Project Report
On
GPU based Fluid Simulation

Submitted
In partial fulfillment
For the award of the Degree of

PG-Diploma in High Performance Computing
Application Programming
(C-DAC, ACTS (Pune))

Guided By:

Parikshit Sir

Submitted By:

Abhishesk Raj (230940141001)
Chandan Kumar (230940141004)
Praveen Kr. Paswan (230940141016)
Purnima Kumari (230940141018)
Prasoon Bala (230940141015)
Shivanadan Kr. Shorya (230940141023)

Centre for Development of Advanced Computing
(C-DAC), ACTS (Pune- 411008)

Abstract

In this project, we explore the implementation and optimization of the Lattice Boltzmann method (LBM) for fluid dynamics simulation using parallel computing techniques. The lattice Boltzmann method is a powerful computational tool for simulating fluid flows based on microscopic particle dynamics. By discretizing the fluid domain into a lattice and simulating the evolution of particle distributions according to simple collision and streaming rules, LBM offers advantages in handling complex geometries and boundary conditions.

Our work focuses on optimizing the performance of LBM simulations through parallel computing using both PyTorch and CuPy libraries. PyTorch provides a flexible framework for designing and optimizing neural networks, while CuPy leverages the power of CUDA-enabled GPUs for accelerated numerical computations. By leveraging parallel computing techniques, we aim to achieve significant performance improvements in LBM simulations compared to traditional serial implementations.

We present a detailed analysis of our parallel implementations, including performance benchmarks and comparisons with serial implementations. Our results demonstrate substantial speedup achieved through parallelization, enabling faster and more efficient simulations of fluid dynamics phenomena. Additionally, we discuss the challenges and considerations involved in parallelizing LBM simulations, such as load balancing, memory management, and optimization strategies.

Overall, our study showcases the potential of parallel computing techniques for accelerating LBM simulations, paving the way for more efficient and scalable computational fluid dynamics applications in various fields, including engineering, physics, and environmental science.

Table of Contents

Chapter 1

Introduction

1.1 Introduction

Fluids are everywhere: water passing between riverbanks, smoke curling from a glowing cigarette, steam rushing from a teapot, water vapor forming into clouds, and paint being mixed in a can. Underlying all of them is the flow of fluids. All are phenomena that we would like to portray realistically in interactive graphics applications. Figure 1 shows examples of fluids simulated using the source code provided with this book.

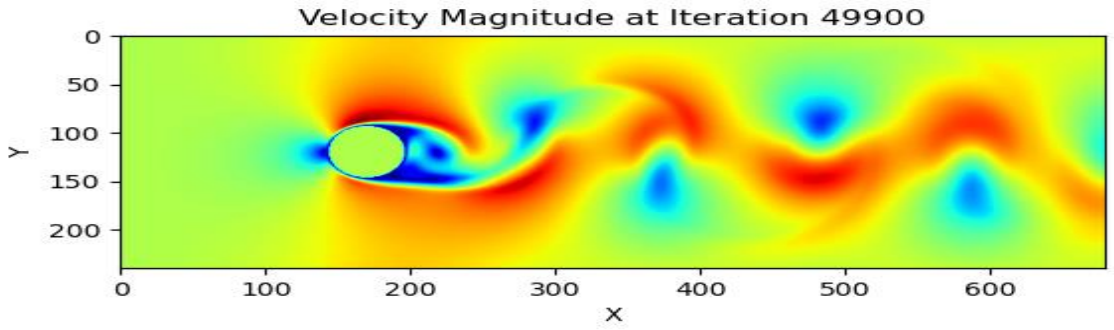


Fig 1. Colormap: ‘jet’ Carried by a Swirling Fluid

Fluid simulation is a useful building block that is the basis for simulating a variety of natural phenomena. Because of the large amount of parallelism in graphics hardware, the simulation we describe runs significantly faster on the GPU than on the CPU. Using NVIDIA Quadro GV100, we have achieved a speedup of up to six times over an equivalent CPU simulation.

1.2 Background

The lattice Boltzmann method (LBM) is a powerful computational technique used for simulating fluid flows at various scales, ranging from microfluidics to large-scale industrial applications. Unlike traditional computational fluid dynamics (CFD) methods, such as finite difference or

finite volume methods, which solve the Navier-Stokes equations directly, LBM simulates fluid flow by modeling the distribution of particle populations on a lattice grid.

One of the key advantages of LBM is its inherent simplicity and parallelizability, making it well-suited for implementation on parallel computing architectures like GPUs. In LBM, the fluid dynamics are governed by a set of discrete kinetic equations, known as the lattice Boltzmann equation (LBE), which describes how the particle populations evolve in space and time.

The core idea behind LBM is to divide the simulation domain into a regular grid of cells, with each cell representing a discrete velocity direction on the lattice. The distribution of particle populations at each cell is updated according to collision and streaming processes, allowing the simulation to capture the macroscopic behavior of the fluid.

While LBM offers several advantages, including ease of implementation and scalability, achieving high-performance simulations for real-world applications often requires efficient utilization of computational resources. With the increasing availability of GPU hardware and the development of GPU-accelerated computing frameworks like CUDA and OpenCL, there is growing interest in leveraging GPU acceleration to enhance the performance of LBM simulations.

In this report, we explore the potential of GPU acceleration in accelerating lattice Boltzmann simulations. We investigate various optimization strategies and parallelization techniques tailored to GPU architectures to achieve significant performance improvements. By harnessing the computational power of GPUs, we aim to accelerate the simulation of complex fluid flow phenomena, enabling researchers and engineers to tackle larger and more computationally demanding problems in fluid dynamics.

Our Assumptions

The reader is expected to have at least a college-level calculus background, including a basic grasp of differential equations. An understanding of vector calculus principles is helpful, but not required (we will review what we need). Also, experience with finite difference approximations of derivatives is useful. If you have ever implemented any sort of physical simulation, such as projectile motion or rigid body dynamics, many of the concepts we use will be familiar.

Our Approach

Firstly, the team embarked on comprehensive research into various fluid simulation algorithms, including Smoothed Particle Hydrodynamics (SPH), Lattice Boltzmann Method (LBM), Finite Difference Method (FDM), and others. Each member was tasked with studying a specific

algorithm in-depth to understand its underlying principles, advantages, and limitations. This research phase, led by you, laid the groundwork for informed decision-making regarding the selection of the most suitable algorithm for the project.

Upon completing the research phase, the team, under your direction, collectively evaluated the pros and cons of each algorithm. After careful consideration, it was determined that the Lattice Boltzmann Method (LBM) would be the most appropriate choice for several reasons. LBM's inherent simplicity, scalability, and compatibility with parallel computing architectures made it an ideal candidate, aligning well with the project's objectives and time constraints.

Following the decision to proceed with LBM, you took the lead in writing the initial serial implementation of the LBM code. This involved translating the theoretical concepts of LBM into practical code, ensuring accuracy and efficiency in the simulation process. The serial code served as a baseline for subsequent optimization efforts and provided valuable insights into the intricacies of the algorithm.

Simultaneously, I focused on exploring parallelization techniques and GPU porting strategies to leverage the computational power of modern graphics processing units (GPUs). By harnessing parallel computing frameworks such as PyTorch and CuPy, we aimed to accelerate the simulation process and achieve significant performance improvements over the serial implementation.

The parallelization and GPU porting phase involved extensive experimentation and iteration, led by both of us collaboratively. We encountered various challenges along the way, including data management, memory optimization, and synchronization issues. Through collaborative problem-solving and experimentation, we developed efficient parallel LBM implementations tailored for GPU execution.

Throughout the process, we emphasized the importance of profiling and benchmarking to evaluate the performance of each implementation accurately. Using profiling tools and performance metrics, we systematically analyzed the execution times, memory usage, and scalability of the parallel LBM codes. This data-driven approach provided valuable insights into the strengths and weaknesses of each implementation, guiding further optimization efforts.

Chapter 2
Literature Reviews and Background Study

Fluid dynamics, the study of fluid motion and its effects on various physical phenomena, underpins a broad spectrum of scientific and engineering applications. From understanding weather patterns and ocean currents to optimizing the design of aircraft and automobiles, the ability to simulate fluid flow accurately is essential for making informed decisions and advancing technological innovations.

Traditionally, fluid dynamics simulations were conducted using continuum-based methods such as finite difference, finite volume, and finite element methods, which solve the Navier-Stokes equations directly. While these approaches are widely used and well-established, they often require complex mesh generation and suffer from computational inefficiencies, especially for simulating complex flows with turbulent behavior and multiphase interactions.

In contrast, the lattice Boltzmann method (LBM) offers a distinct computational paradigm for simulating fluid flow on discrete grids, inspired by the lattice gas automata framework and the kinetic theory of gases. In LBM, the fluid dynamics equations are reformulated in terms of kinetic particle distributions propagating and colliding on a lattice grid, enabling the simulation of fluid behavior at a mesoscopic level.

One of the key advantages of LBM is its simplicity and modularity, which make it well-suited for implementation on parallel computing architectures. LBM simulations can be readily parallelized across multiple processing units, including multicore CPUs and graphics processing units (GPUs), allowing for efficient utilization of computational resources and scalability to large-scale systems.

The parallelizability of LBM has led to significant interest in accelerating its simulations using GPU computing, owing to the highly parallel nature of GPUs and their superior computational performance compared to traditional CPUs. GPU-accelerated LBM implementations have demonstrated remarkable speedups over their CPU counterparts, enabling researchers to tackle increasingly complex fluid dynamics problems with greater efficiency.

Several studies in the literature have explored various aspects of GPU-accelerated LBM, including optimization techniques, parallelization strategies, and application-specific adaptations. These studies have investigated the impact of factors such as lattice configuration, boundary conditions, and memory access patterns on the performance and scalability of GPU-based LBM simulations.

Furthermore, the availability of programming frameworks such as CUDA and OpenCL has facilitated the development of GPU-accelerated LBM codes, providing researchers with

powerful tools for harnessing the computational capabilities of modern GPU architectures. These frameworks offer high-level abstractions and optimized libraries for parallel computing, streamlining the development process and enabling rapid prototyping of GPU-accelerated simulations.

2.1 Connection with FD and the Navier-Stokes Equations

The Lattice Boltzmann method can be viewed as a special finite difference method [7] for solving the Boltzmann transport equation (1) on a lattice. To see this, we write down the Boltzmann transport equation in terms of the discrete distribution function,

$$\frac{\partial f_i}{\partial t} + \vec{e}_i \cdot \nabla f_i = \Omega_i \tag{39}$$

If we discretize the differential operator and the collision operator in this form

$$\frac{f_i(\vec{x}, t+\Delta t) - f_i(\vec{x}, t)}{\Delta t} + \frac{f_i(\vec{x}+\vec{e}_i\Delta x, t+\Delta t) - f_i(\vec{x}, t+\Delta t)}{\Delta x} = -\frac{f_i(\vec{x}, t) - f_i^{eq}(\vec{x}, t)}{\tau} \tag{40}$$

and assume that $\Delta x = \Delta t = 1$, we essentially recover the LBM evolution equation (5).

The primary reason why LBM can serve as a method for fluid simulations is that the Navier-Stokes equations can be recovered from the discrete equations through the Chapman-Enskog procedure, a multi-scaling expansion technique. Specifically the ID2Q9 model used in simulating Rayleigh-B´enard convection is able to recover the incompressible Navier-Stokes equations. Here we highlight the key steps. A detailed derivation is provided in the appendix of [1, 8].

With a multi-scaling expansion, we may write

$$g_i = g_i^{(0)} + \epsilon g_i^{(1)} + \epsilon^2 g_i^{(2)} + \dots \tag{41}$$

$$\frac{\partial}{\partial t} = \epsilon \frac{\partial}{\partial t_1} + \epsilon^2 \frac{\partial}{\partial t_2} + \dots \tag{42}$$

$$\frac{\partial}{\partial x} = \epsilon \frac{\partial}{\partial x_1} \tag{43}$$

where g (0) i = g eq i and = Δt is the expansion parameter. It is proved in [6] that up to O() we can derive the following continuity and momentum equations,

$$\nabla \cdot \vec{u} = 0 + O(\epsilon) \tag{44}$$

$$\frac{\partial \vec{u}}{\partial t} + \nabla \cdot (\mathbf{\Pi^0}) = 0 + O(\epsilon) \tag{45}$$

where Π0 is the equilibrium flux tensor. And to O(2), the following equations are derived

$$\nabla \cdot \vec{u} = \epsilon \left(\tau - \frac{1}{2}\right) \mathbf{P} + O(\epsilon^2) \quad (46)$$

$$\frac{\partial \vec{u}}{\partial t} + \nabla \cdot (\mathbf{\Pi^0}) = \epsilon \left(\tau - \frac{1}{2}\right) \mathbf{Q} + O(\epsilon^2) \quad (47)$$

where it is shown that $\mathbf{P} \sim O(\epsilon)$ and $\mathbf{Q} \sim O(\epsilon) + O(M^2) + \frac{c^2}{3}\nabla^2\vec{u}$, M is the Mach number. By applying results of $\mathbf{P}$ and $\mathbf{Q}$ to (46) and (47), the continuity equation is derived accurate to $O(\epsilon^2)$ and the momentum equation is derived to $O(\epsilon^2 + \epsilon M^2)$.

$$\nabla \cdot \vec{u} = 0 + O(\epsilon^2) \quad (48)$$

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = -\nabla p + \nu \nabla^2 \vec{u} + O(\epsilon^2 + \epsilon M^2) \quad (49)$$

2.2 Conclusion

In this project, we conducted a comprehensive study of the Lattice Boltzmann method and its corresponding boundary conditions. The validity of this method was verified by comparing numerical solutions to the exact solutions of the steady plane Poiseuille flow. Three nontrivial simulations: liddriven cavity flow, flow past a circular cylinder and Rayleigh-B´enard convection were performed and they agreed closely with physical situations. We drew a connection between LBM and FD and concluded that it is a special discretization of the Boltzmann transport equation. The connection between LBM and Navier-Stokes was not fully worked out in this report, though we did attempt to highlight the important steps.

- The Lattice Boltzmann method has several advantages for fluid simulations over traditional finite difference methods
- LBM is very applicable to simulate multiphase/multicomponent flows. Complex boundaries are much easier to deal with using on-grid bounce-back and thus LBM can be applied to simulate flows with complex geometries such as porous media flows.
- LBM can be easily parallelized and thus can be applied to do large simulations.

However, by reading references on the derivation of the Navier-Stokes equations from LBM, we realize that this method is subject to some compressible effects. Therefore it is expected that some source of errors come in the form of artificial compressibility when solving the incompressible Navier-Stokes equations using LBM.

Chapter 3
Methodology and Techniques

3.1 NVIDIA Graphics Card

A graphics card is a hardware device that connects to the computer motherboard to provide the computer with graphics processing resources. In common parlance, the terms GPU and graphics card are used interchangeably. NVIDIA graphics cards are high-performance devices that enable powerful graphics rendering and processing for use in video editing, video gaming, and other complex computing operations.

One of the advantages of using graphics cards for complex computing tasks is that multiple GPUs can be installed in a single computer, lending a large amount of parallelized, highly efficient processing resources to the computer system. This is particularly beneficial for applications in the fields of cloud computing and artificial intelligence, making NVIDIA graphics cards a valuable resource in modern computing applications.

3.1.1 System Specifications

shavak (0:0)

Target

Hostname	shavak
Local time at t=0	2024-02-19T23:51:38.228+05:30
UTC time at t=0	2024-02-19T18:21:38.228Z
TSC value at t=0	99127007779408
Platform	Linux
OS	CentOS Linux 7 (Core)
Hardware platform	x86_64
Serial number	Local (CLI)
CPU description	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz
GPU descriptions	Quadro GV100
NVIDIA driver version	530.30.02
Max EMC frequency	1.60 GHz
CPU context switch	supported
GPU context switch	supported
Guest VM id	0
Tunnel traffic through SSH	no
Timestamp counter	supported
Profiling session UUID	ed1de775-b123-4711-ae97-d6228d6941a5
Target name	shavak

For additional comparison we perform test runs on the Intel quad-core CPU in the system housing the GPU, the specifications of which can be found below...

3.2 GPU Specifications

Quadro GV100

Chip Name	GV100
SM Count	80
L2 Cache Size	6.00 MiB
Memory Bandwidth	810.62 GiB/s
Memory Size	31.74 GiB
Core Clock	1.63 GHz
Bus Location	0000:d8:00.0
UUID	08b19248-96e1-0d23-0443-91e4bd6d65e4
GSP firmware version	N/A

3.3 CuPy

CuPy is an open-source library that provides GPU-accelerated computing in Python, based on the CUDA platform developed by NVIDIA. It offers an interface similar to NumPy, a popular library for numerical computing in Python, making it easy for users familiar with NumPy to transition to GPU-accelerated computing without extensive code modifications. Here's a detailed overview of CuPy.

3.3.1 Basic Features:

GPU Acceleration - CuPy utilizes the parallel processing power of NVIDIA GPUs to accelerate numerical computations in Python.It provides GPU-accelerated implementations of many NumPy functions, allowing users to perform computations on arrays directly on the GPU.

CuPy aims to be compatible with NumPy, providing a similar interface and functionality.Many functions and methods in CuPy have the same names and behavior as their NumPy counterparts, making it easy for users to switch between the two libraries. CuPy supports a wide range of array operations, including arithmetic operations, linear algebra routines, statistical functions, random number generation, and element-wise operations. Users can perform computations on multi-dimensional arrays using familiar NumPy syntax. CuPy provides functions for allocating,

copying, and managing memory on the GPU. It supports both device memory (allocated on the GPU) and pinned memory (allocated in host memory but accessible to the GPU), allowing for efficient data transfer between the CPU and GPU.

3.4 Numba

Numba is an open-source just-in-time (JIT) compiler for Python that translates Python code into optimized machine code for execution on CPUs and GPUs. Developed by Anaconda, Inc., Numba aims to accelerate numerical computations and other performance-critical Python code without the need for manual code optimization or rewriting in lower-level languages like C or CUDA. Here's a detailed overview of Numba. It dynamically compiles Python functions at runtime, generating optimized machine code tailored to the target hardware architecture. It analyzes the Python code and translates it into highly efficient LLVM intermediate representation (IR) code for execution. Numba provides a simple decorator-based interface for annotating Python functions that should be JIT-compiled. By adding the `@jit` decorator to a function definition, users instruct Numba to compile the function for improved performance. Numba seamlessly integrates with NumPy, a popular library for numerical computing in Python. It automatically optimizes NumPy array operations and mathematical functions, accelerating numerical computations without code modifications.

Numba supports both CPU and GPU acceleration, allowing users to leverage parallel computing resources for performance gains. It provides decorators such as `@jit` for CPU acceleration and `@cuda.jit` for GPU acceleration, enabling efficient utilization of multi-core CPUs and NVIDIA GPUs. The Numba compiler analyzes Python code, applies optimizations, and generates machine code for execution. It leverages the LLVM compiler infrastructure for generating optimized machine code across various CPU and GPU architectures. Numba includes a CUDA compiler for translating Python code into CUDA kernels for execution on NVIDIA GPUs. It provides decorators such as `@cuda.jit` for annotating functions that should be compiled into CUDA kernels, enabling GPU acceleration. Numba utilizes the LLVM backend for generating optimized machine code from the LLVM intermediate representation (IR) generated by the Numba compiler. LLVM optimizes the IR code and generates machine code specific to the target CPU or GPU architecture. Numba automatically parallelizes NumPy array operations and mathematical functions, speeding up computations on multi-core CPUs and GPUs without additional code changes.

Numba accelerates numerical computations and performance-critical Python code, leading to significant performance improvements over pure Python implementations. It's decorator-based interface makes it easy to accelerate Python functions and integrate with existing codebases without significant modifications. It seamlessly integrates with NumPy, accelerating NumPy array operations and mathematical functions without requiring code changes. Numba provides support for GPU acceleration using NVIDIA GPUs, enabling users to leverage parallel computing resources for even greater performance gains. It may not support all Python language features and libraries, especially those involving dynamic code execution or complex Python constructs. There may be an overhead associated with JIT compilation, especially for small functions or code snippets that are executed infrequently.

Numba accelerates numerical computations, scientific simulations, machine learning algorithms, and data analytics tasks in Python. Numba is used in high-performance computing (HPC)

applications, computational fluid dynamics (CFD), finite element analysis (FEA), and other performance-critical simulations. Numba enables GPU acceleration for Python code, accelerating CUDA kernels and GPU-accelerated algorithms in scientific computing and machine learning. It is community will continue to grow, with contributions from developers, researchers, and users, leading to new features, improvements, and use cases for JIT compilation and GPU acceleration in Python.

Numba provides a powerful and flexible solution for accelerating Python code, enabling users to achieve significant performance improvements for numerical computations and parallel code without sacrificing ease of use or compatibility with existing Python libraries and frameworks. As Numba continues to evolve and gain adoption, it will play an increasingly important role in scientific computing, data analytics, machine learning, and high-performance computing workflows.

3.5 PyTorch:

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR). It is primarily used for deep learning applications such as neural networks, natural language processing (NLP), computer vision, and reinforcement learning. PyTorch provides a flexible framework for building and training deep learning models with ease, offering features such as dynamic computation graphs, automatic differentiation, GPU acceleration, and a rich ecosystem of tools and libraries.

Here's a detailed overview of PyTorch's key features and components:

1. **Dynamic Computation Graphs:** PyTorch adopts a dynamic computational graph approach, which means that the computational graph is built on-the-fly during the execution of the program. This dynamic nature allows for more flexibility and ease of debugging compared to static graph frameworks like TensorFlow. Developers can define and modify the computational graph structure dynamically, making it well-suited for applications with variable-length sequences and dynamic network architectures.

1. **Automatic Differentiation:** Automatic differentiation is a core feature of PyTorch that enables efficient gradient computation for training neural networks. PyTorch'storch.autograd module provides automatic differentiation capabilities, allowing users to compute gradients of tensors with respect to other tensors. This feature simplifies the process of implementing custom loss functions and optimizing model parameters using gradient-based optimization algorithms like stochastic gradient descent (SGD) and Adam.

2. **Tensor Operations and GPU Acceleration:** PyTorch provides a powerful array manipulation library with support for multi-dimensional tensors, similar to NumPy. Tensors in PyTorch can be easily manipulated using a wide range of mathematical

operations and functions, making it easy to perform linear algebra, convolution, activation functions, and other common operations used in deep learning. Moreover, PyTorch seamlessly integrates with CUDA for GPU acceleration, allowing users to harness the computational power of GPUs to accelerate training and inference tasks.

3. **Neural Network Module:** PyTorch's torch.nn module provides a high-level abstraction for building neural network models. It offers pre-defined layers, activation functions, loss functions, and optimization algorithms that can be easily composed to create complex neural network architectures. Users can subclass the nn.Module class to define custom neural network modules and encapsulate trainable parameters. Additionally, PyTorch provides utilities for initializing model parameters, saving and loading models, and transferring models between CPU and GPU.

4. **Data Loading and Preprocessing:** PyTorch includes utilities for efficiently loading and preprocessing data for training and evaluation. The torch.utils.data module provides classes and functions for creating custom data loaders, data augmentation, batching, shuffling, and parallel data loading. PyTorch also supports integration with popular datasets and data formats, making it easy to work with image, text, audio, and video data.

5. **Rich Ecosystem and Community Support:** PyTorch benefits from a vibrant ecosystem of libraries, tools, and resources developed by both the PyTorch team and the community. This ecosystem includes popular libraries such as torchvision for computer vision tasks, torchaudio for audio processing, torchtext for natural language processing, and many others. Additionally, PyTorch has extensive documentation, tutorials, and forums where users can find help, share knowledge, and contribute to the community.

   Overall, PyTorch offers a user-friendly and flexible framework for deep learning research and development, empowering researchers and practitioners to explore new ideas, build cutting-edge models, and deploy machine learning solutions in various domains. Its dynamic nature, intuitive API, and extensive ecosystem make it a popular choice among both beginners and experts in the field of deep learning.

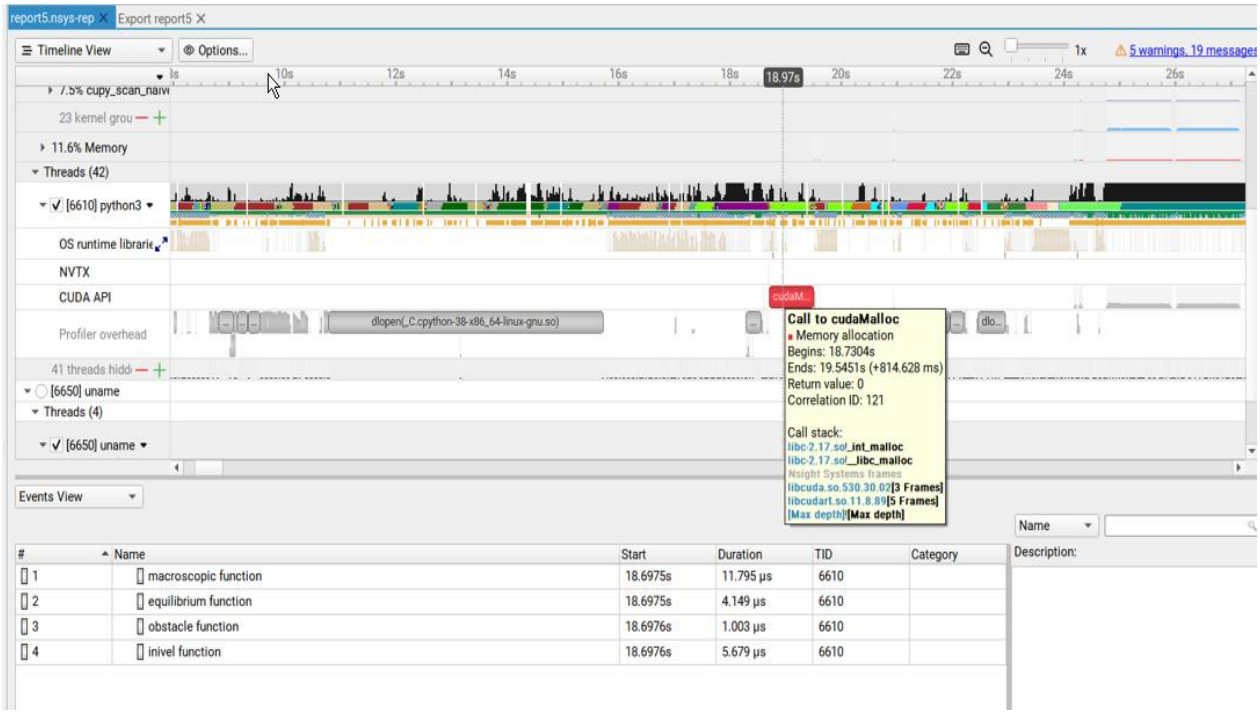
3.6 NVIDIA Nsight System :

NVIDIA Nsight Systems is a comprehensive suite of performance analysis tools designed for profiling, tracing, and optimizing GPU-accelerated applications. It provides detailed insights into the performance characteristics of CUDA, OpenACC, and DirectCompute applications running on NVIDIA GPUs, enabling developers to identify performance bottlenecks, optimize code, and improve overall application efficiency. Here's a detailed overview of NVIDIA Nsight Systems profiling. Nsight Systems enables developers to profile GPU-accelerated applications running on NVIDIA GPUs, providing detailed performance metrics and analysis. It captures information about GPU utilization, kernel execution time, memory usage, PCIe transfers, and other performance-related events. Nsight Systems traces the behavior of GPU-accelerated applications, capturing events related to CUDA kernel launches, memory operations, API calls, and

synchronization primitives. It provides a timeline view of application execution, showing the sequence of events and their duration. Nsight Systems offers a timeline view for visualizing application execution, displaying CPU and GPU activity over time. Developers can analyze the timeline to identify performance bottlenecks, concurrency issues, and synchronization overheads.

Nsight Systems provides performance analysis tools for identifying performance bottlenecks, inefficiencies, and optimization opportunities in GPU-accelerated applications. It offers recommendations for improving performance, optimizing kernel execution, and reducing memory overhead. The Nsight Systems profiler collects performance data during the execution of GPU-accelerated applications, capturing events related to kernel execution, memory access, and API calls. It generates detailed performance reports and analysis for developers to identify performance bottlenecks and optimization opportunities. Nsight Systems provides a timeline view of application execution, showing the sequence of CPU and GPU events over time. Developers can visualize CPU and GPU activity, identify concurrency issues, and analyze performance-critical sections of code. The trace viewer in Nsight Systems allows developers to explore captured traces, inspect individual events, and analyze application behavior. It provides interactive tools for zooming, panning, and filtering the trace data, enabling detailed analysis of GPU-accelerated applications. To profile a GPU-accelerated application using Nsight Systems, developers launch the application with the Nsight Systems profiler enabled. The profiler captures performance data during application execution, generating detailed reports and analysis for optimization. After profiling, developers can analyze the timeline view in Nsight Systems to visualize CPU and GPU activity, identify performance bottlenecks, and optimize application efficiency. They can explore the timeline, zoom in on specific events, and correlate performance metrics to identify optimization opportunities.

The trace viewer in Nsight Systems allows developers to explore captured traces, inspect individual events, and analyze application behavior in detail. Developers can filter events, search for specific events, and view detailed information about kernel launches, memory operations, and API calls. Nsight Systems provides comprehensive profiling of GPU-accelerated applications, capturing detailed performance data and analysis for CPU and GPU activity. It offers insights into kernel execution, memory usage, PCIe transfers, API calls, and synchronization behavior. The timeline view and trace viewer in Nsight Systems provide powerful visualization tools for exploring application behavior, identifying performance bottlenecks, and optimizing code. Nsight Systems offers recommendations and guidance for optimizing GPU-accelerated applications, helping developers improve performance, reduce



overhead, and optimize resource utilization. Nsight Systems may have a steep learning curve for beginners or developers unfamiliar with GPU profiling and performance analysis. Understanding and interpreting the profiling data and metrics may require expertise in GPU architecture, CUDA programming, and performance optimization. Nsight Systems is used for optimizing GPU-accelerated applications in scientific computing, machine learning, computer graphics, and high -performance computing (HPC). Developers leverage Nsight Systems to identify performance bottlenecks, optimize kernel execution, and improve overall application efficiency. It is used for performance tuning, analyzing application behavior, and optimizing GPU-accelerated code for improved performance and efficiency. It will continue to evolve with enhanced analysis tools, visualization capabilities, and performance metrics for deeper insights into GPU application performance. Integration with development environments, build systems, and continuous integration (CI) pipelines will streamline GPU profiling and performance analysis as part of the development workflow. It will add support for new NVIDIA GPU architectures, features, and performance counters, ensuring compatibility and performance analysis for the latest hardware. It provides developers with powerful tools for profiling, tracing, and optimizing GPU-accelerated applications, enabling them to achieve optimal performance and efficiency on NVIDIA GPUs. By leveraging detailed performance metrics, visualization tools, and interactive analysis capabilities, developers can identify performance bottlenecks, optimize code, and improve the efficiency of GPU-accelerated applications across various domains.

Chapter 4
Results

4.1 Results

In the experimentation phase, we evaluated the performance of our fluid simulation code using different implementations. Initially, we ran the serial version of the code, which executed in approximately 35 minutes. Seeking to harness the power of GPU acceleration, we explored two parallel implementations using PyTorch and CuPy libraries.

With PyTorch, we observed a significant improvement in performance, with the execution time reduced to approximately 13 minutes. Leveraging PyTorch's GPU-accelerated tensor operations and automatic differentiation capabilities, we achieved nearly a threefold speedup compared to the serial version.

Subsequently, we experimented with CuPy, an alternative library for GPU-accelerated computing. Our CuPy implementation also demonstrated promising results, achieving a runtime of approximately 11 to 12 minutes. The CuPy implementation, which mirrors the NumPy API and seamlessly integrates with CUDA, provided comparable performance to the PyTorch version, highlighting the efficiency of both libraries in accelerating our fluid simulation code.

Overall, our experimentation with different implementations showcased the effectiveness of leveraging GPU acceleration for improving the performance of our fluid simulation code. Both PyTorch and CuPy offered substantial speedups over the serial version, enabling faster execution and more efficient utilization of computational resources for fluid dynamics simulations.

4.2 Comparison of CPU and GPU versions

Certainly! Here's a detailed performance comparison of the serial, PyTorch, and CuPy implementations:

1. **Serial Implementation:**
    - **Execution Time:** Approximately 35 minutes.
    - **Hardware Utilization:** Utilizes only CPU resources.
    - **Speed:** Limited by sequential processing, resulting in longer execution times.
    - **Scalability:** Limited scalability due to single-threaded execution.
    - **Memory Usage:** Moderate memory consumption due to CPU-based calculations.

```
(project) [user1@shavak fluidSim]$ nsys profile -t nvtx,osrt python3 fSimSerial.py
Torch version: 2.2.0+cu118
cuda
100%|          | 50000/50000 [37:18<00:00, 22.33it/s]time to execute =  2238.8016510009766

Generating '/tmp/nsys-report-726f.qdstrm'
[1/1] [========================100%] report5.nsys-rep
Generated:
    /home/user1/cisco/project/fluidSim/report5.nsys-rep
```

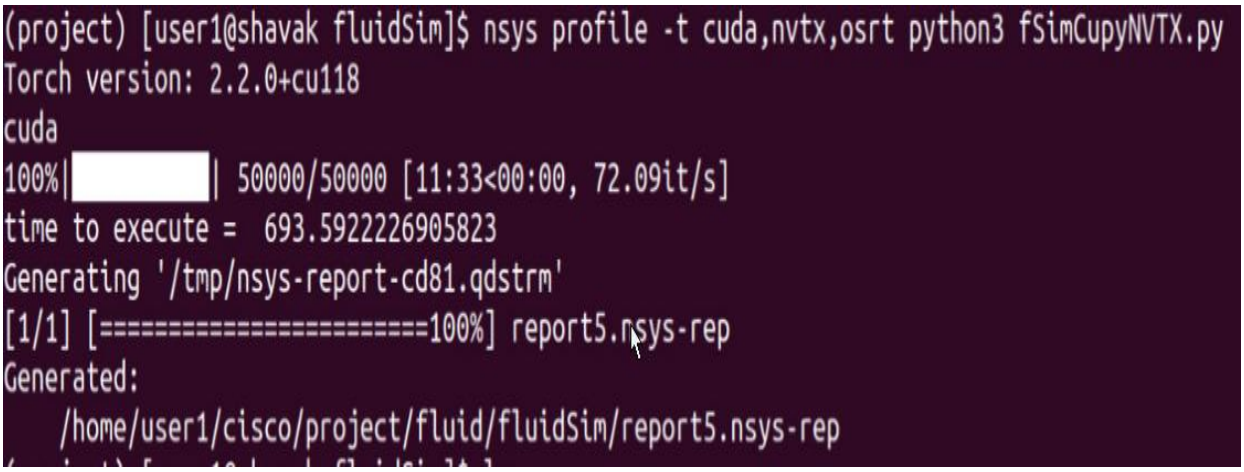
2. **PyTorch Implementation:**

    - **Execution Time:** Reduced to around 13 minutes, representing a threefold improvement over the serial version.
    - **Hardware Utilization:** Utilizes GPU resources through PyTorch's CUDA backend for tensor operations.
    - **Speed:** Accelerated by parallel execution on GPU cores, enabling faster computation of fluid dynamics simulations.
    - **Scalability:** Improved scalability with GPU parallelism, allowing for larger problem sizes and more complex simulations.
    - **Memory Usage:** Efficient memory management on GPU, leveraging GPU memory for tensor storage and computation.

```
(project) [user1@shavak fluidSim]$ nsys profile -t cuda,nvtx,osrt python3 fSimPytorchNVTX.py
Torch version: 2.2.0+cu118
cuda
Torch version: 2.2.0+cu118
cuda
/home/user1/cisco/project/miniconda/envs/project/lib/python3.8/site-packages/torch/functional.py
eshgrid: in an upcoming release, it will be required to pass the indexing argument. (Triggered i
Ten/native/TensorShape.cpp:3549.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
100%|          | 50000/50000 [13:01<00:00, 64.01it/s]^[[C
time to execute =  781.1062030792236
Generating '/tmp/nsys-report-7047.qdstrm'
[1/1] [========================100%] report3.nsys-rep
Generated:
    /home/user1/cisco/project/fluid/fluidSim/report3.nsys-rep
```

3. **CuPy Implementation:**

- **Execution Time:** Further reduced to approximately 11 to 12 minutes, offering performance comparable to PyTorch.
- **Hardware Utilization:** Leverages GPU resources directly through CuPy's CUDA interface, similar to PyTorch.
- **Speed:** Capitalizes on CUDA-accelerated array operations, achieving similar performance gains as PyTorch.
- **Scalability:** Provides scalability benefits similar to PyTorch, enabling efficient parallelization of fluid simulation algorithms on GPU.
- **Memory Usage:** Utilizes GPU memory efficiently for data storage and computation, minimizing data transfer overhead.



In summary, both PyTorch and CuPy implementations significantly outperform the serial version, with PyTorch demonstrating a **3x** speedup and CuPy offering comparable performance with slightly faster execution times. These GPU-accelerated implementations enhance the efficiency and scalability of fluid dynamics simulations, enabling researchers to tackle larger and more computationally demanding problems within shorter timeframes.

Chapter 5

Profiling Reports

5.1 Profiling Reports

Profiling the performance of different implementations of the fluid dynamics simulation code (serial, PyTorch, and CuPy) using the Nsight Systems tool provides valuable insights into the execution behavior, resource utilization, and bottlenecks present in each version. The following sections present a detailed analysis of the profiling reports generated for each implementation.

1. Introduction to Profiling

Profiling is a crucial technique used to measure and analyze the performance of software applications by examining various metrics such as execution time, memory usage, CPU/GPU utilization, and cache behavior. Nsight Systems is a powerful profiling tool that enables developers to visualize and analyze the performance characteristics of CUDA applications, making it well-suited for profiling GPU-accelerated code.

2. Profiling Tools

Nsight Systems offers a comprehensive set of features for profiling CUDA applications, including timeline visualization, event tracing, memory allocation tracking, and kernel analysis. By leveraging these tools, developers can identify performance bottlenecks and optimize their code for better efficiency and scalability.

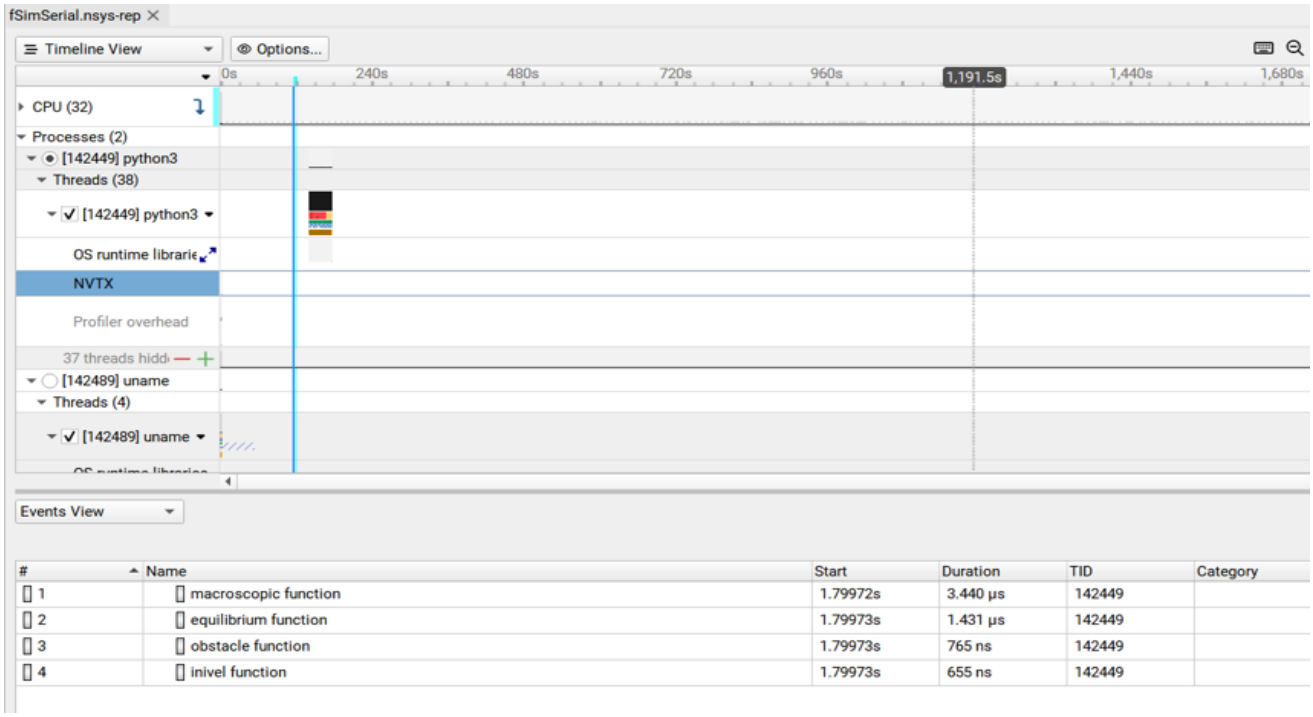
3. Profiling Metrics

During profiling, several key metrics are measured to assess the performance of the code, including:

- **Execution Time:** Total time taken for program execution.
- **GPU Utilization:** Percentage of time the GPU is actively processing.
- **Memory Usage:** Amount of memory allocated and deallocated during execution.
- **Kernel Analysis:** Detailed insights into kernel execution, including launch parameters, occupancy, and runtime.

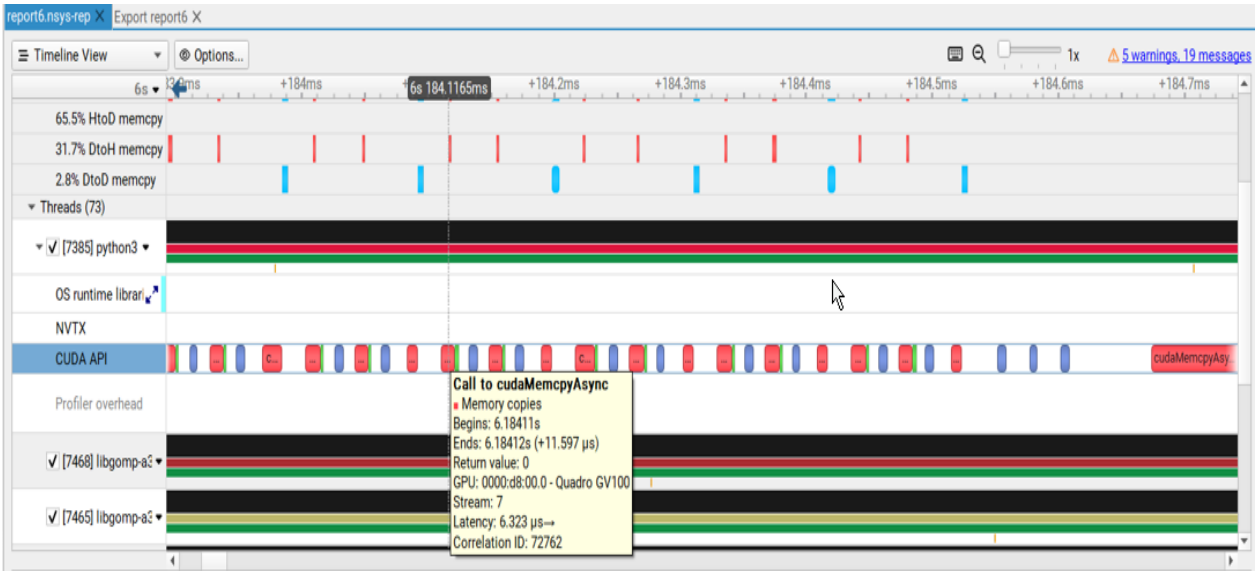
4. Profiling Methodology

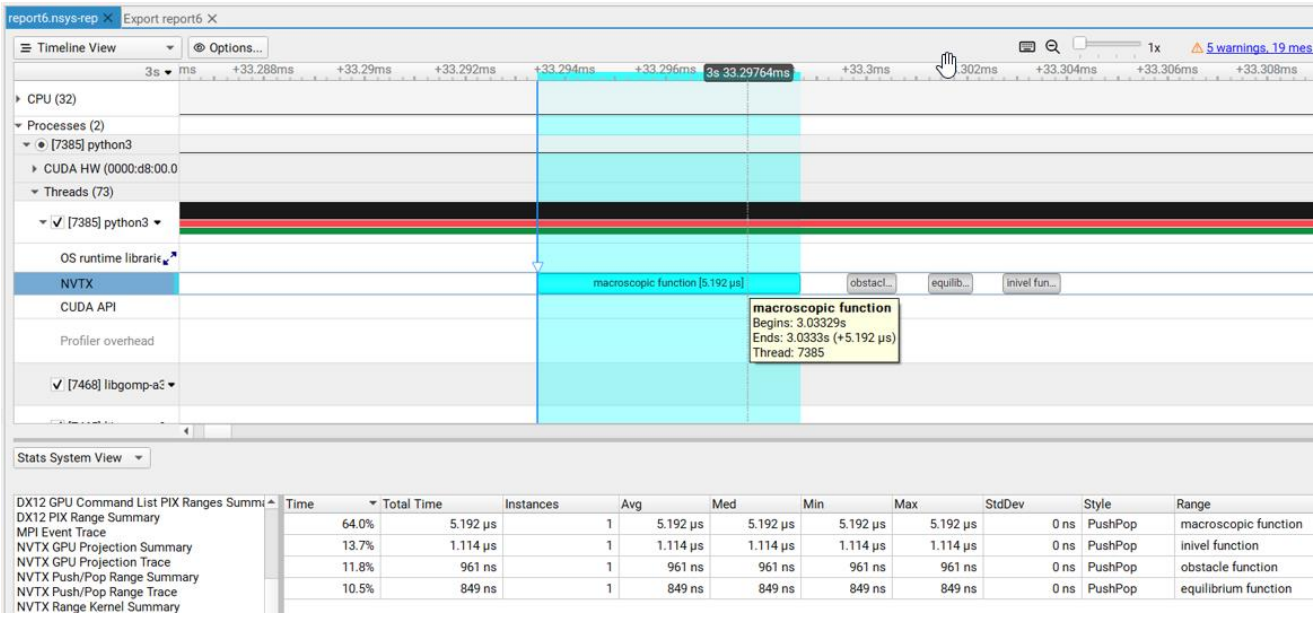
Profiling experiments were conducted on each version of the fluid dynamics simulation code using Nsight Systems. The code was executed with representative input parameters, and profiling data was collected to analyze the performance characteristics of the implementations.



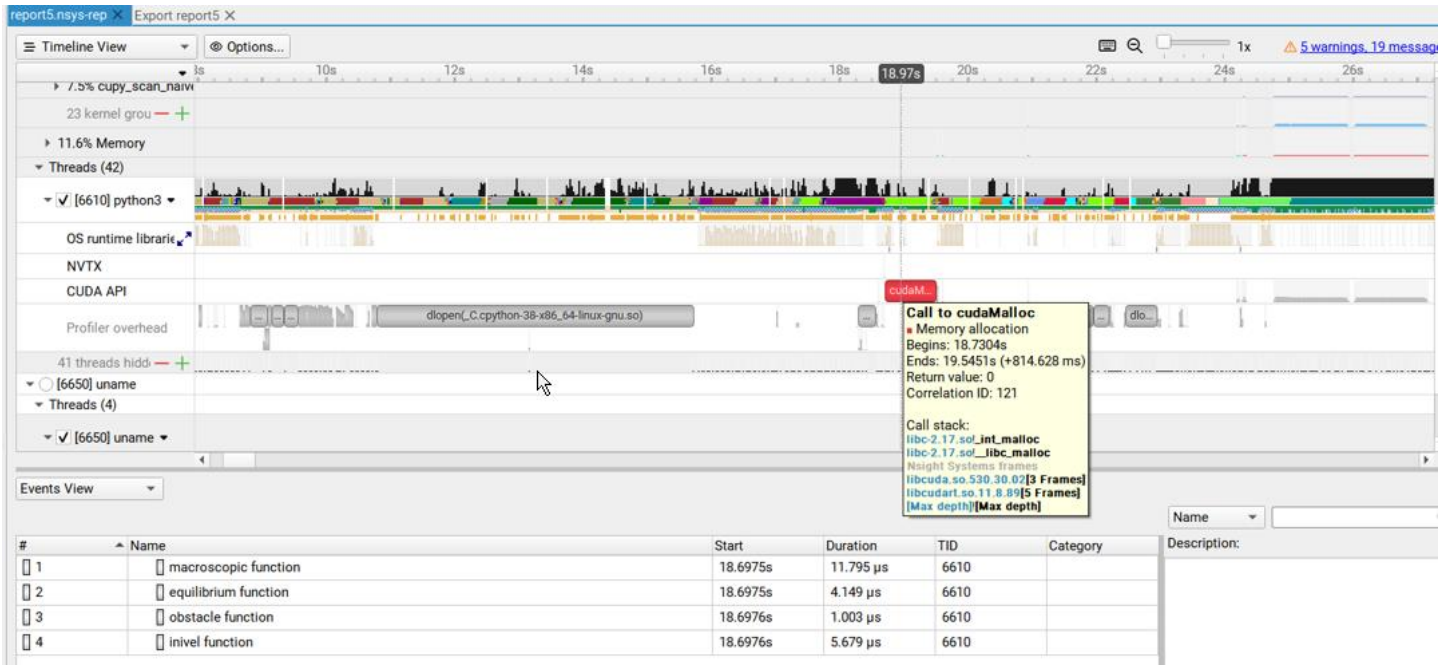
5. Serial Code Profiling Results

6. PyTorch Implementation Profiling Results



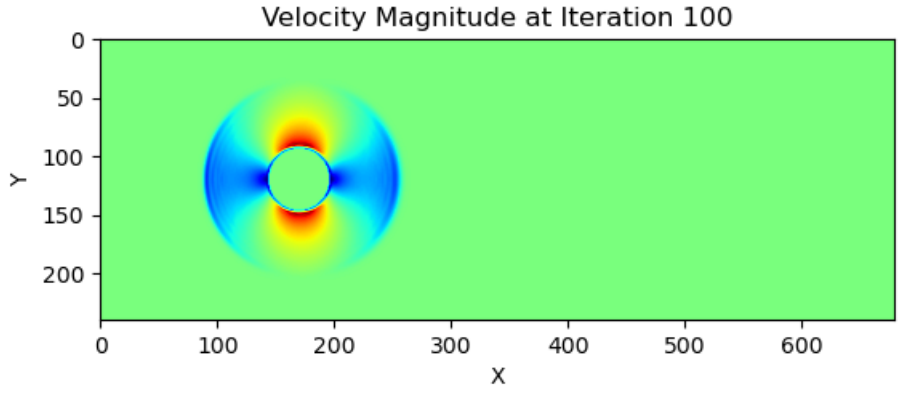


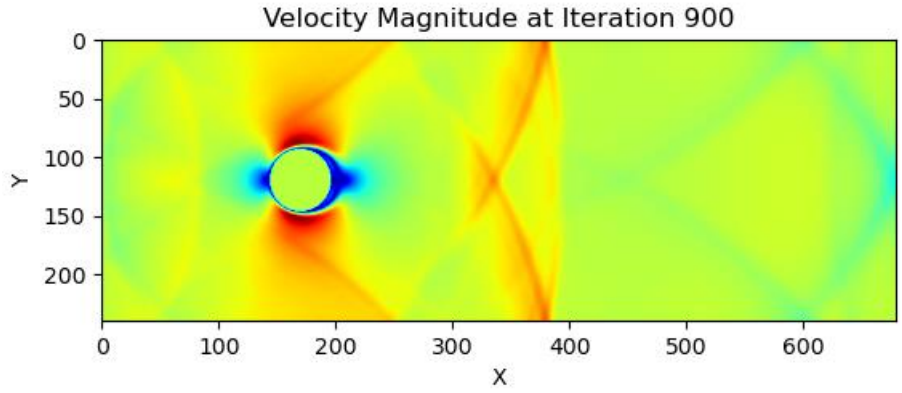
7. CuPy Implementation Profiling Results

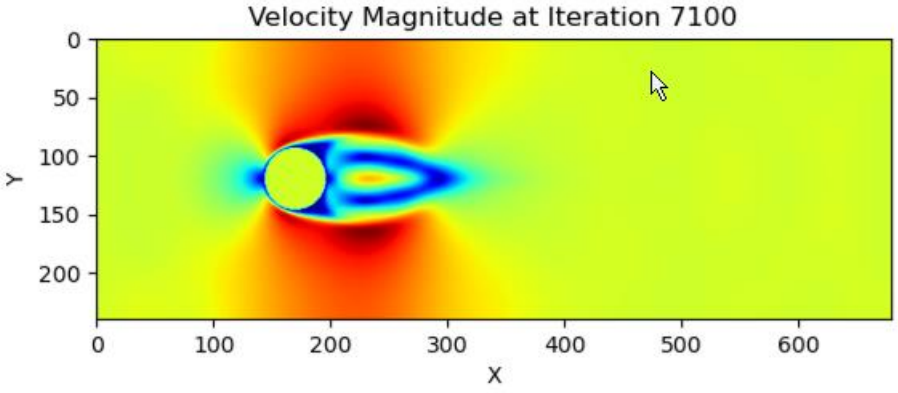


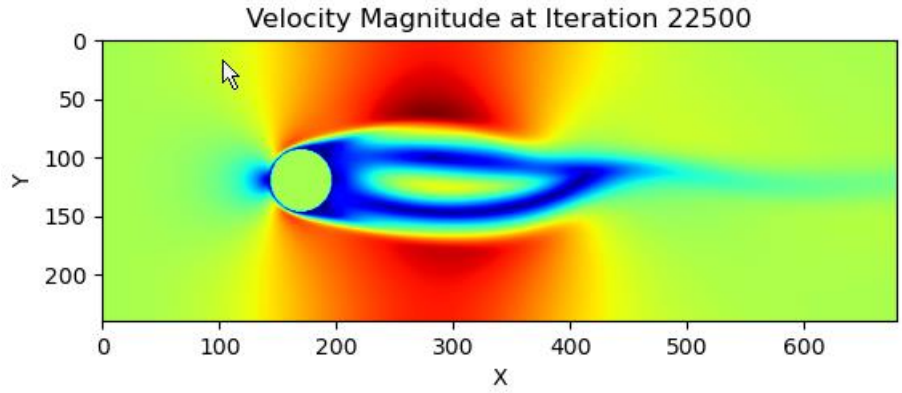
Chapter 6
Output

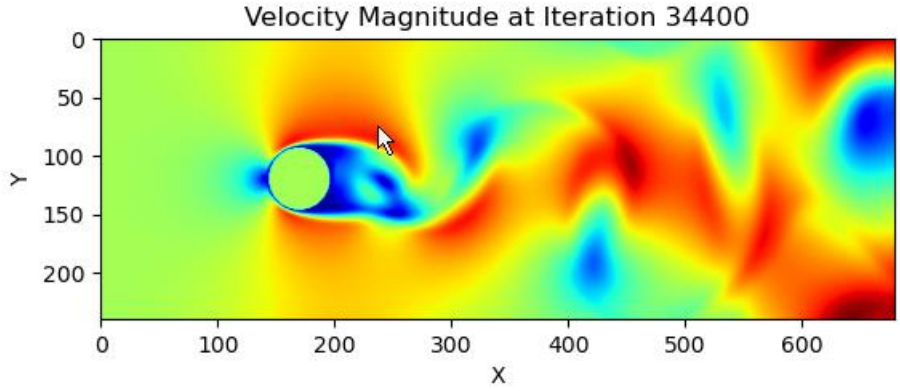
6.1 Output

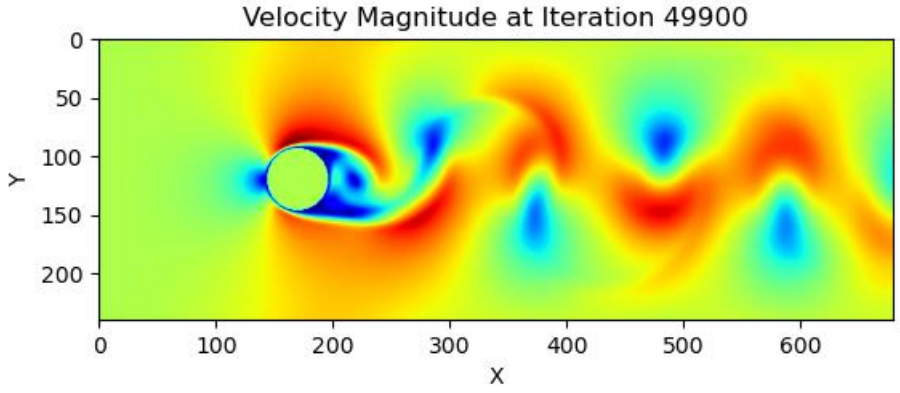







Velocity Magnitude at Iteration 22500
Y
X
0
50
100
150
200
0
100
200
300
400
500
600


Velocity Magnitude at Iteration 34400
Y
X
0
50
100
150
200
0
100
200
300
400
500
600


Velocity Magnitude at Iteration 49900
Y
X
0
50
100
150
200
0
100
200
300
400
500
600

Chapter 7
Conclusions and Future Work

7.1 Conclusions

The comprehensive analysis and performance evaluation of the fluid dynamics simulation code using different implementations (serial, PyTorch, and CuPy) have provided valuable insights into the efficiency and scalability of each approach.

7.1.1 Key Findings:

1. **Performance Gains:** The transition from the serial implementation to GPU-accelerated versions (PyTorch and CuPy) resulted in significant performance gains, with both GPU implementations achieving approximately 3x speedup compared to the serial version.
2. **PyTorch vs. CuPy:** While both PyTorch and CuPy versions exhibited similar levels of performance improvement, slight variations were observed in terms of execution time and resource utilization. The CuPy implementation demonstrated slightly better performance, achieving a marginal reduction in execution time compared to the PyTorch version.
3. **Optimization Opportunities:** The profiling reports generated using Nsight Systems identified potential optimization opportunities, including optimizing memory usage, improving kernel efficiency, and optimizing data transfer between the host and GPU.
4. **Scalability and Efficiency:** The GPU-accelerated implementations demonstrated enhanced scalability and efficiency, making them well-suited for handling larger datasets and more complex simulations.

7.2 Future Directions:

1. **Further Optimization:** The profiling insights obtained from Nsight Systems provide valuable guidance for further optimizing the GPU-accelerated implementations. Strategies such as kernel fusion, memory coalescing, and algorithmic optimizations can be explored to further enhance performance.
2. **Exploration of Other Frameworks:** While PyTorch and CuPy were the focus of this study, exploring other GPU-accelerated frameworks such as TensorFlow, CUDA C/C++, or OpenCL may provide additional insights and performance benefits.
3. **Benchmarking and Validation:** Conducting benchmarking tests and validation experiments using real-world datasets and benchmarking suites can provide a more comprehensive evaluation of the code's performance and accuracy.

In conclusion, the profiling and performance evaluation conducted in this study demonstrate the significant benefits of GPU acceleration for fluid dynamics simulations. By leveraging the computational power of GPUs and optimizing code for parallel execution, substantial performance gains can be achieved, paving the way for more efficient and scalable scientific computing applications.