1, sizeof(float)*NX*NY);

  cudaMalloc((void **)&corr_d, sizeof(float)*NX*NY);

  inix <<< gridDim, blockDim >>> (a_d);
  inix <<< gridDim, blockDim >>> (a_d1);
  ini <<< gridDim, blockDim >>> (b_d);
  ini <<< gridDim, blockDim >>> (b_d1);
  ini <<< gridDim, blockDim >>> (c_d);
  ini <<< gridDim, blockDim >>> (c_d1);
  printf("Start");
  for(t=0;t<T;t++)
  {
  US <<<gridDim, blockDim>>> (a_d, b_d, c_d,
  a_d1); VS <<<gridDim, blockDim>>> (a_d, b_d, c_d,
  b_d1); rubnix <<<gridDim, blockDim>>> (a_d1);
  rubniy <<<gridDim, blockDim>>> (b_d1);
  for(i=0;i<30;i++)
  poiss<<<gridDim,blockDim>>>(a_d1, b_d1, c_d1, corr_d);



  rubnip<<<gridDim,blockDim>>>(c_d1);
  korek<<<gridDim,blockDim>>>(a_d1, b_d1,
  c_d1); rubnix <<<gridDim, blockDim>>> (a_d1);
  rubniy <<<gridDim, blockDim>>> (b_d1);
  suma <<<gridDim, blockDim>>> (c_d1, c_d);
  ini <<<gridDim, blockDim>>> (c_d1);
  rubnip <<<gridDim, blockDim>>> (c_d);

////

  US <<<gridDim, blockDim>>> (a_d1, b_d, c_d,
  a_d); VS <<<gridDim, blockDim>>> (a_d, b_d1, c_d,
  b_d); rubnix <<<gridDim, blockDim>>> (a_d);
  rubniy <<<gridDim, blockDim>>> (b_d);

   for(i=0;i<30;i++)
   poiss<<<gridDim,blockDim>>>(a_d, b_d, c_d1, corr_d);
```

```
rubnip<<<gridDim,blockDim>>>(c_d1);
korek<<<gridDim,blockDim>>>(a_d, b_d,
c_d1); rubnix <<<gridDim, blockDim>>> (a_d);
rubniy <<<gridDim, blockDim>>> (b_d);
suma <<<gridDim, blockDim>>> (c_d1, c_d);
ini <<<gridDim, blockDim>>> (c_d1);
rubnip <<<gridDim, blockDim>>> (c_d);

}
cudaMemcpy(a,  a_d,  NX*NY*sizeof(float),  cudaMemcpyDeviceToHost);
cudaMemcpy(b,  b_d,  NX*NY*sizeof(float),  cudaMemcpyDeviceToHost);
cudaMemcpy(c, c_d, NX*NY*sizeof(float), cudaMemcpyDeviceToHost);
  for(i=0;i<NX;i++)
 for(j=0;j<NY;j++) if(i!=NX-1)
 fprintf(fp, "\n%d %d %f", i, j, a[i*NY+j]);
   for(i=0;i<NX;i++)
 for(j=0;j<NY;j++) if(j!=NY-1)
 fprintf(fp1, "\n%d %d %f", i, j, b[i*NY+j]);
   for(i=0;i<NX;i++)
 for(j=0;j<NY;j++)
 fprintf(fp2, "\n%d %d %f", i, j, c[i*NY+j]);
    t2=clock();

    printf(" %f\n",((float)(t2-
 t1))/CLOCKS_PER_SEC); cudaFree(a_d);
 cudaFree(a_d1);
 cudaFree(b_d);
 cudaFree(b_d1);
 cudaFree(c_d);
 cudaFree(c_d1);

 //Ispis za paraview

 dx1=1.0*NX-1;
 dy1=KT*NY-1;
 for(i=0;i<NX;i++)
    dxx[i]=1.0*i/dx1;
 for(i=0;i<NY;i++)
    dyy[i]=1.0*i/dy1;
 for(i=0;i<NX;i++)
    {
        xi[i][0]=0.0;
        yi[i][0]=0.0;
        xi[i][NY-1]=0.0;
        yi[i][NY-1]=0.0;
    }
 for(i=0;i<NY;i++)
 {
        xi[0][i]=1.0;
        yi[0][i]=0.0;
        xi[NX-1][i]=0.0;
        yi[NX-1][i]=0.0;
 }

 for(i=1;i<=NX-2;i++)
    for(j=1;j<=NY-1;j++)
```

```
        {
            k=(i-1)*NY+j;
            k1=i*NY+j;
            xi[i][j]=0.5*(a[k]+a[k+NY]);
        }
    for(i=1;i<=NX-1;i++)
        for(j=1;j<=NY-2;j++)
        {
            k=i*NY+j-1; k1=i*NY+j;

            yi[i][j]=0.5*(b[k]+b[k+1]);

        }
    i=NX;
    j=NY;
    PRINT_VECT_VTK(i, j, xi, yi, dxx, dyy);


}
```

REFERENCES

[1] J.D. Andersson, *Computational fluid dynamics: The basics with application* (1995)

[2] M. Riherd, S.Roy, *Serpentine geometry plasma actuators for flow control*, Journal of applied physics 114, 083303 (2013)

[3] D.M. Schatzman,F. O. Thomas. *Turbulent boundary-layer separation control with single dielectric barrier discharge plasma actuators*, *AIAA journal* 48, no. 8 (2010)

[4] M.Matyka, *Solution to two-dimensional Incompressible Navier-Stokes Equations with SIMPLE, SIMPLER and Vorticity-Stream Function Approaches. Driven-Lid Cavity Problem: Solution and Visualization*, arXiv preprint physics/0407002 (2004)

[5] NVIDIA, NVIDIA CUDA C Programming Guide, (Version 3.2)

[6] Department of Physics, University of Split, Croatia, cluster homepage: http://www.gpuhybrid.org/

[7] Paraview – Open source scientific visualization http://www.paraview.org/

[8] https://www.thermalfluidscentral.org/encyclopedia/images/thumb/b/b7/Fig4.22.png/400px-Fig4.22.png