Computing Viscoelastic Fluid Dynamics on GPUs

By:
Adam Kagel

Advisor:
Dr. Paloma Gutierrez-Castillo

A senior thesis presented for the degree of
Bachelors of Science

Mathematics Department
The University of California, Davis
June 10, 2019

Abstract

This senior thesis presents simulations of the Stokes-Oldroyd-B viscoelastic fluid model in a 4-roll mill geometry run on Graphical Processing Units (GPU). A theoretical justification detailing why these simulations can be run on GPUs is presented. In addition, results are compared against those of simulations with the same parameters but obtained via traditional computing hardware (CPU). The GPU implementation is found to significantly outperform the previous CPU implementation, and allows for the possibility of more computationally intensive simulations of Stokes-Oldroyd-B to be conducted.

Contents

Introduction

Instabilities in viscoelastic fluids in the low Reynolds number regime, where viscous effects dominate inertia, have been studied for many years. Numerical simulations have been proven to be useful tools to study these fluids. However, these numerical simulations have been largely confined to the two-dimensional (2D) regime due to the already large computational cost of the simulations at acceptable resolutions. Traditionally these numerical simulations have been run on typical computer hardware (on a CPU). However, many large, expensive computations have been made much more feasible with the concept of running simulations in parallel on a computer's Graphics Processing Unit (GPU). However, this is not applicable to all numerical simulations, and we detail exactly why this technique is of use for the study of viscoelastic fluid dynamics.

System of Interest

We are solving the model system

$$\nabla^2 \mathbf{u} - \nabla p = -\beta \nabla \cdot \mathbf{S} - \mathbf{f}\,, \quad \nabla \cdot \mathbf{u} = 0 \qquad \text{Stokes}$$

$$\frac{\partial \mathbf{S}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{S} - \nabla \mathbf{u} \mathbf{S} - \mathbf{S} \nabla \mathbf{u}^T = \frac{1}{Wi}(\mathbf{I} - \mathbf{S}) + \nu_p \Delta \mathbf{S} \qquad \text{Oldroyd-B with diffusion}$$

with $\mathbf{u}$ the fluid velocity, p the fluid pressure, $\mathbf{S}$ the symmetric conformation tensor, $\mathbf{f}$ the background force, and ν_p the diffusion constant. This model is refered to as the Stokes-Oldroyd-B model with diffusion. The symmetric conformation tensor is a macroscopic average of the polymer orientation/stretching related to the polymer stress tensor by $\boldsymbol{\tau}_p = \beta(\mathbf{S} - \mathbf{I})$ [4].

The model depends on two parameters, β, which represents the dimensionless polymer-stiffness, and Wi, the Weissenberg number, which represents the dimensionless relaxation time (or memory), respectively defined as:

$$\beta = \frac{GL}{\mu U} \qquad Wi = \frac{\lambda U}{L}$$

where μ is the fluid viscosity, λ is the fluid relaxation time, G is the polymer elastic modulus, L is the characteristic length scale, and U is a characteristic velocity scale. In our simulations, we fix $\beta \cdot Wi = 0.5$, which is consistent with fluids used in experiments with dilute polymer solutions in highly viscous solvents.

We investigate the fluid model in a four-roll mill geometry. This geometry is realized with:

$$\mathbf{f} = \begin{pmatrix} 2 \sin x \cos y \\ -2 \cos x \sin y \\ 0 \end{pmatrix}$$

which is depicted in Figure 1 along with stagnation points, representing positions where the velocities are less than 10^{-3} in magnitude.

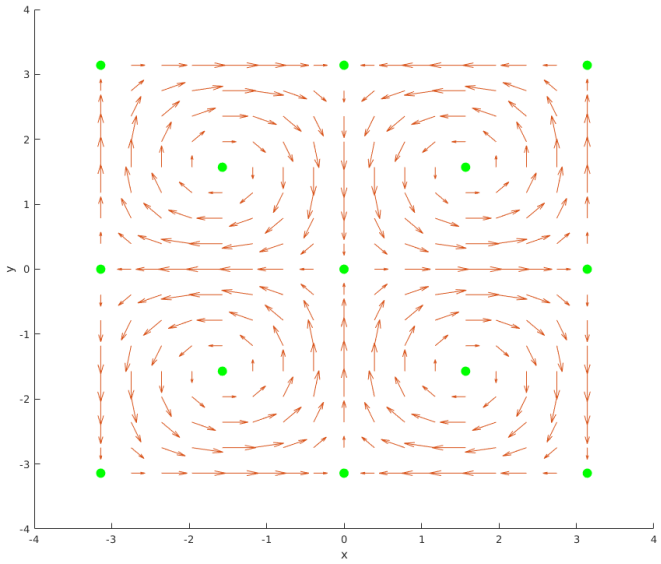


Figure 1: Forces and Stagnation Points

This force field corresponds to a four-roll velocity field $\mathbf{u} = -\frac{1}{2}\mathbf{f}$ in Newtonian-Stokes flow ($\beta = 0$).

We solve this system via a pseudospectral method, in which we time-step terms in the spatial-frequency domain via Adams-Bashforth Crank-Nicholson (ABCN). Previous work by [3] has taken this approach to investigate this 2D-system in a four-roll mill geometry/force-field, with the interest of applying this model towards mixing in biological/polymeric solutions.

In the past, numerical simulations of 2D viscoelastic fluids (VE-fluids) have been useful in explanations of phenomena exhibited in real-world experiments. Numerical simulations of VE-fluids have been traditionally constrained to 2D, because a single 3D simulation on the timescales of interest often requires several weeks to complete. Because of this, simulations of the 3D dynamics of VE-fluid systems are discouragingly slow and are consequently not well researched. This is in spite of their potential usefulness towards experimentalists, some of which have observed dynamics not exhibited by traditional 2D simulations [5].

We seek to greatly improve the speed of these simulations through the use of GPU assisted computation. To understand why GPU assisted computation can improve the speed of our simulations, we examine our methodology for solving the system.

Adams-Bashforth

In order to understand ABCN, we first describe Adams-Bashforth, in the form utilized in ABCN. For an Ordinary Differential Equation (ODE) of the general form:

$$\frac{dy}{dt} = f(t, y)$$

within some region $a \leq t \leq b$ with the initial condition $y(a) = \alpha$, we can numerically compute solutions to a desired accuracy via a two step Adams-Bashforth method:

$$w_0 = \alpha, w_1 = \alpha_1$$

$$w_{i+1} = w_i + \frac{h}{2}[3f(t_i, w_i) - f(t_{i-1}, w_{i-1})]$$

on $i = 1, 2, ..., \frac{b-a}{h} - 1$, where w_i approximates $y(t_i)$, α_1 is set to approximate $y(t_1)$, and $t_{i+1} = t_i + h$ [2].

For our system of interest, we temporarily ignore the diffusion term $\nu_p \Delta \mathbf{S}$. Since the differential equation (DE) is first order in t, if we can compute the term

$$RHS(\mathbf{u}, \mathbf{S}) = \frac{1}{Wi}(\mathbf{I} - \mathbf{S}) - \mathbf{u} \cdot \nabla \mathbf{S} + \nabla \mathbf{u} \mathbf{S} + \mathbf{S} \nabla \mathbf{u}^T$$

given arbitrary $\mathbf{S}$ and $\mathbf{u}$, then we can apply the Adams-Bashforth method detailed above.

In order to compute $RHS(\mathbf{u}, \mathbf{S})$ as simply as possible, we work in Fourier space, where differentiations are represented as multiplications by wavenumbers. Specifically, for any field $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$:

$$\overline{\nabla \mathbf{f}(\mathbf{x})} = i\mathbf{k}\bar{\mathbf{f}}(\mathbf{k})$$

where the $\overline{\cdot}$ operation represents the Fourier transform on a field over some spatial domain D, which for a general field $\mathbf{f}$, is given as:

$$\bar{\mathbf{f}}(\mathbf{k}) = \int_D \mathbf{f}(\mathbf{x}) e^{i\mathbf{x}\cdot\mathbf{k}} d\mathbf{x}$$

With this in mind, we can represent RHS in Fourier space as:

$$\overline{RHS}(\mathbf{u}, \mathbf{S}) = \frac{1}{Wi}(\bar{\mathbf{I}} - \bar{\mathbf{S}}) - \overline{\mathbf{u} \cdot IFT(i\mathbf{k}\bar{\mathbf{S}})} + \overline{IFT(i\mathbf{k}\bar{\mathbf{u}})\ \mathbf{S}} + \overline{\mathbf{S}\ IFT(i\mathbf{k}\overline{\mathbf{u}^T})}$$

where $IFT(f)$ represents the inverse Fourier transform on a field $\mathbf{f}$ over some wavenumber domain $\hat{D}$:

$$IFT(\mathbf{f})(\mathbf{x}) = \int_{\hat{D}} \mathbf{f}(\mathbf{k}) e^{i\mathbf{x}\cdot\mathbf{k}} d\mathbf{k}$$

Discrete Fourier Transforms

In order to utilize these multiple Fourier/inverse Fourier transforms in our computations, we require a numerical algorithm to compute the Fourier transform of a field at a point. Furthermore, because our fields of interest will be discredited, we require an analog of the Fourier transform for discrete

fields. This discrete operation is known as the Discrete Fourier transform (DFT), and, in one dimension, is given as:

$$\overline{f}_k = \sum_{n=0}^{N-1} f_n e^{ik\frac{n}{N}}$$

There are multidimensional extensions to the above, but the arguments presented in this section really only necessitate the use of the 1D case, as generalizations to higher dimensions follow naturally. Direct evaluation of the DFT occurs on the order of $O(n^2)$, however, there is a class of algorithms, detonated as Fast Fourier Transforms (FFT's) which can evaluate the DFT of a discredited field in $O(n \log n)$ [2]. The most ubiquitous of these FFT's is the Cooley-Tukey algorithm, a classic example of a divide-and-conquer algorithm.

Stability and ABCN

Now we address the problem of the diffusion term $\nu_p \Delta \mathbf{S}$. We can not so easily include the term in the Adams-Bashforth scheme, because Adams-Bashforth is not an unconditionally stable scheme. This means that for certain systems, the error in the numerical solutions grows so large that it dominates the calculations. These certain systems are called stiff.

Stokes Oldroyd-B with diffusion is one such stiff system, particularly because of the diffusion term. Because our system is stiff, we must use an unconditionally stable scheme, in particular, ABCN. ABCN is one method in a broader class of methods called Implicit Explicit (IMEX) schemes, which uses efficient time explicit methods on non-stiff terms in combination with more stable time implicit methods on the stiff terms of a system, which makes the method unconditionally stable. For example, consider the DE:

$$\frac{du}{dt} = f_{ex}(u) + \nu f_{im}(u)$$

where $f_{ex}(u)$ contains non-stiff (often non-linear) terms, and $f_{im}(u)$ contains stiff terms. ABCN uses Adams-Bashforth to evaluate f_{ex} and implicitly time averages f_{im} (philosophically similar to the Crank-Nicolson method for parabolic Partial Differential Equations) and adds the terms together for the expression:

$$\frac{u_{i+1} - u_i}{dt} = \frac{1}{2}[3f_{ex}(u_i) - f_{ex}(u_{i-1})] + \frac{\nu}{2}[f_{im}(u_{i+1}) + f_{im}(u_i)]$$

[1].

Setting $u_i = \overline{\mathbf{S}}_i$, $f_{ex}(\mathbf{S}) = \overline{RHS(\mathbf{u}, \mathbf{S})}$, and $f_{im}(\mathbf{S}) = \overline{\Delta \mathbf{S}} = -|\mathbf{k}|^2 \overline{\mathbf{S}}$, we can derive our final time-stepping formula:

$$\overline{\mathbf{S}}_{i+1} = \frac{1-g}{1+g}\overline{\mathbf{S}}_i + \frac{dt}{2}[3 \cdot \overline{RHS(\mathbf{u}_i, \mathbf{S}_i)} - \overline{RHS(\mathbf{u}_{i-1}, \mathbf{S}_{i-1})}]\frac{1}{1+g}$$

where $g = \frac{\nu_p |\mathbf{k}|^2 dt}{2}$.

Parallelizability and GPUs

For a linear system, transforming the system into the spatial-frequency domain and back via FFTs is not necessary for every iteration. Because of the nonlinear terms in our system, several IFFTs are required for each iteration, in order to multiply terms together in the spatial domain. The alternative approach is to compute the convolution of the two terms, which would end up being computationally more expensive. Because of this, the CPU implementation spends the most time performing these IFFTs and FFTs, since all other operations are much simpler matrix additions and direct entry-by-entry multiplications.

The CPU of a traditional computer is specialized for rapidly performing specific sequential tasks. This means that computations/algorithms with steps that can be performed independently of one another are often still run in a particular, step-by-step order. While modern computers have "multi-core" CPUs, which can perform tasks under multiple "threads", the number of independent tasks which can be performed is typically on the order of 8. So when performing an FFT over a large number of data points on a modern CPU, there is a similarly large number of independent steps which are needlessly ran in sequence.

However, the CPU is not the only piece of computational hardware in a modern computer. Most modern computers have a dedicated Graphics Processing Unit (GPU) entirely separate from the CPU, which performs all of the computations relevant to displaying graphics on a computer's monitor. Because these graphical computations are comprised of several independent additions/multiplications, GPUs are specifically designed to perform these numerous tasks independently of one another. While a CPU can perform each specific addition/multiplication several times faster than a GPU, a GPU can perform between hundreds and thousands of these operations at the same time.

For example, suppose a CPU can perform a single addition/multiplication 10 times faster than a GPU. However, the GPU can perform 100 of these operations at the same time. Naturally, by the time a CPU has performed 10 such operations, the GPU will have performed 100. This observation illustrates that GPUs are naturally specialized for divide-and-conquer algorithms. Because FFTs are divide-and-conquer algorithms, we can use GPUs to dramatically reduce the amount of time our code spends performing IFFTs and FFTs.

GPU aided computation is already a well known, well documented tool and, because of this, many programming languages/platforms geared towards computational endeavors have built in Application Programming Interfaces (APIs) to interact with GPUs. In particular, recent versions of MATLAB have built in support for GPU computing, in a form resembling the typical MATLAB language/design philosophy. MATLAB's GPU computing API automatically calibrates several of MATLAB's native functions to most optimally run on the GPU (e.g "fft" and "ifft"). Furthermore, MATLAB's language natively has N-D matrix data-structures with their corresponding operations, and its GPU computing API smoothly accommodates these structures without any unnecessary alterations from the user. For these reasons, it was decided to develop the GPU implementation in MATLAB.

Another potential candidate solution was Nvidia's C++/CUDA API. Nvidia is one of the primary manufacturers of GPUs, and their C++ API, CUDA, allows for the development of compiled

code to run on their GPUs as efficiently as possible. While C++/CUDA implementations of parallelizable algorithms have been shown to significantly outperform MATLAB implementations, C++ has no native support for matrix structures/operations. Furthermore, while the CUDA API provided by Nvidia does have sub-packages dedicated to FFT's (cuFFT) and efficient matrix operations (cuBLAS), its implementation is still not as intuitive as those of MATLAB and pose a significant learning curve to a developer new to the interface. While in the future, an implementation may be written for C++/CUDA, overcoming the steep learning curve is not worth the effort for a primary test of the advantages of the GPU implementation over a CPU implementation.

Results

Development of 2D GPU implementation

Despite the speedup potential of the GPU implementation, there are a few important shortcomings of GPU's which were considered in the development of the 2D GPU implementation.

Repetitively calculated or defined variables were initialized at the start and stored on the GPU preemptively. Any quantities to be computed on the GPU were made to be computed from quantities/variables already stored on the GPU. Even simple matrices, such as the identity, were stored on the GPU at the start of the program. Otherwise, the identity matrix will repetitively be moved from the CPU to the GPU. This was done for two reasons. Firstly, CPU to GPU data transfer rates are slow relative to normal internal CPU data transfer rates. Secondly, GPU memory allocation is considerably slower than CPU memory allocation.

Further compounding issues like this, GPUs have a small amount of onboard memory compared to a CPU. The specific GPUs used in my implementation have only about 8 GB of available memory, meaning that only that much data can be initialized on the GPU. While it wasn't necessary, adding a buffer variable to store calculated results was considered in order to reduce memory usage. This would have only been necessary if there were numerous computations which could not have been grouped into a larger one, but this was not the case.

Lastly, GPUs have a smaller memory bandwidth. In other words, performing operations on larger datasets perform slower than trends established on smaller datasets would suggest. In other words, there are diminishing returns in performance beyond a certain threshold amount of data. This was not crucial to the success of the 2D GPU implementation, but it still imposes a limit on the grid resolution.

Speed-Ups

The GPU implementation of our simulations significantly outperformed the CPU implementation, as illustrated in Table 1.

Resolution	CPU(s)	GPU(s)	Speed-Up
128×128	11.2275	3.55122	$\times 3.166$
256×256	98.8118	10.0157	$\times 9.866$
512×512	1904.8	42.1485	$\times 45.193$

Table 1: 2D Simulation Speed Ups on run from t = 0 - 5

Numerical details and parameters

While the GPU implementation outperforms the CPU implementation, a thorough comparison between the results obtained by each implementation is warranted in order to verify that the GPU implementation gives expected results. This is done by comparing results obtained in [3] with results obtained by the GPU implementation under the same details and parameters.

In 2D, the Stokes-Oldroyd B system is solved in a periodic domain, $[-\pi, \pi)^2$ with $N = 256$ grid points in both directions, yielding $dx = \frac{2\pi}{256}$. The time-step, dt, is scaled according to the grid size, and for these conditions, $dt = 0.0025$. The dynamics of interest occur on large time-scales, so in order to capture their behavior, simulations are typically ran to $t = 1000 \cdot Wi$. Simulations are started with an initial condition consisting of an isotropic initial conformation tensor, $\mathbf{S}_0 = \mathbf{I}$, randomly perturbed by low frequency Fourier modes.

It's well documented that for $Wi > 2$, the four-roll mill geometry forms a singularity at the origin [4]. This is largely due to the unphysicality of the Oldroyd-B model, which allows for infinitely stretched polymers (hence infinite polymer stress). The inclusion of a scale-dependent polymer stress diffusion term has been shown to numerically smooth singularities out of these simulations. Furthermore, solutions using the Oldroyd-B model with diffusion are shown to exhibit nearly identical dynamics to solutions using a FENE-P model [3]. Therefore, we add this diffusion term in our simulations to avoid central singularities, which is simpler and less computationally expensive than implementing models like Geisekus, PTT, or Finite Extension. The diffusion is set to $\nu = 0.001$ for all results presented.

Proper Orthogonal Decomposition

The main tool we employ to analyze the solutions from our simulations is the Proper Orthogonal Decomposition (POD). We decompose our solutions to a minimal number of basis functions or modes that capture a desired amount of the energy in the system. For vectors of interest, $\mathbf{q}(\boldsymbol{\xi}, t)$, we look for a separation of spatial and temporal modes and decompose about the temporal mean as:

$$\mathbf{q}(\boldsymbol{\xi}, t) - \overline{\mathbf{q}}(\boldsymbol{\xi}) = \sum_j a_j(t)\phi_j(\boldsymbol{\xi})$$

where $a_j(t)$ are our temporal modes and $\phi_j(\boldsymbol{\xi})$ are our spatial modes [6].

In order to analyze our system via POD, we must find quantities which are representative our the energy in our system. It can be shown that these quantities are the unique symmetric square roots, $\mathbf{b}$, of the symmetric conformation tensor $\mathbf{S} = \mathbf{b}^2$ [3]. So for our purposes, we look for spatial and temporal modes of $\mathbf{b}$ about the temporal mean:

$$\mathbf{b}(\boldsymbol{\xi}, t) - \overline{\mathbf{b}}(\boldsymbol{\xi}) = \sum_j a_j(t)\phi_j(\boldsymbol{\xi})$$

Also of note, we have a target of 95% energy capture, meaning only the highest energy modes, whose net contribution is 95% of the total energy, are considered. Typically this will capture all dynamic activity of interest.

2D Implementation Results

For similar computational set-ups of the same system, it hass been observed that there are several different flow regimes depending on Wi. Previous results by [3] chart the long term dynamics of

this system from $Wi \leq 5$ through $Wi = 12$ in increments of 0.25 as follows:

Wi	# flow transitions	long-time dynamics
≤ 5.00	0	Steady with 2 symmetries
5.25 - 6.00	1	Steady with 1 symmetry
6.25 - 7.75	1	Periodic (oscillatory)
8.00 - 8.75	1	Periodic (loop)
9.00 - 10.25	2	Aperiodic
10.50 - 12.00	2	Periodic (Dominant Vortex)

Table 2: Categorization of flow states at different Wi

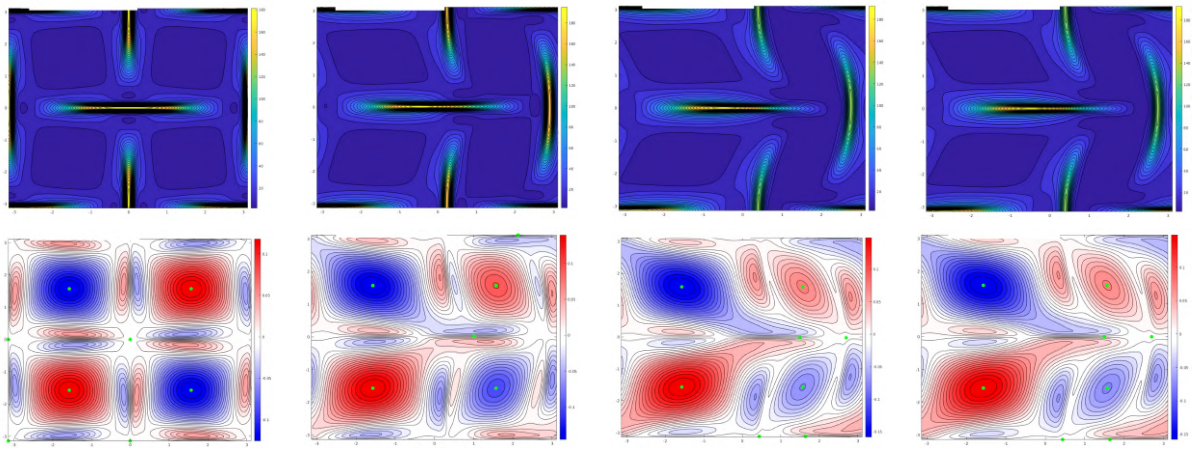
The GPU implementation is coded correctly if simulations over these parameters/conditions yeild the same long-time dynamic behavior at each Wi. Otherwise, there is a problem with the implementation.

Verifying the results of the GPU implementation

Wi = 6

For Wi = 6, we expect to observe the system evolve from a 2-symmetry state to a steady state with 1-symmetry. Additionally, we expect this evolution to occur gradually through small decaying oscillations which temporarily break the 1-symmetry.

Figure 2 shows our observed evolution at various points of time, starting with the expected 2-symmetry state in Figure 2(a) and ending with the expected 1-symmetry state in Figure 2(d).

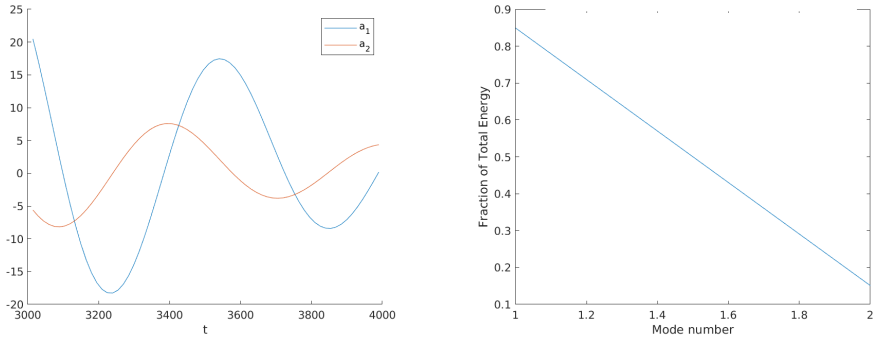


(a) t=510 (b) t=2220 (c) t=2505 (d) t=5985

Figure 2: Time evolution of Wi = 6. tr$\mathbf{S}$ in the first row and vorticity superimposed with the stagnation points in the second row.

The temporal modes in Figure 3(a) illustrate that a decaying oscillation is occurring within the

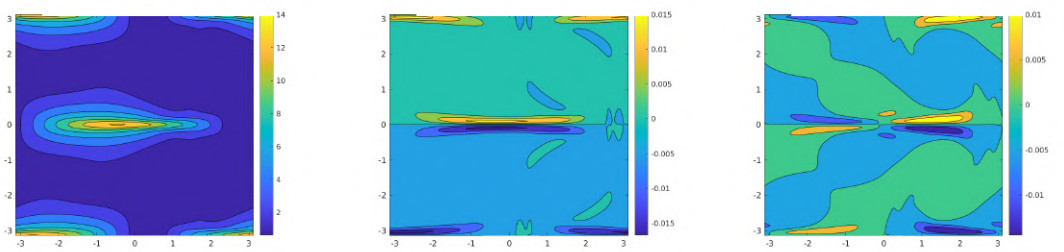
system, although referencing the spatial modes will be necessary to determine whether or not these oscillations are related to the periodic breaking of the remaining 1-symmetry.



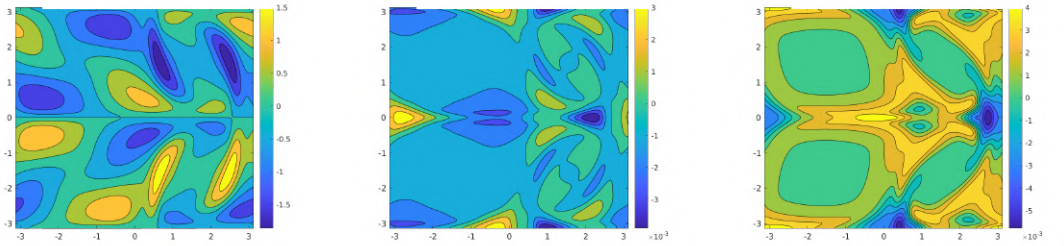
(a) Temporal modes a_1 and a_2 (b) Energy distribution across modes

Figure 3: (a) Temporal modes and (b) energy distribution from the POD for Wi = 6 over times between $t = 3000 - 4005$, with $\Delta t = 15$.

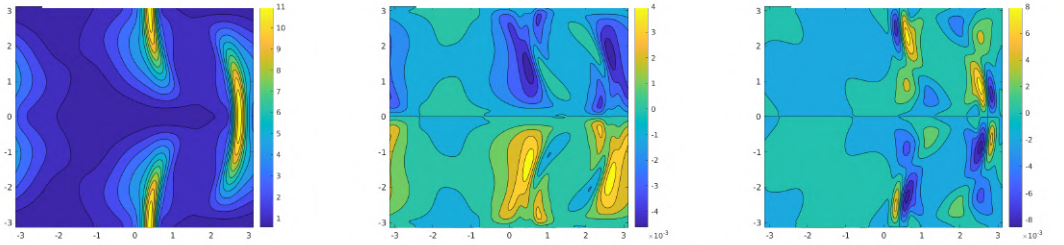
We can observe notable anti-symmetry in the spatial modes for components b_{11} and b_{22} in Figure 4(b), (c), (h), and (i). When these components are multiplied with their respective temporal modes, summed together with symmetric components, and squared, they create the expected time-oscillating asymmetry which periodically breaks the remaining 1-symmetry.



(a) b_{11} temporal mean (b) b_{11} mode 1 (c) b_{11} mode 2



(d) b_{12} temporal mean (e) b_{12} mode 1 (f) b_{12} mode 2



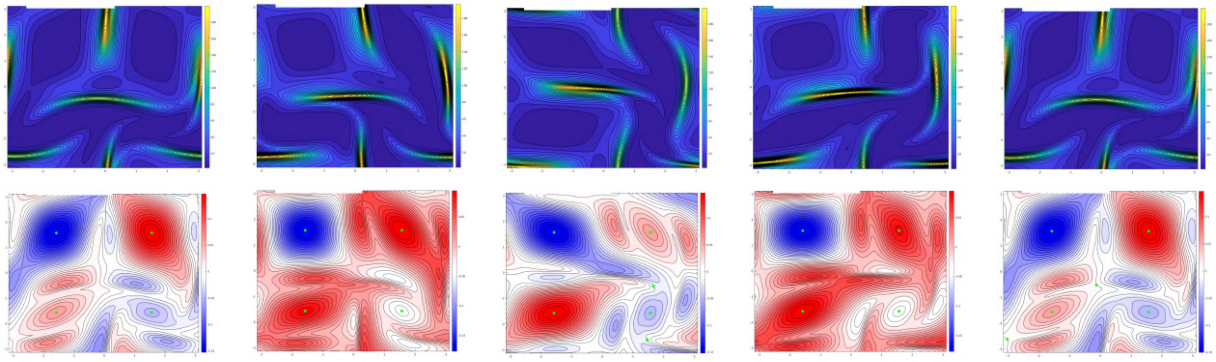
(g) b_{22} temporal mean (h) b_{22} mode 1 (i) b_{22} mode 2

Figure 4: Geometric modes obtained from the POD for Wi = 6 for times between $t = 3000 - 4005$, with $\Delta t = 15$.

Therefore, we find that the GPU implementation's Wi = 6 case behaves the same as that of the CPU implementation in [3].

Wi = 7

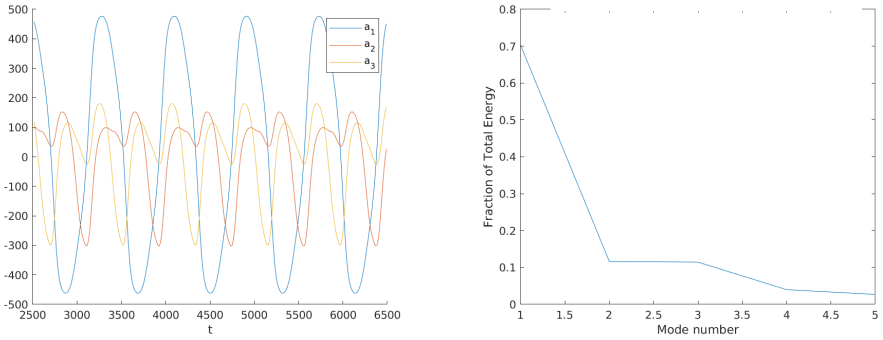
For Wi = 7, we expect to observe the system evolve from a 2-symmetry state to a periodic oscillatory state, characterized by oscillations between near up-down symmetry and near left-right symmetry. This behavior occurs and the characteristic oscillation is observed as in Figure 5, most clearly visualized by the vorticity.



(a) $t = t_0$ (b) $t = t_0 + \frac{T}{4}$ (c) $t = t_0 + \frac{T}{2}$ (d) $t = t_0 + \frac{3T}{4}$ (e) $t = t_0 + T$

Figure 5: Time evolution of Wi = 7. tr$\mathbf{S}$ in the first row and vorticity superimposed with the stagnation points in the second row.

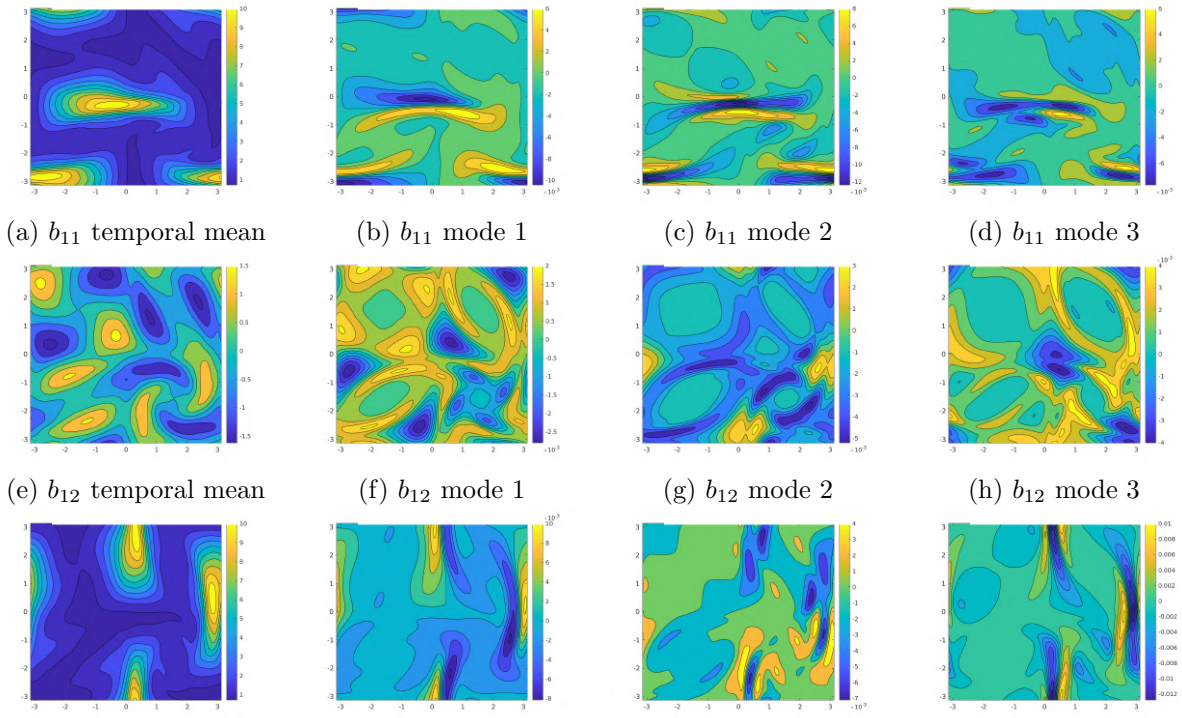
To further confirm that this is the expected behavior and not alternative phenomena which only looks visually similar to the expected behavior, we can compare the POD data of the GPU implementation and the CPU implementation of [3]. Indeed, the profiles of the temporal modes in Figure 6(a) and the energy distribution in Figure 6(b) are nearly identical to those in the CPU implementation.



(a) Temporal modes a_1 - a_3 (b) Energy distribution across modes

Figure 6: (a) Temporal modes and (b) energy distribution from the POD for Wi = 7 using 5 periods $T \approx 800$, with $\Delta t = 15$.

Furthermore, noting the swapped x-y axis, the forms of the spatial modes in Figure 7 are also nearly identical to those collected in the CPU implementation.



(a) b_{11} temporal mean (b) b_{11} mode 1 (c) b_{11} mode 2 (d) b_{11} mode 3

(e) b_{12} temporal mean (f) b_{12} mode 1 (g) b_{12} mode 2 (h) b_{12} mode 3

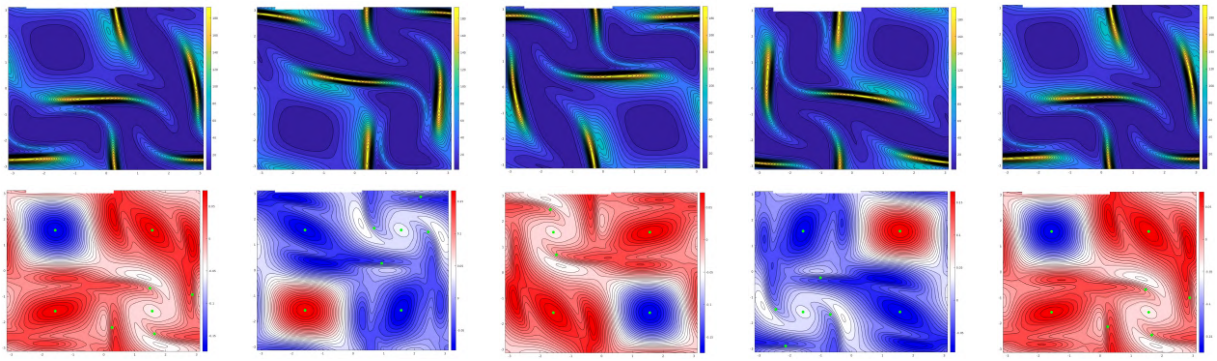
(i) b_{22} temporal mean (j) b_{22} mode 1 (k) b_{22} mode 2 (l) b_{22} mode 3

Figure 7: Geometric modes obtained from the POD for Wi = 7 using 5 periods $T \approx 800$, with $\Delta t = 15$.

Therefore, we find that the GPU implementation's Wi = 7 case behaves the same as that of the CPU implementation in [3].

Wi = 8

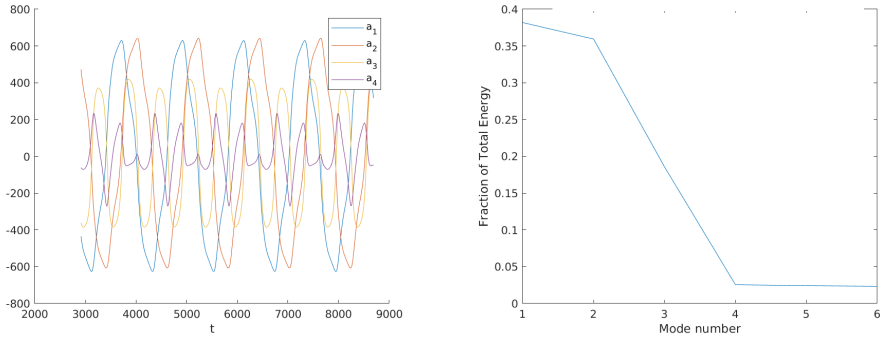
For Wi = 8, we expect to observe the system evolve from a 2-symmetry state to a periodic "loop" state, characterized by a dominant vortex "looping" around all 4-rolls rotating clockwise (for traditional y-up, x-right axis). This behavior occurs and the characteristic loop is observed as in Figure 8, most clearly visuallized by the vorticity. Note that the xy axis have been swapped relative to those in [3], so the observed counter-clockwise rotation is expected.



(a) $t = t_0$ (b) $t = t_0 + \frac{T}{4}$ (c) $t = t_0 + \frac{T}{2}$ (d) $t = t_0 + \frac{3T}{4}$ (e) $t = t_0 + T$

Figure 8: Time evolution of Wi = 8. tr**S** in the first row and vorticity superimposed with the stagnation points in the second row.

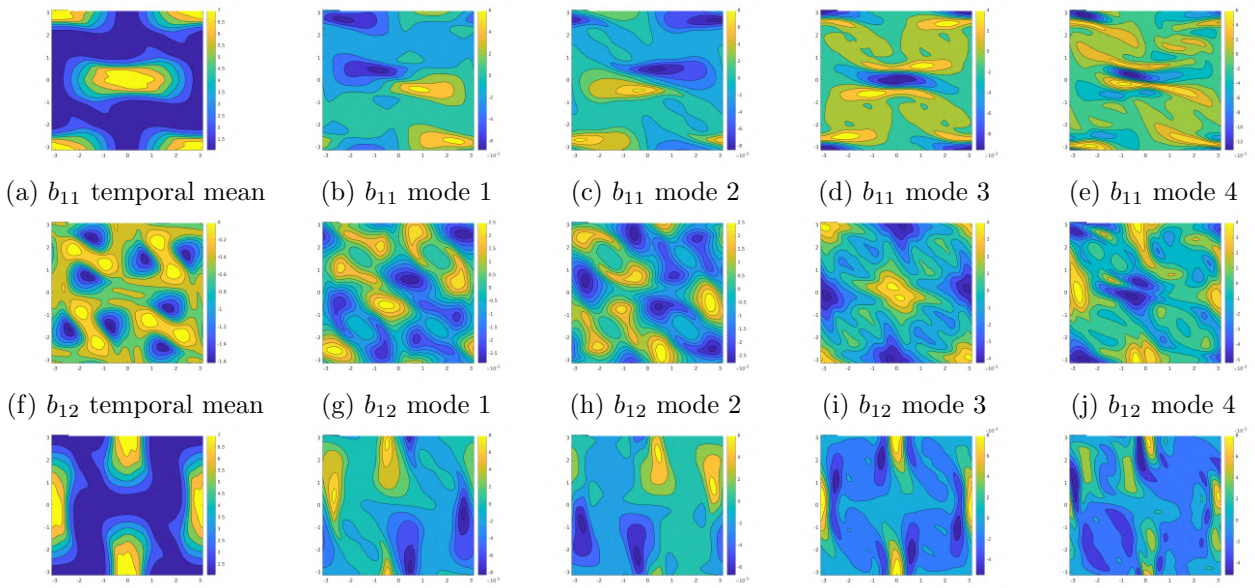
Similar to the Wi = 7 case, we further confirm that this is the expected behavior by comparing the POD data of the GPU implementation and the CPU implementation of [3]. Indeed, the profiles of the temporal modes in Figure 9(a) and the energy distribution in Figure 9(b) are nearly identical to those in the CPU implementation.



(a) Temporal modes a_1 - a_3 (b) Energy distribution across modes

Figure 9: (a) Temporal modes and (b) energy distribution from the POD for Wi = 8 using 5 periods $T \approx 1200$, with $\Delta t = 15$.

Furthermore, noting the swapped x-y axis, the forms of the spatial modes in Figure 7 are also nearly identical to those collected in the CPU implementation.



(a) b_{11} temporal mean (b) b_{11} mode 1 (c) b_{11} mode 2 (d) b_{11} mode 3 (e) b_{11} mode 4

(f) b_{12} temporal mean (g) b_{12} mode 1 (h) b_{12} mode 2 (i) b_{12} mode 3 (j) b_{12} mode 4

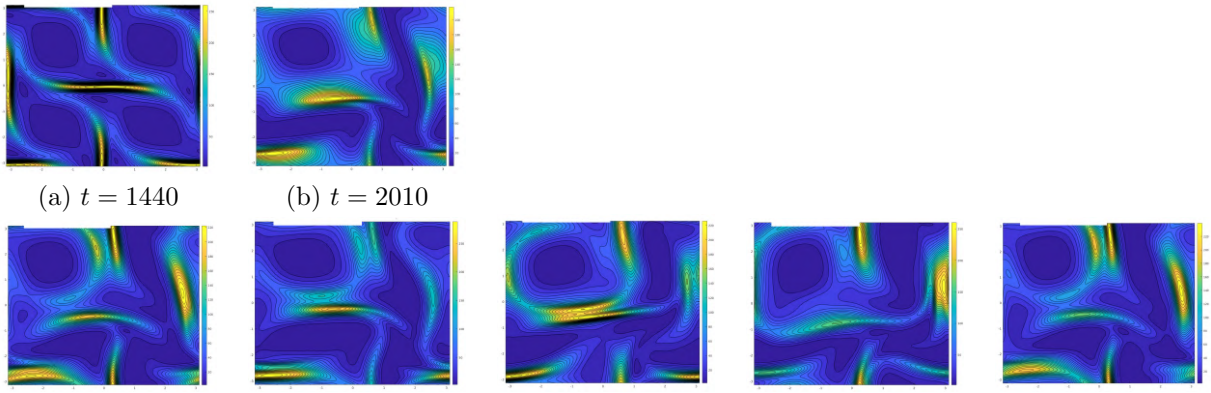
(k) b_{22} temporal mean (l) b_{22} mode 1 (m) b_{22} mode 2 (n) b_{22} mode 3 (o) b_{22} mode 4

Figure 10: Geometric modes obtained from the POD for Wi = 8 using 5 periods $T \approx 1200$, with $\Delta t = 15$.

Therefore, we find that the GPU implementation's Wi = 8 case behaves the same as that of the CPU implementation in [3].

Wi = 10

For Wi = 10, we expect to observe the system evolve from a 2-symmetry state, to a periodic oscillatory state, like that of Wi = 7, and finally to an aperiodic state characterized by a single dominant vortex stretching and contracting in quick oscillations. This behavior occurs and is observed as shown in Figure 10.

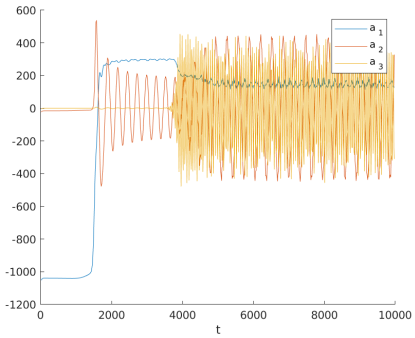


(a) $t = 1440$ (b) $t = 2010$

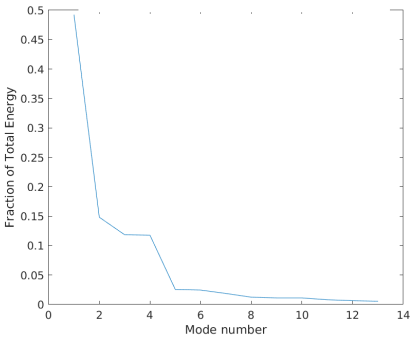
(c) $t = t_0$ (d) $t = t_0 + \frac{T}{4}$ (e) $t = t_0 + \frac{T}{2}$ (f) $t = t_0 + \frac{3T}{4}$ (g) $t = t_0 + T$

Figure 10: Time evolution of tr$\mathbf{S}$ for Wi = 10.

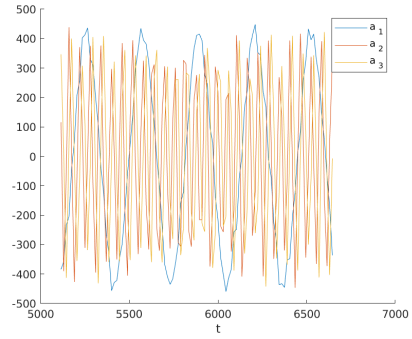
Similarly to the other cases, we can further confirm that this is the expected behavior by comparing the POD data of the GPU implementation and the CPU implementation of [3]. Indeed, the profiles of the temporal modes in Figure 11(a) and (c) and the energy distributions in Figure 11(b) and (d) are nearly identical to those in the CPU implementation.



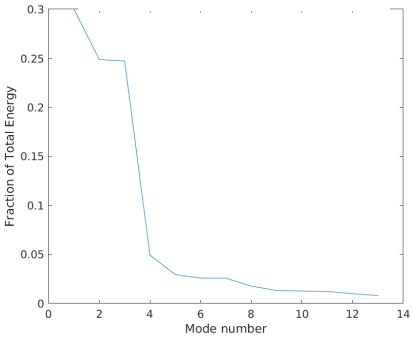
(a) Temporal modes a_1 - a_3



(b) Energy distribution across modes



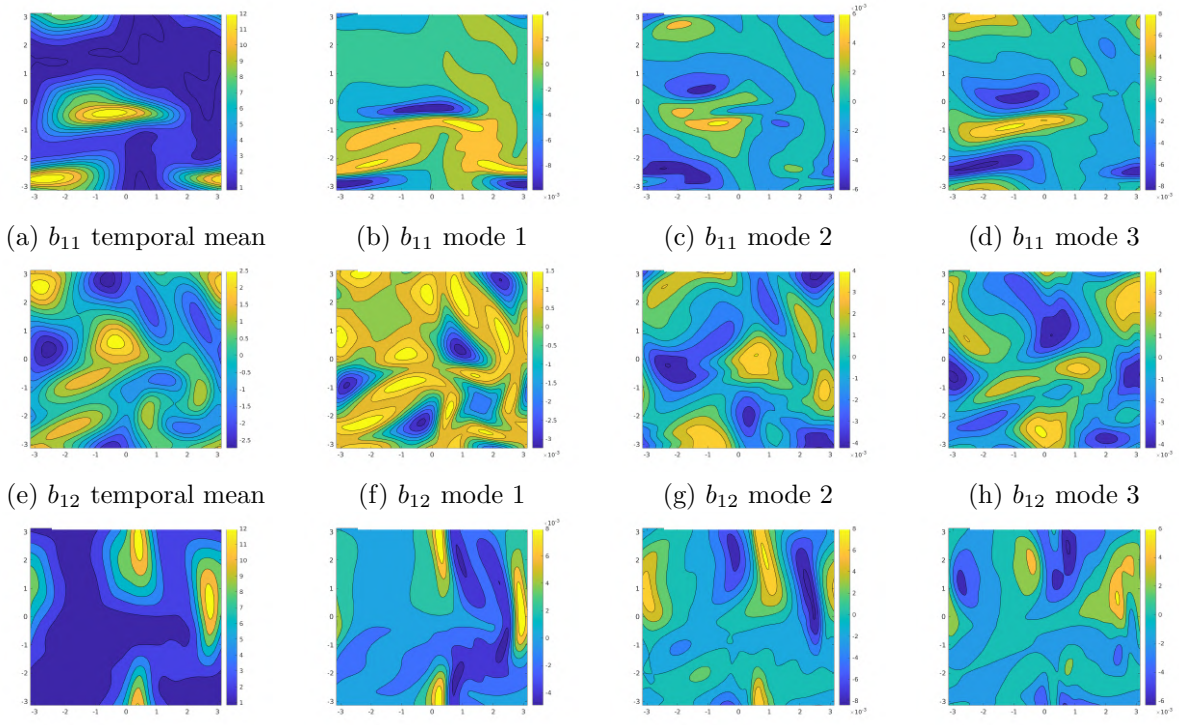
(c) Temporal modes a_1 - a_3



(d) Energy distribution across modes

Figure 11: Features obtained from the POD for Wi = 10. (a) Temporal modes and (b) energy distribution from the POD over t=0-10000, with $\Delta t = 15$. (c) Temporal modes and (d) energy distribution from the POD over 5 periods $T \approx 312$, with $\Delta t = 5$.

Furthermore, noting the swapped x-y axis, the forms of the spatial modes in Figure 12 are also nearly identical to those collected in the CPU implementation.



(a) b_{11} temporal mean (b) b_{11} mode 1 (c) b_{11} mode 2 (d) b_{11} mode 3

(e) b_{12} temporal mean (f) b_{12} mode 1 (g) b_{12} mode 2 (h) b_{12} mode 3

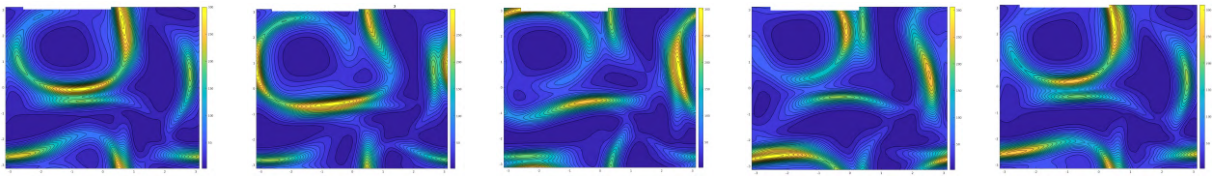
(i) b_{22} temporal mean (j) b_{22} mode 1 (k) b_{22} mode 2 (l) b_{22} mode 3

Figure 12: Geometric modes obtained from the POD for Wi = 10 over 5 periods $T \approx 312$, with $\Delta t = 5$.

Therefore, we find that the GPU implementation's Wi = 10 case behaves the same as that of the CPU implementation in [3].

Wi = 12

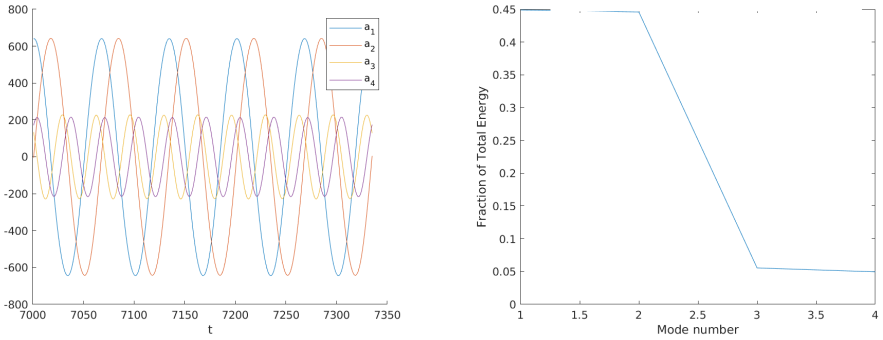
For Wi = 12, we expect to observe the system evolve from a 2-symmetry state, to an unsteady asymmetric steady state, and finally to a periodic "dominant vortex" state characterized by a single dominant vortex stretching and contracting in quick oscillations similar to the Wi = 10 case. Unlike the Wi = 10 case, these oscillations of the dominant vortex are synchronized with the other oscillatory behavior within the system, which removes the aperiodicity present in the Wi = 10. This behavior occurs and is observed as shown in Figure 13.



(a) $t = t_0$ (b) $t = t_0 + \frac{T}{4}$ (c) $t = t_0 + \frac{T}{2}$ (d) $t = t_0 + \frac{3T}{4}$ (e) $t = t_0 + T$

Figure 13: Time evolution of of tr$\mathbf{S}$ for Wi = 12

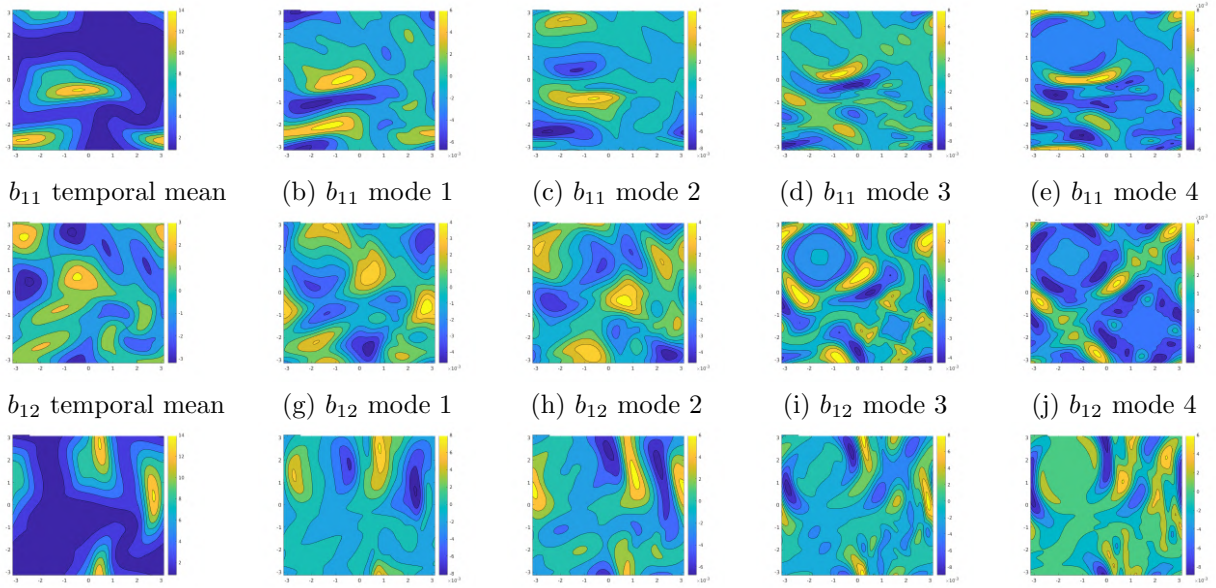
Similarly to the other cases, we can further confirm that this is the expected behavior by comparing the POD data of the GPU implementation and the CPU implementation of [3]. Indeed, the profiles of the temporal modes in Figure 14(a) and the energy distributions in Figure 14(b) are nearly identical to those in the CPU implementation.



(a) Temporal modes a_1 - a_3 (b) Energy distribution across modes

Figure 14: (a) Temporal modes and (b) energy distribution from the POD for Wi = 12 using 5 periods $T \approx 66$, with $\Delta t = 1$.

Furthermore, noting the swapped x-y axis, the forms of the spatial modes in Figure 15 are also nearly identical to those collected in the CPU implementation.



(a) b_{11} temporal mean (b) b_{11} mode 1 (c) b_{11} mode 2 (d) b_{11} mode 3 (e) b_{11} mode 4

(f) b_{12} temporal mean (g) b_{12} mode 1 (h) b_{12} mode 2 (i) b_{12} mode 3 (j) b_{12} mode 4

(k) b_{22} temporal mean (l) b_{22} mode 1 (m) b_{22} mode 2 (n) b_{22} mode 3 (o) b_{22} mode 4

Figure 15: Geometric modes obtained from the POD for Wi = 12 using 5 periods $T \approx 66$, with $\Delta t = 1$.

Therefore, we find that the GPU implementation's Wi = 12 case behaves the same as that of the CPU implementation in [3].

Conclusion

We can conclude that the GPU implementation of the Stokes-Oldroyd-B simulations successfully reproduces the results documented previously by [3], in nearly ten times less total simulation time. Speed-ups of this magnitude open up the possibility of larger or more traditionally expensive simulations, such as a parameter exploration in the three-dimensional regime of the system.

Acknowledgments

First, I would like to thank Dr. Paloma Gutierrez-Castillo for giving me the opportunity to write this senior thesis under her supervision. I'd also like to thank Dr. Gutierrez-Castillo for amazing patience and guidance throughout the writing of this thesis. I'd also like to thank Dr. Becca Thomases for additional support and guidance in writing this thesis, as well as Dr. Robert Guy for introducing me to this project.

Bibliography

[1] Uri M Ascher, Steven J Ruuth, and Brian TR Wetton. Implicit-explicit methods for time-dependent partial differential equations. *SIAM Journal on Numerical Analysis*, 32(3):797–823, 1995.

[2] Richard L Burden and J Douglas Faires. Numerical analysis. 2001.

[3] Paloma Gutierrez-Castillo and Becca Thomases. Proper orthogonal decomposition (pod) of the flow dynamics for a viscoelastic fluid in a four-roll mill geometry at the stokes limit. *Journal of Non-Newtonian Fluid Mechanics*, 264:48–61, 2019.

[4] Robert D Guy and Becca Thomases. Computational challenges for simulating strongly elastic flows in biology. In *Complex fluids in biological systems*, pages 359–397. Springer, 2015.

[5] Boyang Qin, Paul F Salipante, Steven D Hudson, and Paulo E Arratia. Upstream vortex and elastic wave in the viscoelastic flow around a confined cylinder. *Journal of Fluid Mechanics*, 864, 2019.

[6] Kunihiko Taira, Steven L Brunton, Scott TM Dawson, Clarence W Rowley, Tim Colonius, Beverley J McKeon, Oliver T Schmidt, Stanislav Gordeyev, Vassilios Theofilis, and Lawrence S Ukeiley. Modal analysis of fluid flows: An overview. *Aiaa Journal*, pages 4013–4041, 2017.