GPU Computing Aiming at Vortex Filament Evolution

Yu-Hsun Lee

Graduate School of Informatics, Kyoto University

Abstract

In this thesis, we investigate a high-performance parallel computation for the vortex filament method based on Rosenhead's regularized Biot–Savart law for incompressible fluid on Graphics Processing Units (GPUs). Since it is hard to treat mathematical properties of the Biot–Savart law, which is a non-linear integro-differential equation, we focus on the numerical treatment of the Biot–Savart law and provide computation reliability for vortex filament evolution.

Our first achievement is efficient implementation of vortex filament evolution with GPU architecture. The computational complexity grows quadratically as an increase of discretization point in the discretized Biot–Savart law we use in this research. We accelerate computations by GPU architecture and improve the computational speed by optimizing the choice of warp size that affects the manipulation of GPU computing resources. We also use shared memory to get the best computing performance. We measure the rounding error quantitatively by comparing the numerical solutions with quadruple precision and multiple-precision arithmetic.

The second achievement is the case studies under bounded and unbounded vortex filaments proposed by Kimura and Moffatt (2018). We reproduce the numerical reconnection of vortex filaments by GPU computation and establish reliability based on numerical experiments. Our proposed GPU computation enables us to investigate the relation of temporal and spatial discretization parameters to mitigate disturbance in numerical solutions in reasonable computational time. We also propose a method to estimate the reconnection time. Moreover, the numerical solutions by the proposed discretization scheme converge exponentially to the solutions with the most significant spatial discretization parameter we use in the bounded case.

Acknowledgment

The author would like to thank the author's advisor, Professor Hiroshi Fujiwara, for his guidance and encouragement during the doctoral course. Without his support, the author could not tide over author's Ph.D. study in Japan.

The author thanks Professor Yuusuke Iso, Professor Emeritus Takaaki Nishida and Professor Daisuke Kawagoe for their advice and criticism.

The author also thanks Professor Yoshifumi Kimura at Graduate School of Mathematics, Nagoya University whose advice and works on fluid dynamics made enormous contribution to the author's Ph.D. study.

The author also thanks the members of sub-department of Applied Analysis for many help and suggestion.

Finally, the author would like to express his gratitude to his family for their supports in every way.

Contents

Chapter 1

Introduction

In this thesis, we discuss a design and implementation of Graphics Processing Units (GPU) computation for the sake of reliability of numerical treatment of three dimensional vortex filament evolution by the Biot-Savart law for incompressible fluid. The proposed acceleration enables us to process computation of the vortex filament evolution with various computational conditions, and thus we establish reliability based on numerical experiments.

We study the self-induced motion of a figure-of-eight vortex filament and a tilted hyperbola with two branches of vortex filaments with Rosenhead regularized Biot–Savart law [1, 2]. Let $\xi \in I$ be a parameter defined on an interval $I \subset \mathbb{R}$, and define the space curve $\boldsymbol{r}_\mu(t, \xi)$ where t is the time parameter. Then the velocity is represented by

$$\boldsymbol{v}_\mu(t, \boldsymbol{x}) = -\frac{\Gamma}{4\pi} \int_I \frac{\big(\boldsymbol{x} - \boldsymbol{r}_\mu(t, \xi)\big) \times \frac{\partial \boldsymbol{r}_\mu}{\partial \xi}}{\big(|\boldsymbol{x} - \boldsymbol{r}_\mu(t, \xi)|^2 + \mu^2\big)^{3/2}} \, \mathrm{d}\xi, \quad \boldsymbol{x} \in \mathbb{R}^3 \tag{1.1}$$

where μ is a regularization parameter, and the evolution of the space curve is evaluated by velocity

$$\frac{\partial \boldsymbol{r}_\mu}{\partial t} = \boldsymbol{v}_\mu\left(t, \boldsymbol{r}_\mu(t, \xi)\right). \tag{1.2}$$

The initial shapes of vortex filaments we shall treat are proposed by Y. Kimura and H. K. Moffatt [3, 4], and they reproduce the vortex reconnection with the cut-off regularized Biot-Savart law and have the consistency of Leray scaling for self-similarity in Navier–Stokes equations.

Since the regularized Biot-Savart law is a non-linear integro-differential equation, it is challenging to consider the properties and behavior of the exact solution by mathematical analysis. Local existence and uniqueness of solutions have been only proved for a specific function space [2], while estimation for finite-time singularity and behaviors as regularization have not been established. Therefore numerical computations play essential roles in investigating motion of vortex filament in three dimensions. However, there have been no detailed discussions on the reliability of numerical solutions as far as the author knows.

From the viewpoint of numerical computation, a discretized Biot–Savart law acts as an N-body problem, which has a quadratic computation complexity and takes huge computational time among the interaction of massive N bodies. As the first release of NVIDIA CUDA [5], a parallel computation programming interface for NVIDIA's GPU, Lars Nyland et al. implemented fast N-body simulation with CUDA in 2007 [6]. They focus on all-pairs N-body simulation rather than hierarchical approach like fast multipole method [7] that provide an idea to design and implement the accurate numerical computation of vortex filament evolution on GPU.

We study the acceleration of numerical methods for vortex filament evolution mainly on GPU and establish reliability examinations of numerical solutions. Our main contribution is investigating the relation between temporal and spatial discretization parameters to reproduce the reconnection of vortex filament by the high-performance GPU computing and investigate the reliability of GPU computation by estimating the accumulation of rounding errors. Moreover, we achieve 8.58 times faster computation on a personal computer by NVIDIA TITAN V GPU than the supercomputer used in the research of Y. Kimura and H. K. Moffatt [3].

This thesis contains four chapters. In Chapter 2, we discuss the detail of GPU implementation and algorithms of the vortex filament evolution. In Chapter 3, we study the evolution of vortex with bounded vortex filament and discuss the reliability of numerical experiments. In Chapter 4, we follow the similar discussions flow as Chapter 3 with unbounded vortex filaments. Later in this chapter, we will introduce the background of the Biot–Savart law and the vortex filament method.

Biot-Savart Law for Incompressible Fluid

Under the assumption of incompressible fluid, velocity $\boldsymbol{v}$ and the vorticity $\boldsymbol{\omega}$ satisfy relations

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{v}, \quad \nabla \cdot \boldsymbol{v} = 0. \tag{1.3}$$

By solving the Poisson equation on $\boldsymbol{v}$ in $\mathbb{R}^3$ under suitable assumptions, the velocity field $\boldsymbol{v}$ is represented by

$$\boldsymbol{v}(\boldsymbol{x}) = -\frac{1}{4\pi} \int_{\mathbb{R}^3} \frac{(\boldsymbol{x} - \boldsymbol{x'}) \times \boldsymbol{\omega}(\boldsymbol{x'})}{|\boldsymbol{x} - \boldsymbol{x'}|^3} \, \mathrm{d}\boldsymbol{x'}, \quad \boldsymbol{x} \in \mathbb{R}^3 \tag{1.4}$$

which is known as the Biot-Savart law for incompressible fluid [8]. Once we specified the vorticity, we could "recover" the velocity field by the Biot-Savart formula.

In order to investigate motion of the vortex, Helmholtz has proposed the concepts of vortex lines and vortex filaments (the fluid bounded by the vortex lines passing through the points of an infinitely small closed curve) [9, 10]. Helmholtz proved that vortex lines moved as material lines and described the properties of vortex filaments. In 1895, H. Lamb published the book "Hydrodynamics." He interpreted the vortex as a thin filament and connected the result with the Biot-Savart law in electromagnetism [11],

$$\boldsymbol{v}(\boldsymbol{x}) = -\frac{\Gamma}{4\pi} \int_C \frac{\big(\boldsymbol{x} - \boldsymbol{r}(s)\big) \times \frac{\partial \boldsymbol{r}}{\partial s}}{|\boldsymbol{x} - \boldsymbol{r}(s)|^3} \, \mathrm{d}l, \quad \boldsymbol{x} \in \mathbb{R}^3 \backslash C \tag{1.5}$$

where the right hand side is the line integral.

This thin vortex filament approximation describes the vicinity behavior of vortex filaments. As the point approach the vortex filament, the self-induced velocity results

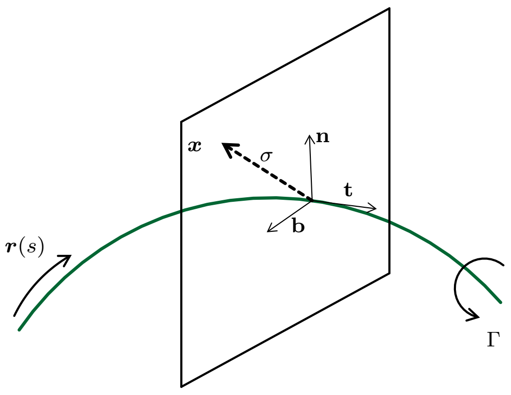


Figure 1.1: Illuatration of a vortex filament under the Frenet framework

in logarithmic divergence. The analysis of this divergence is firstly proposed by Da Rios [12], which is called the localized induction approximation (LIA).

$$\boldsymbol{v}(\boldsymbol{x}) = \frac{\Gamma}{4\pi}\kappa\mathbf{b}\ln\left(\frac{L}{\sigma}\right) + O(1), \quad \boldsymbol{x}\in\mathbb{R}^3\backslash C \tag{1.6}$$

where $\mathbf{b}$ is the binormal unit vector in the Fernet framework, L is the length of a neighboring segment and σ is the distance from the thickness of vortex tube. See Figure 1.1. Under the vortex filament approximation, this approximation did not consider the structures of the vortex core. The first attempt to take the finite core into account was done by Rosenhead [1].

For the development of the vortex filament method, Leonard [13] discussed the generalized Biot-Savart integral in three-dimension, which extends from the two-dimensional vortex blob method. The vortex filament method considers sufficiently smooth vortex structure and the accuracy of vortex stretching. The convergence of the vortex filament method is discussed by Greengard [14] and Soler [15].

Vortex Filament Method

In this section, we introduce the vortex filament method reviewed by Leonard [13]. In fluid dynamics, a vortex means the flow revolves around an axis line in a region which vorticity is distributed over a finite core with radius σ. Considering a thin vortex tube with radius $\sigma \ll 1$, we call the vorticity tube as a vortex filament which can be treated as a line vortex. To idealize the vortex filament as a smooth space curve, suppose the curve has no cross-sectional area. By Kelvin's circulation theorem, the circulation of a filament is a constant Γ in time for inviscid fluid [16]. For the configuration of space curves at time t is $\boldsymbol{r}(t,\xi)$, where $\xi\in I$ is a parameter on an interval $I\subset\mathbb{R}$. Thus the vorticity field is represented as

$$\boldsymbol{\omega}(t,\boldsymbol{x}) = \Gamma\int_I \gamma\left(\boldsymbol{x}-\boldsymbol{r}(t,\xi)\right)\frac{\partial\boldsymbol{r}}{\partial\xi}\,\mathrm{d}\xi, \quad \boldsymbol{x}\in\mathbb{R}^3 \tag{1.7}$$

where γ is a smooth function with rapid decay and normalization

$$\int_{\mathbb{R}^3} \gamma(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 1. \tag{1.8}$$

Assumed that γ has the form

$$\gamma(\boldsymbol{x}-\boldsymbol{r}) = \frac{1}{\sigma^3} p\left(\frac{|\boldsymbol{x}-\boldsymbol{r}|}{\sigma}\right). \tag{1.9}$$

Therefore a regularized Biot–Savart law of (1.4) by (1.7) is represented by

$$\boldsymbol{v}_g(t,\boldsymbol{x}) = -\frac{\Gamma}{4\pi}\int_I \frac{\big(\boldsymbol{x}-\boldsymbol{r}(t,\xi)\big)\times\frac{\partial \boldsymbol{r}}{\partial\xi}\, g(|\boldsymbol{x}-\boldsymbol{r}(t,\xi)|/\sigma)}{|\boldsymbol{x}-\boldsymbol{r}(t,\xi)|^3}\,\mathrm{d}\xi, \quad \boldsymbol{x}\in\mathbb{R}^3 \tag{1.10}$$

where g is defined by

$$g(y) = 4\pi\int_0^y p(t)t^2\,\mathrm{d}t \tag{1.11}$$

and from the normalization of γ, g has properties

$$\begin{aligned} g(y) &= O(y^3), && y\to 0\\ g(y) &\to 1, && y\to\infty. \end{aligned}$$

One of the well known regularization methods is proposed by Rosenhead [1], and the regularized Biot–Savart equation is represented by

$$\boldsymbol{v}_\mu(t,\boldsymbol{x}_\mu) = -\frac{\Gamma}{4\pi}\int_I \frac{\big(\boldsymbol{x}_\mu-\boldsymbol{r}_\mu(t,\xi)\big)\times\frac{\partial \boldsymbol{r}_\mu}{\partial\xi}}{\big(|\boldsymbol{x}_\mu-\boldsymbol{r}_\mu(t,\xi)|^2+\mu^2\big)^{3/2}}\,\mathrm{d}\xi, \quad \boldsymbol{x}_\mu\in\mathbb{R}^3 \tag{1.1}$$

where $\mu = O(\sigma)$. Under the Rosenhead regularization, smoothing function is considered as $g(y) = y^3/(y^2+\alpha)^{3/2}$, where α is a positive number. For the evolution of the curve $\boldsymbol{r}_\mu(t,\xi)$, it moves with the velocity,

$$\frac{\partial \boldsymbol{r}_\mu}{\partial t} = \boldsymbol{v}_\mu\left(t,\boldsymbol{r}_\mu(t,\xi)\right). \tag{1.2}$$

Moreover, if we consider there are M vortex filaments $\boldsymbol{r}^k$, $1\le k\le M$ in the space, then we could obtains the evolution of M vortex filaments by extending the (1.10) as:

$$\boldsymbol{v}_g(t,\boldsymbol{x}) = -\sum_{k=1}^{M}\frac{\Gamma_k}{4\pi}\int_{I_k} \frac{\Big(\boldsymbol{x}-\boldsymbol{r}_g^k(t,\xi)\Big)\times\frac{\partial \boldsymbol{r}_g^k}{\partial\xi}\, g(|\boldsymbol{x}-\boldsymbol{r}_g^k(t,\xi)|/\sigma_k)}{|\boldsymbol{x}-\boldsymbol{r}_g^k(t,\xi)|^3}\,\mathrm{d}\xi, \tag{1.12}$$

and the evolution of the curve $\boldsymbol{r}_g^k(t,\xi)$, it moves with the local velocity,

$$\frac{\partial \boldsymbol{r}_g^k}{\partial t} = \boldsymbol{v}_g\Big(t,\boldsymbol{r}_g^k(t,\xi)\Big). \tag{1.13}$$

Chapter 2

GPU Computation for the Biot–Savart law

In this study, we investigate the numerical computation of the Biot–Savart law with Rosenhead regularization which is presented in Eq. (1.1).

$$\boldsymbol{v}_\mu(t,\boldsymbol{x}) = -\frac{\Gamma}{4\pi}\int_I \frac{\big(\boldsymbol{x}-\boldsymbol{r}_\mu(t,\xi)\big)\times\frac{\partial \boldsymbol{r}_\mu}{\partial\xi}}{\big(|\boldsymbol{x}-\boldsymbol{r}_\mu(t,\xi)|^2+\mu^2\big)^{3/2}}\,\mathrm{d}\xi, \quad \boldsymbol{x}\in\mathbb{R}^3 \tag{1.1}$$

We consider the initial shape of vortex filament as a closed curve and has a Lagrangian parameterization as $C_0 = \{\boldsymbol{r}_\mu(0,\theta) \mid 0 \le \theta < 2\pi\}$. Let the time step $\Delta t > 0$, N be a positive integer, and $\theta_i = 2\pi i/N$ for $0 \le i < N$. We discretize the initial shape C_0 and allocate N nodes to $\boldsymbol{x}_{0,i} = \boldsymbol{r}_\mu(0,\theta_i)$. Then the discretized scheme of Eq. (1.1) for $\boldsymbol{x}_\mu \in \boldsymbol{r}_\mu(t,\xi)$ becomes

$$\boldsymbol{v}_{t,i} = -\frac{\Gamma}{4\pi}\frac{2\pi}{N}\sum_{j=0}^{N-1}\frac{(\boldsymbol{x}_{t,i}-\boldsymbol{x}_{t,j})\times\boldsymbol{\tau}_{t,j}}{\big(|\boldsymbol{x}_{t,i}-\boldsymbol{x}_{t,j}|^2+\mu^2\big)^{3/2}}, \quad 0\le i<N, \tag{2.1}$$

where $\boldsymbol{v}_{t,i}$ is velocity at $\boldsymbol{x}_{t,i} = \boldsymbol{r}_\mu(t,\theta_i)$. For the tangent vector $\boldsymbol{\tau}_{t,j}$, we use the Fourier method to ensure spectral accuracy. Define the Discrete Fourier Transform for each component x_j be

$$X_k = \sum_{j=0}^{N-1} x_j \mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}kj}, \tag{2.2}$$

then the corresponding component τ_j of tangent vector $\boldsymbol{\tau}_{t,j}$ can be approximated by the inverse Discrete Fourier Transform

$$\tau_j \approx \frac{1}{N}\left(\sum_{k=0}^{\frac{N}{2}-1}(\mathrm{i}\cdot k)X_k e^{\mathrm{i}\frac{2\pi}{N}jk} + \sum_{k=\frac{N}{2}}^{N-1}\big(\mathrm{i}\cdot(k-N)\big)X_k e^{\mathrm{i}\frac{2\pi}{N}j(k-N)}\right). \tag{2.3}$$

For the time-stepping, the 4th order Runge-Kutta method is used.

2.1 Introduction

In this section, we shall introduce the computation on Graphics Processing Units (GPUs). With the development of general-purpose computing on graphics processing units (GPGPU), we can perform high-performance scientific computing on GPUs. In 2007, NVIDIA released the Compute Unified Device Architecture (CUDA) as a parallel computing application programming interface (API) for NVIDIA's GPUs and developers can use C/C++ interface to the CUDA.

In this study, we use NVIDIA TITAN V, which was released at the end of 2017. NVIDIA TITAN V uses Volta micro-architecture GV100 and contains 80 streaming multiprocessors (SMs). For each GV100 SM, it contains 64 FP32 cores for single precision arithmetic processing and 32 FP64 cores for double precision arithmetic processing [17].

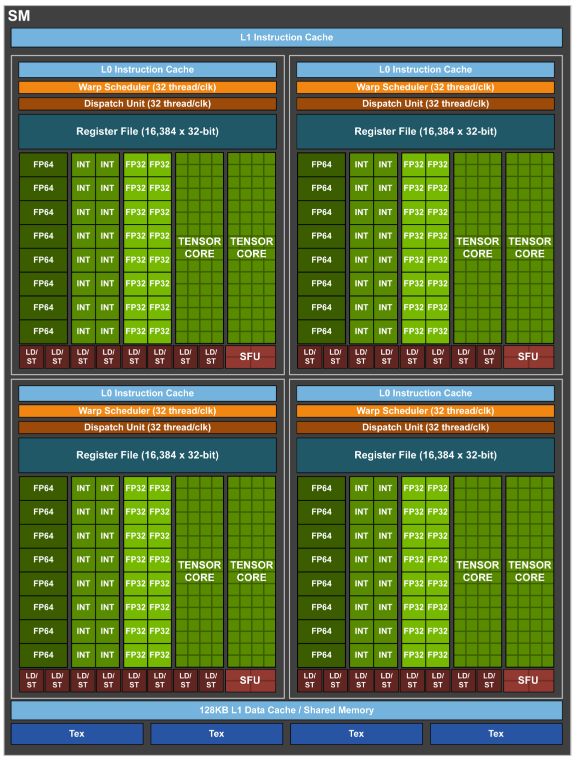

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Figure 2.1: Volta GV100 streaming multiprocessor [17]

In the CUDA GPU programming model, we use the terms: Grids, Blocks, Threads to describe how to manipulate the GPU resources. In general, we divide the computation on nodal points N into several GPU grids `G`; for each grid, we have GPU blocks `B` and serval threads `T` inside a block. Fig. 2.2 shows the grid, block, and thread hierarchy of the CUDA programming model [5]. Note that the maximum number of threads per multiprocessor is 2048, and the maximum number of threads per block is 1024 for TITAN V.

Table 2.1: Computational environment

CPU	Ryzen Threadripper 2990WX (3.0GHz, 32 cores)
Memory	DDR4-1866 128GB
OS	CentOS Linux 7.9.2009
GPU	TITAN V (5120 CUDA cores, 12GB HBM2 memory) with CUDA 10.2
C++ Compiler	GCC Version 4.8.5
FFT	fftw Version 3.3.8
exflib	Version 20180620

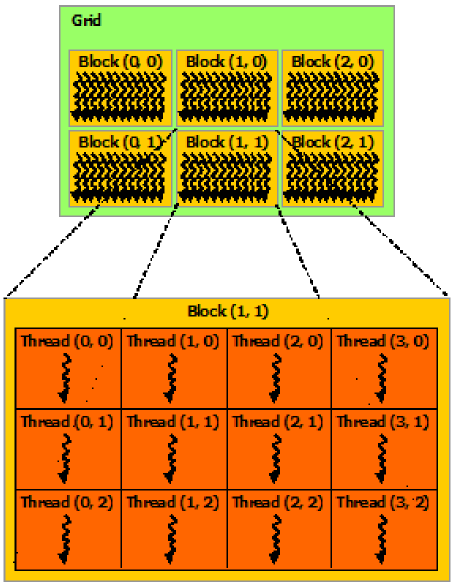


Figure 2.2: Grid of thread blocks [5]

2.2 Parallel Algorithm

In this section, we shall describe two parallelization methods to compute the Biot-Savart law. In the Eq. (2.1), we need to calculate the velocity by the Biot-Savart law at each nodal point. This step contains a computation complexity of $O(N^2)$.

Before we start a GPU computation, we need to generate the initial value on host memory then copy the values to GPU device memory. CUDA provides `malloc` like function `cudaMallocHost` and `cudaMalloc` to allocate the host and device memory. To copy the memory from host memory to GPU device memory, we use

```
cudaMemcpy(void *dst, void *src, size_t nbytes, cudaMemcpyHostToDevice)
```

where `dst` is destination pointer, `src` is source pointer, and `nbytes` is the size of memory we would like to copy. After the memory copy, all of the computation and memory accesses are done on the GPU device. Since the exchange of host memory and GPU memory is expensive, we need to reduce this kind of operation during computation.

After that, we introduce the algorithm to update the position of the nodal points by the Runge-Kutta method.

Algorithm 1: Runge-Kutta algorithm for time-stepping

```
Result: Update $\boldsymbol{x}_{t+\Delta t,i}$
Data: Input: $\boldsymbol{x}_{t,i}$ for $i = 0, 1, \cdots, N-1$
Step 1 :
    1. Calculate tangent vector from $\boldsymbol{x}_{t,i}$ by Eq. (2.3)
    2. Calculate $\boldsymbol{k}_i^1 = \boldsymbol{v}_{t,i}(\boldsymbol{x}_t)$ by Eq. (2.1)
    3. Update temporary position $\widetilde{\boldsymbol{x}}_i = \boldsymbol{x}_{t,i} + \Delta t \frac{\boldsymbol{k}_1}{2}$
Step 2 :
    1. Calculate tangent vector from $\widetilde{\boldsymbol{x}}_i$ by Eq. (2.3)
    2. Calculate $\boldsymbol{k}_i^2 = \boldsymbol{v}_{t,i}(\widetilde{\boldsymbol{x}})$ by Eq. (2.1)
    3. Update temporary position $\widetilde{\boldsymbol{x}}_i = \boldsymbol{x}_{t,i} + \Delta t \frac{\boldsymbol{k}_2}{2}$
Step 3 :
    1. Calculate tangent vector from $\widetilde{\boldsymbol{x}}_i$ by Eq. (2.3)
    2. Calculate $\boldsymbol{k}_i^3 = \boldsymbol{v}_{t,i}(\widetilde{\boldsymbol{x}})$ by Eq. (2.1)
    3. Update temporary position $\widetilde{\boldsymbol{x}}_i = \boldsymbol{x}_{t,i} + \Delta t \boldsymbol{k}_3$
Step 4 :
    1. Calculate tangent vector from $\widetilde{\boldsymbol{x}}_i$ by Eq. (2.3)
    2. Calculate $\boldsymbol{k}_i^4 = \boldsymbol{v}_{t,i}(\widetilde{\boldsymbol{x}})$ by Eq. (2.1)
Sum. Update $\boldsymbol{x}_{t+\Delta t,i} = \boldsymbol{x}_{t,i} + \frac{1}{6}\Delta t(\boldsymbol{k}_i^1 + 2\boldsymbol{k}_i^2 + 2\boldsymbol{k}_i^3 + \boldsymbol{k}_i^4)$
```

In each step of calculating the temporary velocity $\boldsymbol{k}$ by the Biot-Savart law from Eq. (2.1) contains a computation complexity of $O(N^2)$. For the closed curve initial condition, while we calculate the tangent vector, we also meet the complexity of $O(N^2)$ if we use the traditional DFT algorithm. `cuFFT` [18] helps us for doing the Fast Fourier Transform, which has a computational complexity of $O(N \log N)$ on CUDA enabled GPUs. We introduce two parallelization strategies to manipulate the $O(N^2)$ computation while updating the velocity. Throughout the chapter, we define the computation time for each step by updating one time step Δt, which includes four times of the Biot-Savart law calculation and 24 times (DFT and inverse DFT for each dimension per Runge-Kutta step) of the Fast Fourier Transform.

In the following subsections, we will introduce two parallelization strategies to manipulate GPU computing resources of calculating $\boldsymbol{k}_i$ in the Runge-Kutta algorithm. For the sample code present in subsections, we execute the GPU kernel code by

```
gpu_biot<<<B,T>>>
  ( double *const d_x, double *const d_y, double *const d_z,
    double *const d_tx,double *const d_ty,double *const d_tz,
    double *d_kx,double *d_ky,double *d_kz,
    const int LENGTH, const double d_dh, const double gamma
  )
```

where `B` and `T` represent the number of blocks and threads we manipulate for GPU computing respectively. Table 2.2 describes the meaning of each parameters.

Table 2.2: Parameters for computing the Biot-Savart law on GPU

parameter	meaning
`d_x`, `d_y`, `d_z`	components of $\boldsymbol{x}_i$, $0 \le i < N$
`d_tx`, `d_ty`, `d_tz`	components of tangent vector at $\boldsymbol{x}_i$, $0 \le i < N$
`d_kx`, `d_ky`, `d_kz`	empty array for storing temporary velocity $\boldsymbol{k}_i$, $0 \le i < N$
`LENGTH`	number of nodal points N
`d_dh`	$\Delta\theta = 2\pi/N$
`gamma`	constant strength Γ for vortex filament

2.2.1 One-dimensional Indexing

In the 1-D indexing, each thread computes the velocity of the i-th node contributed from N points. Suppose the size of a block, which means the number of threads inside the block, is $\texttt{T} = 2^n, n \in \mathbb{N}$. Under the limitation of CUDA, the maximum of `T` is 1024. The total number of blocks we need is $N/\texttt{T}$. For the mapping of threads and discrete indexing, the index i is calculated by `blockIndex` $\times$ `T + threadIndex`. With this thread allocation, the computation in each thread is $O(N)$ and all of threads compute in parallel. The sample code for each thread is shown in Listing 2.1 and a sketch of threads manipulating is illustrated in Fig. 2.3.

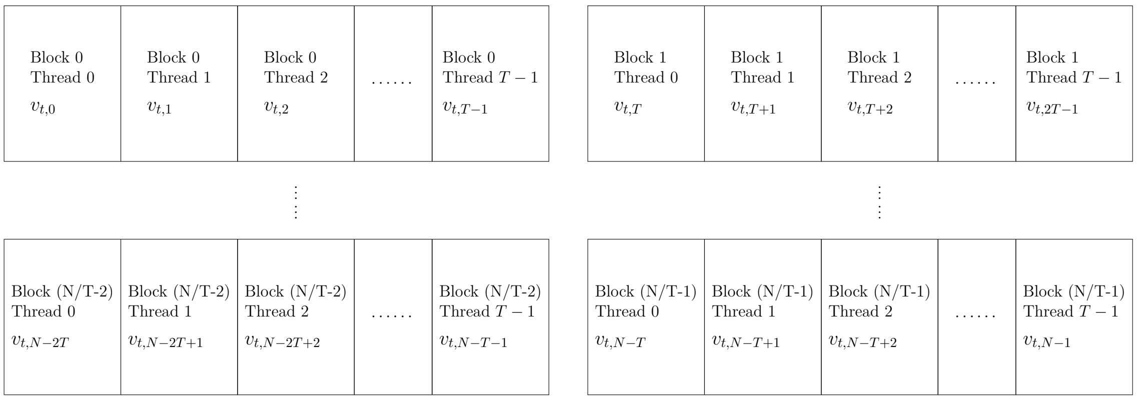

Block 0 Thread 0 $v_{t,0}$
Block 0 Thread 1 $v_{t,1}$
Block 0 Thread 2 $v_{t,2}$
......
Block 0 Thread $T-1$ $v_{t,T-1}$
Block 1 Thread 0 $v_{t,T}$
Block 1 Thread 1 $v_{t,T+1}$
Block 1 Thread 2 $v_{t,T+2}$
......
Block 1 Thread $T-1$ $v_{t,2T-1}$
Block (N/T-2) Thread 0 $v_{t,N-2T}$
Block (N/T-2) Thread 1 $v_{t,N-2T+1}$
Block (N/T-2) Thread 2 $v_{t,N-2T+2}$
......
Block (N/T-2) Thread $T-1$ $v_{t,N-T-1}$
Block (N/T-1) Thread 0 $v_{t,N-T}$
Block (N/T-1) Thread 1 $v_{t,N-T+1}$
Block (N/T-1) Thread 2 $v_{t,N-T+2}$
......
Block (N/T-1) Thread $T-1$ $v_{t,N-1}$

Figure 2.3: Sketch of threads manipulating for 1-D indexing

```
__global__ void gpu_biot
( double *const d_x, double *const d_y, double *const d_z,
  double *const d_tx,double *const d_ty,double *const d_tz,
  double *d_kx,double *d_ky,double *d_kz,
  const int LENGTH, const double d_dh, const double gamma)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  double dxj,dyj,dzj;
  double dtxj,dtyj,dtzj;
  double norm,ds_over_norm3;

  double mu = 1e-5;

  dxi = d_x[i]; dyi = d_y[i]; dzi = d_z[i];

  for(int j = 0; j < LENGTH; j++){
    dxj = d_x[j]; dtxj = d_tx[j];
    dyj = d_y[j]; dtyj = d_ty[j];
    dzj = d_z[j]; dtzj = d_tz[j];

    norm = sqrt((dxi-dxj)*(dxi-dxj)+(dyi-dyj)*(dyi-dyj)
                +(dzi-dzj)*(dzi-dzj)+mu*mu);
    ds_over_norm3 = (d_dh/(norm*norm*norm)*-1*gamma/(4*M_PI));

    d_kx[i] += (dtzj*(dyi-dyj)-dtyj*(dzi-dzj))*ds_over_norm3;
    d_ky[i] += (dtxj*(dzi-dzj)-dtzj*(dxi-dxj))*ds_over_norm3;
    d_kz[i] += (dtyj*(dxi-dxj)-dtxj*(dyi-dyj))*ds_over_norm3;
  }
}
```

Listing 2.1: Sample code for 1-D indexing

2.2.2 Two-dimensional Indexing

For a 2-D indexing method, we allocate the blocks and threads in 2D by `B` $= (N/T_x, N/T_y)$, and `T` $= (T_x, T_y)$. By the limitation of CUDA, $T_x \times T_y \leq 1024$. In this method, each thread only computes the velocity of the i-th point induced by the j-th point. The index i is calculated by `blockIndex` $\times$ T_x `+` `threadIndex` x and the index j is calculated by `blockIndex` $\times$ T_y `+` `threadIndex` y. In order to get the summation from each thread, we use `atomicAdd` to guarantee the result does not interfere with other threads. If we follow the way to sum of the result by following codes,

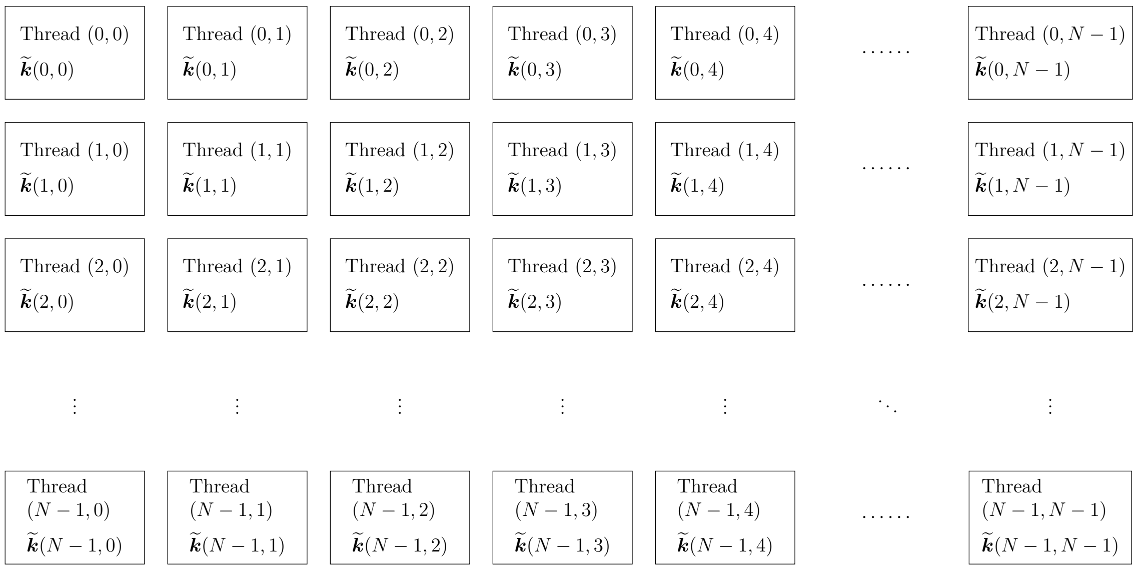
```
d_kx[i] += (dtzj*(dyi-dyj)-dtyj*(dzi-dzj))*ds_over_norm3;
d_ky[i] += (dtxj*(dzi-dzj)-dtzj*(dxi-dxj))*ds_over_norm3;
d_kz[i] += (dtyj*(dxi-dxj)-dtxj*(dyi-dyj))*ds_over_norm3;
```

this works well in 1-D indexing since the summation is serial. However in the 2-D indexing case, the summation to `d_kx`, `d_ky`, and `d_kz` is read and modify in parallel that causes a race condition. Several threads might read the same old values at the same time, then update the global memory with wrong values. `atomicAdd` use the compare-and-swap (CAS) operation which is an atomic instruction used in multithreading to realize lock-free data updates. Note that the atomic access to memory are all serialized but out-of-order.

Let the calculation of inside of summation in Eq. (2.1)

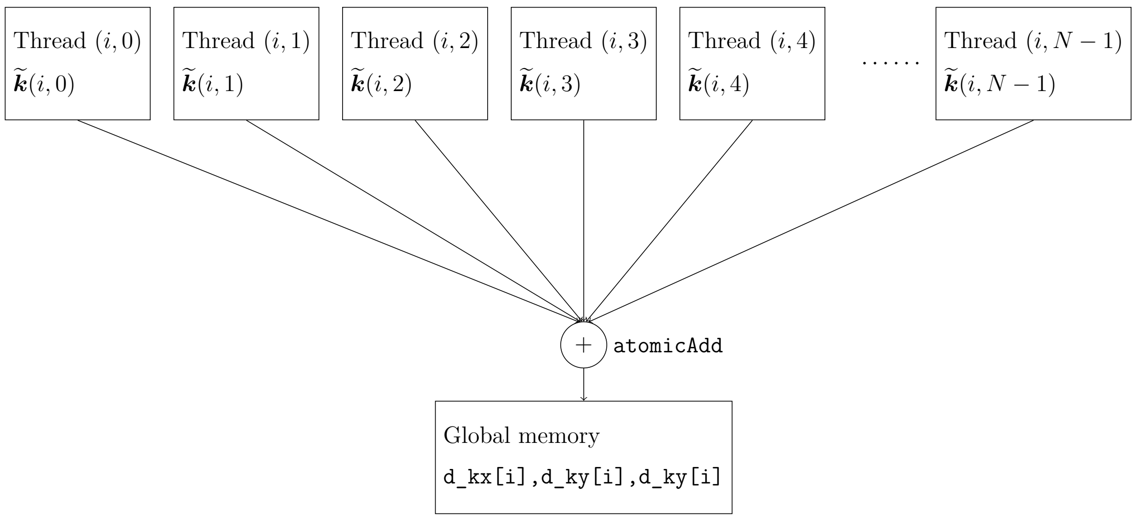
$$-\frac{\Gamma}{4\pi}\frac{2\pi}{N}\frac{(\boldsymbol{x}_{t,i}-\boldsymbol{x}_{t,j})\times\frac{\partial \boldsymbol{x}_{t,j}}{\partial\theta}}{\left(|\boldsymbol{x}_{t,i}-\boldsymbol{x}_{t,j}|^2+\mu^2\right)^{3/2}} \tag{2.4}$$

be $\widetilde{\boldsymbol{k}}(i,j)$. Fig. 2.4 represents the thread manipulating for 2-D indexing. For the sketch of threads computing and `atomicAdd` while calculating $\boldsymbol{k}_i$ is illustrated in Fig. 2.5.



Thread $(0,0)$ $\widetilde{\boldsymbol{k}}(0,0)$
Thread $(0,1)$ $\widetilde{\boldsymbol{k}}(0,1)$
Thread $(0,2)$ $\widetilde{\boldsymbol{k}}(0,2)$
Thread $(0,3)$ $\widetilde{\boldsymbol{k}}(0,3)$
Thread $(0,4)$ $\widetilde{\boldsymbol{k}}(0,4)$
......
Thread $(0,N-1)$ $\widetilde{\boldsymbol{k}}(0,N-1)$
Thread $(1,0)$ $\widetilde{\boldsymbol{k}}(1,0)$
Thread $(1,1)$ $\widetilde{\boldsymbol{k}}(1,1)$
Thread $(1,2)$ $\widetilde{\boldsymbol{k}}(1,2)$
Thread $(1,3)$ $\widetilde{\boldsymbol{k}}(1,3)$
Thread $(1,4)$ $\widetilde{\boldsymbol{k}}(1,4)$
......
Thread $(1,N-1)$ $\widetilde{\boldsymbol{k}}(1,N-1)$
Thread $(2,0)$ $\widetilde{\boldsymbol{k}}(2,0)$
Thread $(2,1)$ $\widetilde{\boldsymbol{k}}(2,1)$
Thread $(2,2)$ $\widetilde{\boldsymbol{k}}(2,2)$
Thread $(2,3)$ $\widetilde{\boldsymbol{k}}(2,3)$
Thread $(2,4)$ $\widetilde{\boldsymbol{k}}(2,4)$
......
Thread $(2,N-1)$ $\widetilde{\boldsymbol{k}}(2,N-1)$
Thread $(N-1,0)$ $\widetilde{\boldsymbol{k}}(N-1,0)$
Thread $(N-1,1)$ $\widetilde{\boldsymbol{k}}(N-1,1)$
Thread $(N-1,2)$ $\widetilde{\boldsymbol{k}}(N-1,2)$
Thread $(N-1,3)$ $\widetilde{\boldsymbol{k}}(N-1,3)$
Thread $(N-1,4)$ $\widetilde{\boldsymbol{k}}(N-1,4)$
......
Thread $(N-1,N-1)$ $\widetilde{\boldsymbol{k}}(N-1,N-1)$

Figure 2.4: Sketch of thread manipulating for 2-D indexing



Thread $(i,0)$ $\widetilde{\boldsymbol{k}}(i,0)$
Thread $(i,1)$ $\widetilde{\boldsymbol{k}}(i,1)$
Thread $(i,2)$ $\widetilde{\boldsymbol{k}}(i,2)$
Thread $(i,3)$ $\widetilde{\boldsymbol{k}}(i,3)$
Thread $(i,4)$ $\widetilde{\boldsymbol{k}}(i,4)$
......
Thread $(i,N-1)$ $\widetilde{\boldsymbol{k}}(i,N-1)$
+
atomicAdd
Global memory
d_kx[i],d_ky[i],d_ky[i]

Figure 2.5: Sketch of `atomicAdd` for 2-D indexing while calculating for index i

The sample code for the 2-D indexing and using `atomicAdd` is shown in Listing 2.2. Compare to the 1-D indexing, 2-D indexing is fully parallelized and faster than the 1-D indexing under small N. In Table 2.3, we show the time comparison for average execution time for each step under different N. We observe that in large N, the 1-D indexing has a shorter execution time.

```
__global__ void gpu_biot
( double *const d_x, double *const d_y, double *const d_z,
  double *const d_tx,double *const d_ty,double *const d_tz,
  double *d_kx,double *d_ky,double *d_kz,
  const int LENGTH, const double d_dh, const double gamma)
{
  double dxi,dyi,dzi;
  double dxj,dyj,dzj;
  double dtxj,dtyj,dtzj;
  double norm,ds_over_norm3;

  double mu = 1e-5;

  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  dxi = d_x[i]; dxj = d_x[j]; dtxj = d_tx[j];
  dyi = d_y[i]; dyj = d_y[j]; dtyj = d_ty[j];
  dzi = d_z[i]; dzj = d_z[j]; dtzj = d_tz[j];

  norm = sqrt((dxi-dxj)*(dxi-dxj)+(dyi-dyj)*(dyi-dyj)
              +(dzi-dzj)*(dzi-dzj)+mu*mu);
  ds_over_norm3 = (d_dh/(norm*norm*norm)*-1*gamma/(4*M_PI));

  atomicAdd(&d_kx[i], (dtzj*(dyi-dyj)-dtyj*(dzi-dzj))*ds_over_norm3);
  atomicAdd(&d_ky[i], (dtxj*(dzi-dzj)-dtzj*(dxi-dxj))*ds_over_norm3);
  atomicAdd(&d_kz[i], (dtyj*(dxi-dxj)-dtxj*(dyi-dyj))*ds_over_norm3);
}
```

Listing 2.2: Sample code for the 2-D indexing

Table 2.3: Comparison of average calculation time of two methods for calculate once of the Biot–Savart kernel code `gpu_biot`

unit: millisec.

N	1-D Indexing	2-D Indexing
8192	4.1	1.8
16384	8.1	8.0
32768	20.3	30.8
65536	80.9	133.1

2.3 Performance Optimization

2.3.1 Shared Memory

Inside CUDA programming, there are several kinds of memory. Each thread has private local memory. Each block has shared memory which is visible to threads that belong to the same block. All of the threads can access the global memory. Compare to global memory, shared memory provides a faster speed and lower latency to access data inside shared memory.

We represent Algorithm 2 to show how do we manipulate the shared memory for the Biot-Savart computation. Note that in order to use the shared memory in CUDA C++, we should declare the variable by using the `__shared__` variable declaration specifier. The CUDA built-in command `__syncthreads()` enables us to synchronize threads. The sample code and usage of shared memory are listed in Listing 2.3 lines 16-18. In this algorithm, we optimize the computation with 1-D indexing by the use of shared memory. However, the size of shared memory in NVIDIA TITAN V only has 48KB in each block. Acceleration is not apparent with a small size of shared memory. In Tables 2.4 and 2.5 we present two different size of block with $\texttt{T} = 32$ and $\texttt{T} = 1024$ which are consist to the size of shared memory. In the case $N = 65536$, larger shared memory shows more efficiency of computational speed. Note that the choice of `T` only affects the manipulation of threads for `gpu_biot`. For calculation of tangent vectors, the size of grids `T` is fixed. However, it is more improved if we choose a suitable thread size `T`. We will discuss the choice of thread size `T` and the warp in the next subsection.

Algorithm 2: Algorithm for manipulate the shared memory

```
Result: Update $\boldsymbol{v}_i$
Data: Input: $\boldsymbol{x}_i, \boldsymbol{\tau}_i = \partial \boldsymbol{x}_i / \partial \theta$ for $i = 0, 1, \cdots, N-1$
Step 1 : Set $i \leftarrow \texttt{blockIndex} * \texttt{T} + \texttt{threadIndex}$ be global index
Step 2 : Initialize shared memory $\widetilde{\boldsymbol{x}}$ with size $\texttt{T}$
Step 3 : Iterate by blockIndex $b$
    for $b \leftarrow 0$ to $N/T - 1$ do
        (a) Threads synchronization
        (b) Assign global data $\boldsymbol{x}_{b*\texttt{T}+\texttt{threadIndex}}$ to shared memory $\widetilde{\boldsymbol{x}}_{\texttt{threadIndex}}$.
        (c) Threads synchronization
        (d) Iterate by threadIndex $k$
            for $k \leftarrow 0$ to $T - 1$ do
                Update $\boldsymbol{v}_i$ by Eq. (2.1) where $\boldsymbol{x}_j$ is assigned in $\widetilde{\boldsymbol{x}}_k$
            end
    end
```

```
__global__ void gpu_biot
( double *const d_x, double *const d_y, double *const d_z,
  double *const d_tx,double *const d_ty,double *const d_tz,
  double *d_kx,double *d_ky,double *d_kz,
  const int LENGTH, const double d_dh, const double gamma)
```

```
{
  double dxi,dyi,dzi;
  double dtxj,dtyj,dtzj;
  double norm,ds_over_norm3;

  double mu = 1e-5;

  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int jj = threadIdx.x;

  __shared__ double d_xx[T];
  __shared__ double d_yy[T];
  __shared__ double d_zz[T];

  dxi = d_x[i];
  dyi = d_y[i];
  dzi = d_z[i];

  for(int j=0; j<LENGTH; j+= blockDim.x)
  {
    __syncthreads();
    d_xx[jj] = d_x[j+jj];
    d_yy[jj] = d_y[j+jj];
    d_zz[jj] = d_z[j+jj];
    __syncthreads();

    for(int k=0; k<blockDim.x;k++)
    {
      dtxj = d_tx[j+k];
      dtyj = d_ty[j+k];
      dtzj = d_tz[j+k];
      norm = std::sqrt((dxi-d_xx[k])*(dxi-d_xx[k]) +
          (dyi-d_yy[k])*(dyi-d_yy[k]) +
          (dzi-d_zz[k])*(dzi-d_zz[k])+mu*mu);
      ds_over_norm3=(d_dh/(norm*norm*norm)*-1*gamma/(4*M_PI));

      d_kx[i]+=(dtzj*(dyi-d_yy[k])-dtyj*(dzi-d_zz[k]))*ds_over_norm3;
      d_ky[i]+=(dtxj*(dzi-d_zz[k])-dtzj*(dxi-d_xx[k]))*ds_over_norm3;
      d_kz[i]+=(dtyj*(dxi-d_xx[k])-dtxj*(dyi-d_yy[k]))*ds_over_norm3;
    }
  }
}

```

Listing 2.3: Sample code for 1-D indexing with shared memory

2.3.2 Warp Optimization

Warps are the basic execution unit in each CUDA core. Inside a warp, it contains 32 consecutive threads, which are executed simultaneously with same instructions. No matter how we design the grid blocks in 1-D or 2-D, the hardware will partition the thread blocks into warps. In Tables 2.4 and 2.5, it is a small improvement of computation speed by using shared memory. However, it is essential to choose the

Table 2.4: Comparison of average calculation time of using shared memory for each time step under $T = 32$

unit: millisec.

	1-D Using Global Memory			1-D Using Shared Memory		
N	sum.	tangent	total	sum.	tangent	total
8192	16.5	0.40	17.1	16.2	0.33	16.7
16384	33.5	0.50	34.2	32.6	0.49	33.4
32768	80.8	0.51	81.7	81.4	0.48	82.3
65536	324.6	0.51	325.6	323.8	0.50	324.9

Table 2.5: Comparison of average calculation time of using shared memory for each time step under $T = 1024$

unit: millisec.

	1-D Using Global Memory			1-D Using Shared Memory		
N	sum.	tangent	total	sum.	tangent	total
8192	35.2	0.36	35.8	34.8	0.36	35.3
16384	70.2	0.48	70.9	69.5	0.47	70.2
32768	140.1	0.53	141.1	138.8	0.51	139.7
65536	366.0	0.53	367.1	359.5	0.52	360.7

size of threads. It takes twice of execution time if we do not optimize the size of threads. In implementation of Rosenhead's scheme, there are no branches during the calculation that will not cause the warp divergence. Tables 2.6 and 2.7 compare computation time depending on the choice of block sizes. In the 1-D indexing case, `T` $= 32$ performs the shortest execution time, where `T` $= 32$ coincides with the thread size of a warp. In 2-D indexing, `T` $= (16, 16)$ attains the shortest execution time.

Table 2.6: Comparison of average calculation time of block size for each step with 1-D indexing

unit: millisec.

$N \backslash T$	16	32	64	128	256	512	1024
8192	16.4	16.4	16.3	16.3	17.1	20.6	35.0
16384	37.9	32.7	34.4	34.1	34.4	40.9	69.6
32768	125.5	82.0	85.0	86.9	90.5	91.6	138.8
65536	529.3	325.1	346.0	341.5	372.6	358.9	361.7

2.4 Quantitative Estimate of Rounding Errors by Multiple-Precision Arithmetic

In GPUs, single precision arithmetic and half precision arithmetic are commonly used for graphics processing purposes. With the development of GPUs, many applications require the accuracy of floating-point computation. Nowadays, GPGPU

Table 2.7: Comparison of average calculation time of block size for each step with 2-D indexing

unit: millisec.

$N\backslash T$	(8,8)	(16,16)	(32,32)
8192	9.2	8.3	9.1
16384	36.2	34.7	36.6
32768	126.6	115.3	135.6
65536	539.5	533.3	612.5

is used for high-performance scientific computing, and double precision arithmetic is also supported by some NVIDIA GPUs.

NVIDIA GPUs comply with the IEEE754-2008 standard [19, 20, 21] of single and double precision binary floating-point format, which are widely used in scientific computation, for hardware-level support. In the IEEE754 standard, a floating-point number consists of three fields: sign, exponent, and fraction. IEEE754 double precision has 53-bits in the fraction part, which is approximately 16 decimal digits. However, in some advanced scientific and engineering computations, the influence of rounding errors is serious. Particularly, it is hard to measure rounding errors in parallel computing since the computational order may not be preserved.

One strategy to reduce rounding errors is to extend the precision of the fraction part in floating-point expressions. In IEEE754-2008 standard, it specifies quadruple precision arithmetic as a 128-bits binary floating-point format which contains 113-bits in the fraction part. This gives approximately 34 significant decimal digits precision. However, quadruple precision arithmetic has not yet been popular for native hardware support. For software libraries level implementation, one of the de facto standard compiler GNU Compilers Collection (GCC) provides a quadruple precision arithmetic environment *quadmath* [22] on AMD64 and Intel64 architectures, which are popular processor architectures for PC. Furthermore, we also use a multiple-precision arithmetic environment *exflib* [23], which enables us to specify arbitrarily computational precision in advance. Table 2.8 presents bits lengths and precisions of floating-point types used in the present thesis.

In order to verify the reliability of double precision computation on GPU and estimate rounding errors introduced by the approximation of the Rosenhead regularization method, we process computation with three different precisions: double precision, quadruple precision, and multiple precision (50 digits) . We assume the results with exflib (50 digits) have the most accuracy, and then we are able to estimate rounding errors in results with IEEE754 double precision and quadruple precision arithmetic by using those with exflib as a reference. For further numerical results, discussions are presented in Section 3.3 and Section 4.3.

Table 2.8: Precision of floating-point types used in experiments. In exflib, 50 decimal digits is specified as desired computational precision.

type	total	sign	exponent		fraction
	(bits)	(bits)	(bits)	(bits)	(decimal digits)
IEEE754 double	64	1	11	53	(15.95)
quadmath	128	1	15	113	(34.02)
exflib (50 digits)	256	1	63	193	(58.10)

2.5 Computational Time of Figure-of-eight Vortex Filament Evolution

We apply the proposed design and implementation to the figure-of-eight vortex filament evolution [3]. Table 2.9 shows elapse times for some pairs of N and Δt in the computational environment presented in Table 2.1. For the time-stepping, the Runge-Kutta method shown in Algorithm 1 is adopted. For instance, the proposed GPU computation consumes approximately 0.0839 seconds to obtain $\boldsymbol{x}_{t+\Delta t,i}$ from $\boldsymbol{x}_{t,i}$ for the case $N = 32768$.

In order to avoid the singularity, Y. Kimura and H. K. Moffatt have adopted the cut-off method [3] for the Biot-Savart law with respect to arc-length which is given by

$$\boldsymbol{v}_i = -\frac{\Gamma}{4\pi}\sum_{\substack{j=0\\ j\neq i}}^{N-1}\frac{(\boldsymbol{x}_i-\boldsymbol{x}_j)\times\boldsymbol{\tau}_j}{|\boldsymbol{x}_i-\boldsymbol{x}_j|^3}\Delta s_j, \quad 0\le i<N, \tag{2.5}$$

where

$$\Delta s_j = \begin{cases} (|x_0-x_{N-1}|+|x_1-x_0|)/2, & j=0,\\ (|x_j-x_{j-1}|+|x_{j+1}-x_j|)/2, & 1\le j\le N-2,\\ (|x_{N-1}-x_{N-2}|+|x_0-x_{N-1}|)/2, & i=N-1. \end{cases} \tag{2.6}$$

and $\boldsymbol{\tau}_j \equiv \partial\boldsymbol{x}_j/\partial s$ is the unit tangent vector at $\boldsymbol{x}_j$.

In computation presented by Kimura and Moffatt [4], they have employed the supercomputer Fujitsu PREIMEHPC FX100 in Information Technology Center, Nagoya University. Their computational time is approximately 403312 seconds as the wall clock time with $N = 32768$ and $\Delta t = 6.25\times10^{-7}$ for $0\le t\le 0.35$ ($= 560000\Delta t$). It means that their computation consumes approximately 0.720 seconds to compute $\boldsymbol{x}_{t+\Delta t,i}$ from $\boldsymbol{x}_{t,i}$.

From the results, the proposed GPU computation achieves 8.58 times faster computation than CPU shared memory parallelization.

Table 2.9: Timing comparison of different N with figure-of-eight vortex filament under GPU computation, $0 \leq t \leq 0.4$

N	Δt	Total
4096	5×10^{-5}	1m 8s
8192	2×10^{-5}	5m 36s
16384	4.5×10^{-6}	50m 12s
32768	1×10^{-6}	9h 19m 18s
65536	3×10^{-7}	5d 10h 33m 29s

Chapter 3

Bounded case : Figure-of-eight Vortex Filament

The purpose of Chapter 3 and Chapter 4 is to demonstrate an effectiveness of the proposed GPU computation to ensure reliability of numerical computation of the Biot-Savart law. In particular, we adopt the figure-of-eight vortex as the initial condition in this chapter.

The figure-of-eight vortex is proposed by Y. Kimura and H. K. Moffatt [3]. This shape is described as the skewed vortices by adding two ring parts at the ends of vortices. The following particular parametrization of the curve is used in this chapter. The initial shape of the figure-of-eight vortex is illustrated in Fig. 3.1a:

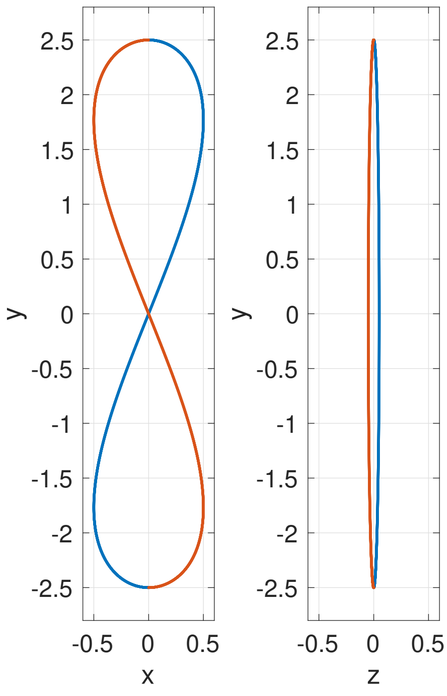
$$\boldsymbol{x} = \begin{cases} x(\theta) = 0.5 \sin 2\theta, \\ y(\theta) = 2.5 \sin \theta, \quad (0 \le \theta < 2\pi) \\ z(\theta) = 0.05 \cos \theta. \end{cases} \tag{3.1}$$

Let us recall the discretization of Biot–Savart equation with Rosenhead regularization for a closed curve,

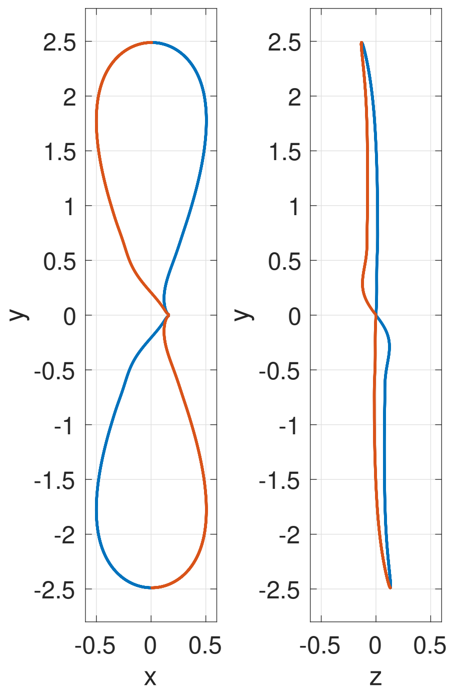
$$\boldsymbol{v}_{t,i} = -\frac{\Gamma}{4\pi}\frac{2\pi}{N}\sum_{j=0}^{N-1}\frac{(\boldsymbol{x}_{t,i}-\boldsymbol{x}_{t,j}) \times \frac{\partial \boldsymbol{x}_{t,j}}{\partial \theta}}{\left(|\boldsymbol{x}_{t,i}-\boldsymbol{x}_{t,j}|^2+\mu^2\right)^{3/2}}, \quad 0 \le i < N \tag{2.1}$$

In the discretization, we allocate the initial shape by $\boldsymbol{x}_{0,i} = \boldsymbol{x}(\theta_i)$ with $\theta_i = 2\pi i/N$, where $0 \le i < N$. Throughout this chapter, $\Gamma = 4\pi/50$, $\mu = 10^{-5}$ and the proposed GPU implementation with double precision arithmetic are used unless otherwise stated.

Our numerical solutions at $t = 0.338$ and $t = 0.340$ are shown in Fig. 3.2. The central parts of the curve which are separated at $t = 0$ as shown in Fig. 3.1a touch each other near $y = 0$ depicted in Fig. 3.1b and 3.2b where blue and orange parts represent those in $z > 0$ and $z < 0$ in the initial condition (3.1). This is called the reconnection of the vortex filament.

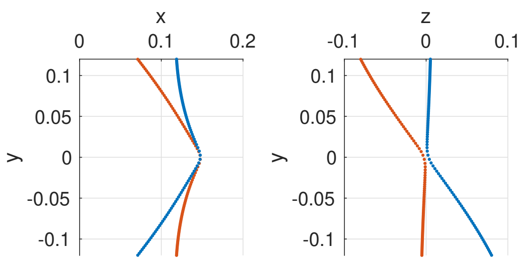


(a) Initial shape (3.1)

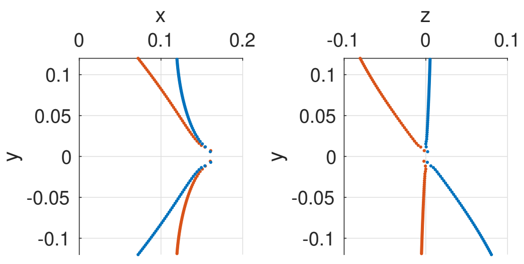


(b) Reconnection which is observed at $t = 0.340$ by numerical simulation with $N = 8192$ and $\Delta t = 2 \times 10^{-5}$

Figure 3.1: Evolution of figure-of-eight vortex filament



(a) $t = 0.338$



(b) $t = 0.340$

Figure 3.2: Reconnection of figure-of-eight vortex filament near the estimated reconnection time $t = 0.3385$ ($N = 8192$, $\Delta t = 2 \times 10^{-5}$)

3.1 Reliability Criteria

We investigate the relation between temporal and spatial discretization parameters. In the discussions on finite difference schemes for time evolutional problems, the stability of numerical schemes plays an essential role in ensuring the reliability of numerical solutions. For typical cases, the stability of schemes is involved with temporal and spatial discretization parameters. On the other hand, conditions for stability of the scheme (2.1) have not been developed as far as the author knows.

In numerical computation with $N = 8192$ and $\Delta t = 4 \times 10^{-5}$ by the Runge-Kutta method, the numerical results start oscillating at a very early stage $t = 0.0020$ ($= 50\Delta t$) as Fig. 3.3a and Fig. 3.3b which are respectively obtained under the double precision and multiple precision arithmetic computation, while numerical solutions with $N = 8192$ and $\Delta t = 2 \times 10^{-5}$ under double precision arithmetic computation have no disturbance as Fig. 3.3c.

In order to discuss the reason of the disturbance, we calculate

$$|a_{t,n}|^2 = |X_{t,n}|^2 + |Y_{t,n}|^2 + |Z_{t,n}|^2, \tag{3.2}$$

where $\{X_{t,n}\}$, $\{Y_{t,n}\}$, and $\{Z_{t,n}\}$ are discrete Fourier transform defined by

$$X_{t,n} = \sum_{k=0}^{N-1} x_{t,k} \mathrm{e}^{-i\frac{2\pi}{N}kn}, \tag{3.3}$$

$$Y_{t,n} = \sum_{k=0}^{N-1} y_{t,k} \mathrm{e}^{-i\frac{2\pi}{N}kn}, \tag{3.4}$$

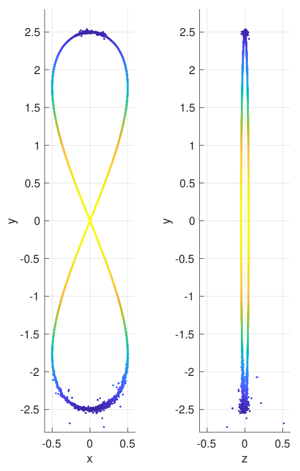
and

$$Z_{t,n} = \sum_{k=0}^{N-1} z_{t,k} \mathrm{e}^{-i\frac{2\pi}{N}kn}, \tag{3.5}$$

where $x_{t,k}$, $y_{t,k}$, and $z_{t,k}$ are components of $\boldsymbol{x}_{t,k}$. Fig. 3.4 depicts behaviors of $\{a_{t,n}\}$. For the initial shape, the amplitude decays exponentially with respect to the Fourier modes. As time progresses in the case $\Delta t = 4 \times 10^{-5}$, the effect from high frequency modes grow rapidly. Around $t = 0.0006$, the amplitude of high frequency modes reaches same level as low frequency modes. On the contrary, $|a_{t,n}|$ with $N = 8192$ and $\Delta t = 2 \times 10^{-5}$ decays exponentially in the period $0 \leq t \leq 0.004$ as Fig. 3.4c. This corresponds to the differences between Figs. 3.3a, b and Fig. 3.3c.

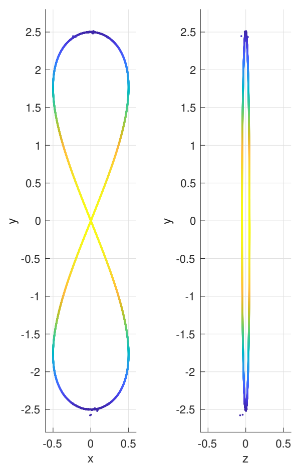
With the efficiency of GPU computation, we found the parameter pairs of N and Δt, which enable us to meet reconnection of the vortex filament numerically. For the cases which could reproduce the reconnection called *Type S* parameters. For the cases which diverge in a very early stage, we called it *Type U* parameters.

Fig. 3.5 shows our numerical results with N versus Δt on the log-log scale. The red circles(○) in the figure mean the *Type S* pairs of N and Δt, while the blue crosses($\times$) mean the *Type U* pairs we have examined. The green line is our estimated interface of the *Type S* and *Type U* region calculated by the least-square method. We state that the interface is proportional to $1/N^2$. We also note that the interface is influenced by the choice of regularization parameters. Comparison among $\mu = 10^{-2}, 10^{-3}, 10^{-4}$, and 10^{-5} is shown in Fig. 3.6.



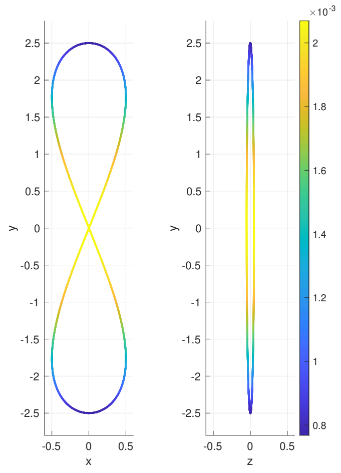
$\Delta t = 4 \times 10^{-5}$

(a) Type U, double precision



$\Delta t = 4 \times 10^{-5}$

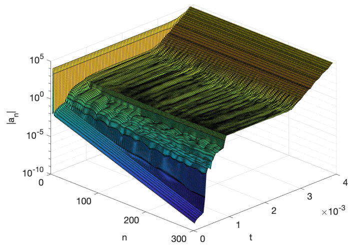
(b) Type U, multiple precision



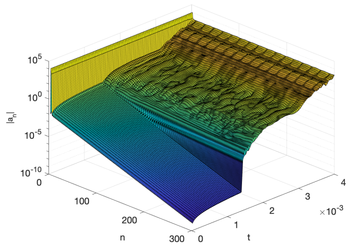
$\Delta t = 2 \times 10^{-5}$

(c) Type S, double precision

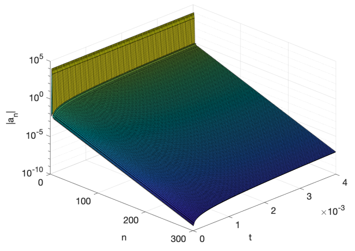
Figure 3.3: Numerical results by Type U and Type S parameter pairs ($t = 0.0020$, $N = 8192$). The color bar corresponds to the distance calculated by (2.6).



(a) $\Delta t = 4 \times 10^{-5}$, double precision



(b) $\Delta t = 4 \times 10^{-5}$, multiple precision



(c) $\Delta t = 2 \times 10^{-5}$, double precision

Figure 3.4: Amplitude of Fourier modes $|a_{t,n}|$ for $0 \leq t \leq 0.004$, $N = 8192$

Moreover, we plot the interface line for several Γ in Fig. 3.7 , which exhibits that Δt will proportional to $1/|\Gamma|$.

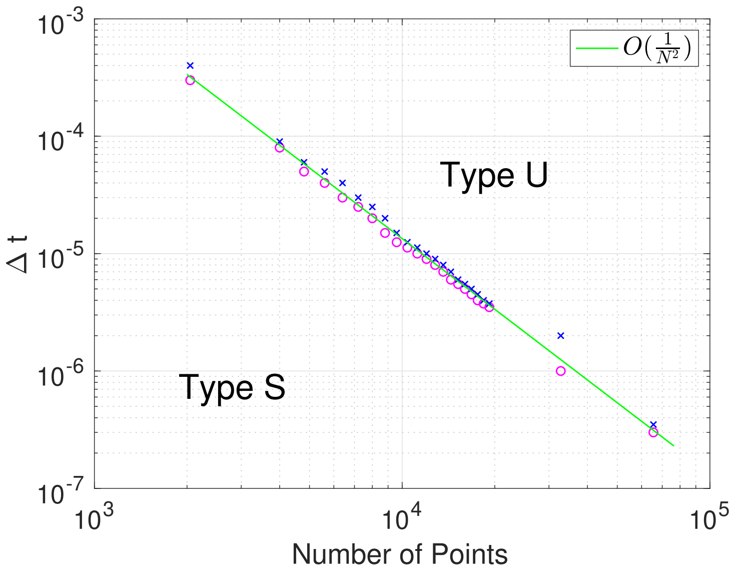


Figure 3.5: Reliable region for figure-of-eight vortex, $\Gamma = \frac{4\pi}{50}$ and $\mu = 10^{-5}$

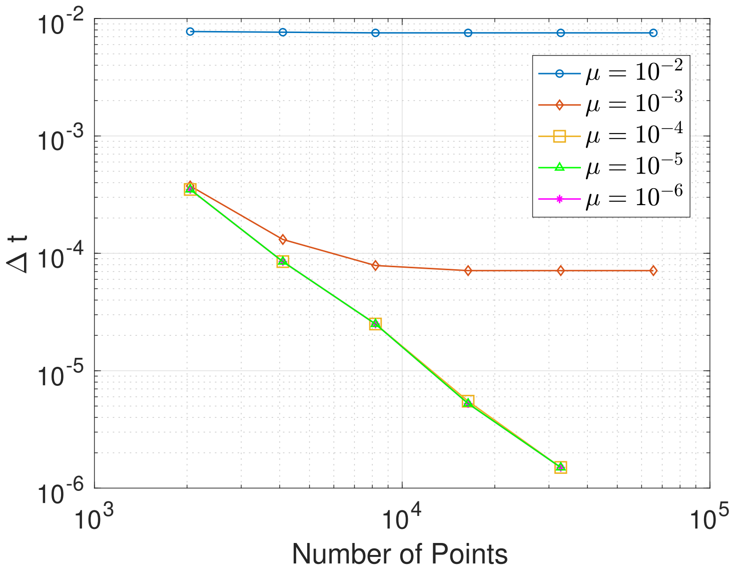


Figure 3.6: Reliable parameter interface for figure-of-eight vortex under different μ

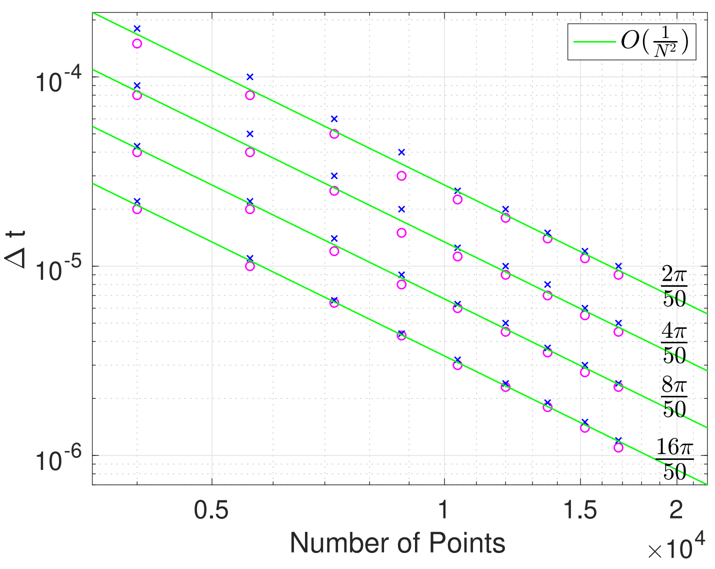


Figure 3.7: Reliable region for figure-of-eight vortex under $|\Gamma| = \frac{2\pi}{50}, \frac{4\pi}{50}, \frac{8\pi}{50}, \frac{16\pi}{50}$ and $\mu = 10^{-5}$. The red circles are the Type S parameters and the blue cross are the Type U parameters.

3.2 Estimate of Reconnection Time

In order to justify the ocurrence of vortex filament reconnection, the minimum separation $D_{\min}$ which measures the distance between the skewed parts of two branches is introduced. Fig. 3.8 is the development of the square of minimum distances $(D_{\min})^2$ with figure-of-eight vortex under several N, while Fig. 3.9 is its magnification around reconnection moments. These figures suggest the results of $D_{\min} \sim (t^*-t)^{1/2}$ for each N and Δt where t^* is the reconnection time depending on N. A scaling property

$$D_{\min}(t) \sim \sqrt{|\Gamma|(t_c - t)} \quad \text{as } t \to t_c \tag{3.6}$$

under the assumption of existence of the reconnection time t_c has also been observed in many studies [24, 25] and can be verified numerically. To obtain the result in Fig. 3.8 and Fig. 3.9, parameters we used and the computational times are listed in Table 2.9. The discretization parameters N and Δt are chosen from *Type S* identically the red circles in Fig. 3.5.

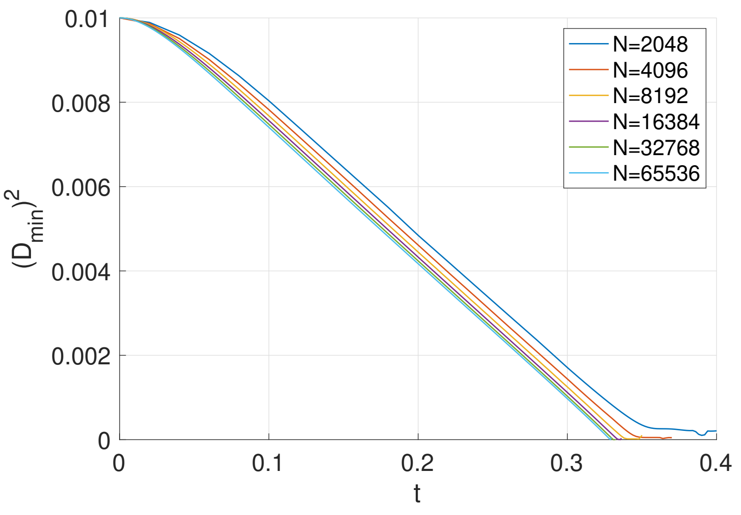


Figure 3.8: Development of the square of minimum distance $(D_{\min})^2$

We apply the least square method to find an estimated line for the square of minium distance $(D_{\min})^2$ on the interval $[0.30, 0.32]$. From the crossing point of approximated line and the horizontal axis, we get the estimated reconnection time $t^* = t^*(N)$. Fig. 3.10 gives the approximated straight line and the estimated reconnection time for the Fig. 3.8. Table 3.1 lists the estimated reconnection time t^* respect to N.

In order to support the convergence of the method, let the estimated reconnection time be t_{erc} of the vortex filament evolution (1.1) and suppose the error $|t^*(N) - t_{\mathrm{erc}}|$ has relation

$$|t^*(N) - t_{\mathrm{erc}}| = O\left(N^{-\alpha}\right). \tag{3.7}$$

Since t_{erc} and α are unknown, we find the optimal t_{erc} and α such that the error $|t^*(N) - t_{\mathrm{erc}}|$ fits the relation (3.7) by linear regression with logarithmic transformation. In this research, we use the golden section search on a specified interval and

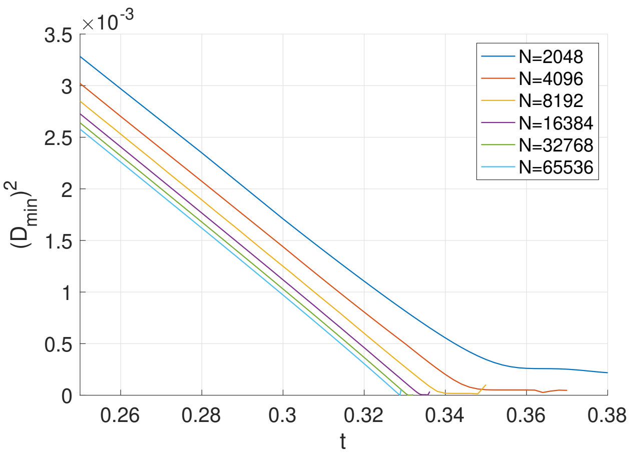


Figure 3.9: Development of $(D_{\min})^2$ near estimated reconnection times

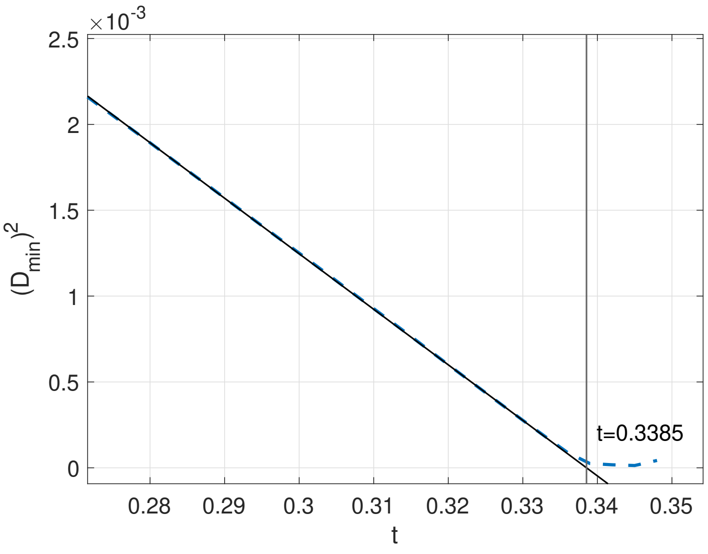


Figure 3.10: The approximated line and estimated reconnection time $t = 0.3385$ is obtained for the figure-of-eight vortex ($N = 8192$)

Table 3.1: Estimated reconnection time t^* in different N

N	t^*	N	t^*
2400	0.3554	16800	0.3339
4800	0.3436	19200	0.3333
7200	0.3397	21600	0.3327
9600	0.3373	24000	0.3323
12000	0.3359	26400	0.3319
14400	0.3348	28800	0.3316

use the norm of the residuals of least square method as cost function to find the t_{erc} such that the error is minimized. As the result, the estimated reconnection time $t_{\text{erc}} = 0.3266$ and order $\alpha = 0.6990$ are derived by N and t^* presented in Table 3.1. In Fig. 3.11, we plot the relation (3.7) by the read line with data in Table 3.1 by blue circles. In order to verify the assumption, we also plot $(N, t^*) = (32768, 0.3311)$ and $(65536, 0.3293)$ which are not used in fitting by red squares. In order to establish the goodness-of-fit, we use the coefficient of determination R^2 by measuring the error between $|t^*(N) - t_{\text{erc}}|$ and estimated value from $O(N^{-\alpha})$.

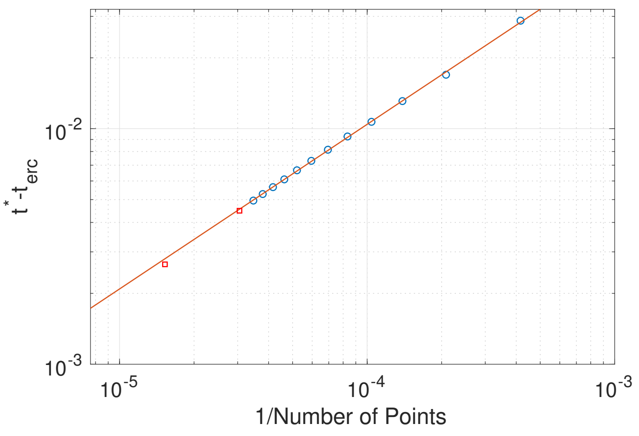


Figure 3.11: Convergence of errors $|t^*(N) - t_{\text{erc}}|$ under figure-of-eight vortex

With our estimate, $t_{\text{erc}} = 0.3266$ is a lot far from our largest computational result $t^* = 0.3292$ for $N = 65536$. If we reproduce the result with error less than 10^{-3}, we need around $N = 350000$ which is 5 times as large as the largest parameter we examined.

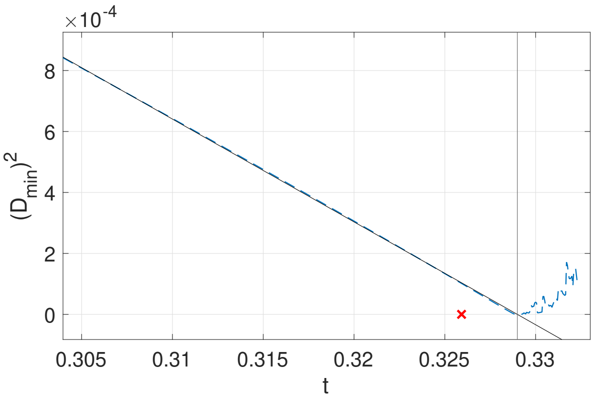


Figure 3.12: Comparison of the $t_{\text{erc}} = 0.3266$ indicated by the red cross with the estimated reconnection time $t^* = 0.3292$ for $N = 65536$

Moreover, we also use Richardson extrapolation to estimate the reconnection time. We estimate convergence rate α by following approximation,

$$t_{\text{erc}}^{(\text{Richardson})} \approx t^*(14400) + \frac{t^*(14400) - t^*(2400)}{6^\alpha - 1} \approx t^*(28800) + \frac{t^*(28800) - t^*(2400)}{12^\alpha - 1}, \tag{3.8}$$

where $N = 2400, 14400, 28800$ are the smallest, middle, and the largest data in Table 3.1. The estimated reconnection time $t_{\text{erc}}^{(\text{Richardson})} = 0.3264$, and $\alpha = 0.6942$. As the result, our estimate by optimization method shows high correlation $R^2 = 0.9990$ and estimate by Richardson extrapolation is $R^2 = 0.9994$. Two methods give coincident result, thus proposed optimal approximation is reliable.

3.3 Estimate of Accumulation of Rounding Errors

In this section, we discuss the accumulation of rounding errors in double precision arithmetic quantitatively near the estimated reconnection time for the sake of reliability. Particularly, in the numerical evolution of the figure-of-eight vortex filament, we meet the singularity at the reconnection time. This kind of singularity usually causes a rapid accumulation of rounding errors. However, the standard GPU architecture supports single and double precision arithmetic. Therefore we compare results by using different computational environments as stated in Section 2.4: double precision on GPU, quadratic precision on CPU, and multiple precision on CPU.

We measure the errors between double precision arithmetic and 50 decimal digits by

$$\max_j \frac{\left|\boldsymbol{x}_{t,j}^{(\text{double})} - \boldsymbol{x}_{t,j}^{(\text{50 digits})}\right|}{\left|\boldsymbol{x}_{t,j}^{(\text{50 digits})}\right|} \tag{3.9}$$

and those between quadruple precision arithmetic and 50 decimal digits by

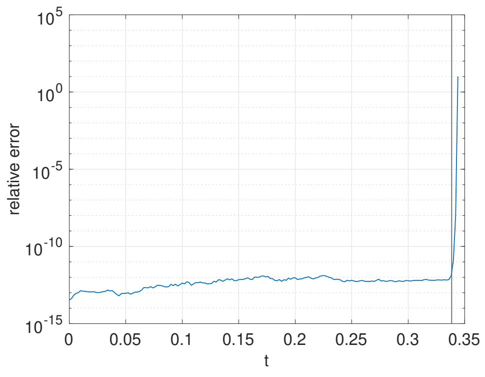
$$\max_j \frac{\left|\boldsymbol{x}_{t,j}^{(\text{quadruple})} - \boldsymbol{x}_{t,j}^{(\text{50 digits})}\right|}{\left|\boldsymbol{x}_{t,j}^{(\text{50 digits})}\right|}, \tag{3.10}$$

where $\boldsymbol{x}_{t,j}^{(\text{double})}$, $\boldsymbol{x}_{t,j}^{(\text{quadruple})}$, and $\boldsymbol{x}_{t,j}^{(\text{50 digits})}$ are respectively numerical solutions under double, quadruple, and 50 decimal digits by multiple precision arithmetic. The results with $N = 8192$ and $\Delta t = 2 \times 10^{-5}$ are shown in Table 3.2 and Fig. 3.13. The results indicate that the rounding errors are not significant even after a short period of the estimated reconnection time $t^* = 0.3385$, while the rapid growth of errors are observed after reconnection for double precision and quadruple precision. Thus we conclude that the numerical simulation, including reconnection, is reliable in terms of rounding errors.

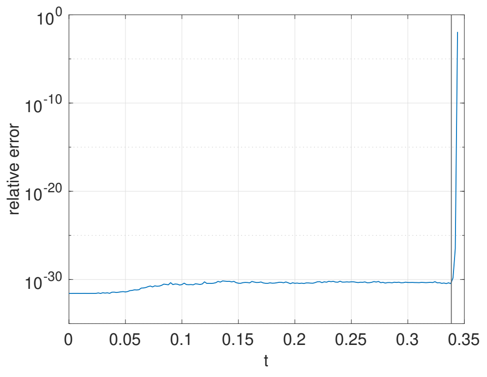
Table 3.2: Maximum relative errors (3.9) and (3.10) of figure-of-eight vortex filament under double precision, quadruple precision with multiple precision, where $t^* = 0.3385$. ($N = 8192,\ \Delta t = 2 \times 10^{-5}$)

	Double (GPU)	Quadruple (CPU)	
0.320	6.45×10^{-13}	4.76×10^{-31}	before t^*
0.338	1.23×10^{-12}	3.44×10^{-31}	near t^*
0.342	9.58×10^{-9}	3.27×10^{-27}	after t^*
0.344	1.02×10^{1}	1.15×10^{-1}	after t^*

Table 3.3 gives the computational times with different arithmetic for $0 \leq t \leq 0.4$ ($= 20000\Delta t$) in the environment given in Table 2.1. In the table, the second column gives elapsed times to find tangent vectors by FFT, while the third column



(a) Errors (3.9) in double precision



(b) Errors (3.10) in quadruple precision

Figure 3.13: Maximum relative errors (3.9) and (3.10) of figure-of-eight vortex filament under double precision, quadruple precision with multiple precision ($N = 8192,\ \Delta t = 2 \times 10^{-5}$) for $0 \le t \le 0.344$. The vertical line is the estimated reconnection time $t^* = 0.3385$.

gives those to calculate numerical integration and time stepping. Computation of quadruple precision and multiple precision are still taken much more time than GPU computation. It is not practical to investigate the rounding errors with case $N = 65536$.

Table 3.3: Elapse times of different precision with figure-of-eight vortex filament, $N = 8192,\ \Delta t = 2 \times 10^{-5}$ for $0 \le t \le 0.4$

	Tangent	Sum.	Total
double (16 digits, GPU, 5120 cores)	7s	5m 27s	5m 36s
double (16 digits, 32 threads)	41s	25m 40s	26m 30s
quadmath (34 digits, 32 threads)	12m 17s	1d 19h 01m	1d 19h 14m
exflib (58 digits, 32 threads)	19m 21s	2d 15h 51m	2d 16h 10m

3.4 Convergence of Numerical Solutions

This section is devoted to investigating the convergence of numerical solutions based on numerical experiments. This gives another standpoint to ensure the reliability of the results obtained by the proposed scheme (2.1).

Let $\boldsymbol{x}_{t,i}^{(N)}$, $0 \leq i < N$, be numerical solutions corresponding to $\boldsymbol{x}(t, \theta_i)$ under the spatial discretization parameter N and double precision arithmetic. We define

$$\max_k |\boldsymbol{x}_{t,k}^{(N)} - \widetilde{\boldsymbol{x}}_{t,k}^{(N)}|, \tag{3.11}$$

where $\widetilde{\boldsymbol{x}}_{t,k}^{(N)} = \boldsymbol{x}_{t,(65536/N)k}^{(65536)}$ which maps the fine discretization points to those in the specified N. Since it is hard to obtain the exact solution, the error measures difference by the numerical results obtained under the maximum possible parameter $N = 65536$, which is assumed to be the closest to the exact solution.

Firstly, we fix $\mu = 10^{-2}$ and $\Delta t = 7.5 \times 10^{-3}$. Fig. 3.14 is the error at $t = 0.15$ $(= 20\Delta t)$. In this case, we meet the error at 10^{-12}, which is at the rounding error level. In Fig. 3.14, we also compare the error under multiple precision. For the case $N = 32768$, its error between $N = 65536$ is less than 10^{-40}. Note that the horizontal axis is in the linear scale, while the vertical axis is in the logarithmic scale. It means exponential decay of maximum error with respect to N estimated by error $\approx 0.02444 \times 0.9969^N$. On the other hand, in approximation to (1.1) by

$$\boldsymbol{v}_{t,i} = -\frac{\Gamma}{4\pi} \sum_{j=0}^{N-1} \frac{(\boldsymbol{x}_{t,i} - \boldsymbol{x}_{t,j}) \times \frac{\partial \boldsymbol{x}_{t,j}}{\partial s}}{\left(|\boldsymbol{x}_{t,i} - \boldsymbol{x}_{t,j}|^2 + \mu^2\right)^{3/2}} \Delta s_j, \quad 0 \leq i < N \tag{3.12}$$

where Δs_i is defined by (2.6), the errors show $O(N^{-2.131})$ convergence. The result is exhibited in Fig. 3.15, where both horizontal and vertical axes are in the logarithmic scale.

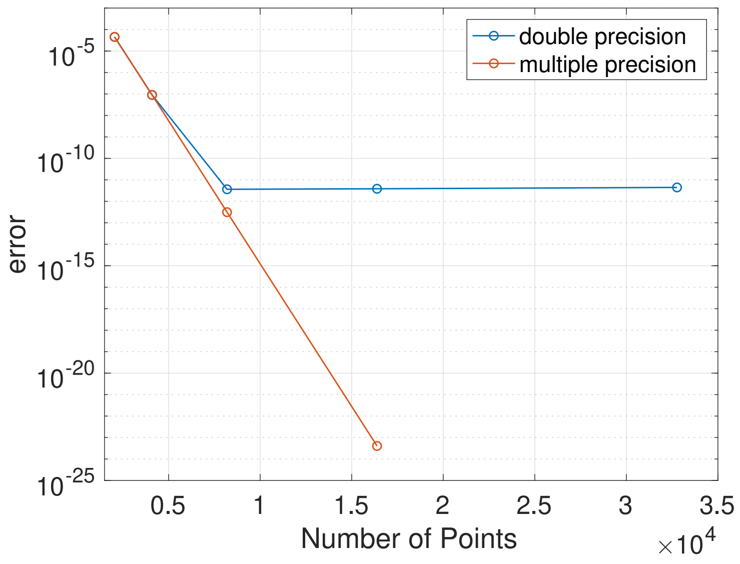


Figure 3.14: Errors in log scale versus N when $\mu = 10^{-2}$, $t = 0.15$ with scheme (2.1)

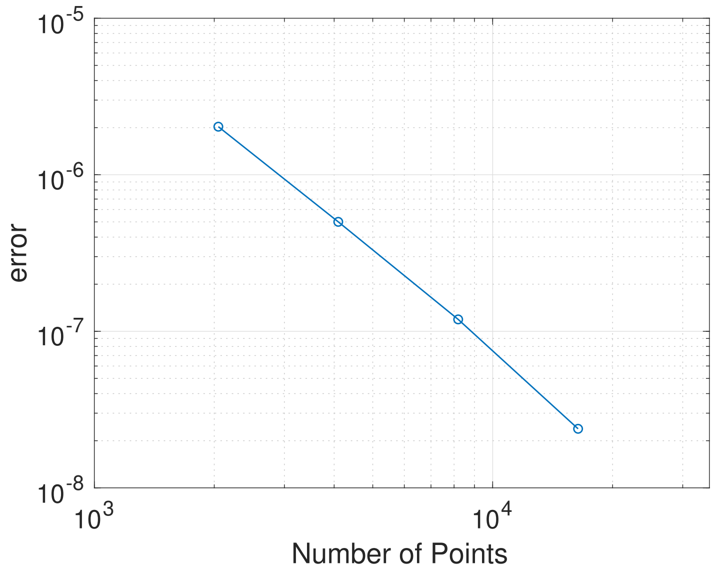


Figure 3.15: Log-log plot of errors versus N when $\mu = 10^{-2}$, $t = 0.15$ with scheme (3.12)

Secondly, we also choose $\mu = 10^{-2}$ but fix $N = 8192$ and change Δt to measure the error of temporal discretization by

$$\max_k |\boldsymbol{x}_{t,k}^{(\Delta t)} - \boldsymbol{x}_{t,k}^{(\Delta t_c)}|. \tag{3.13}$$

We use $\Delta t_c = 4.6875 \times 10^{-4}$ as reference and measure the error at $t = 0.15$ under $\Delta t = 7.5 \times 10^{-3},\ 3.75 \times 10^{-3}, 1.875 \times 10^{-3}$, and 9.375×10^{-4}. Fig. 3.16 shows the $O(\Delta t^4)$ convergence which consists with the order of the Runge-Kutta method. Note that errors come from temporal discretization are smaller than those come from spatial discretization under scheme (3.12).

Finally, we choose $\mu = 10^{-5}$. The errors along to different N is shown in Fig. 3.17. In this case, we use the maximum Δt which could meet numerical reconnection without turbulence. In this case, the error is also observed to decay exponentially as N goes large.

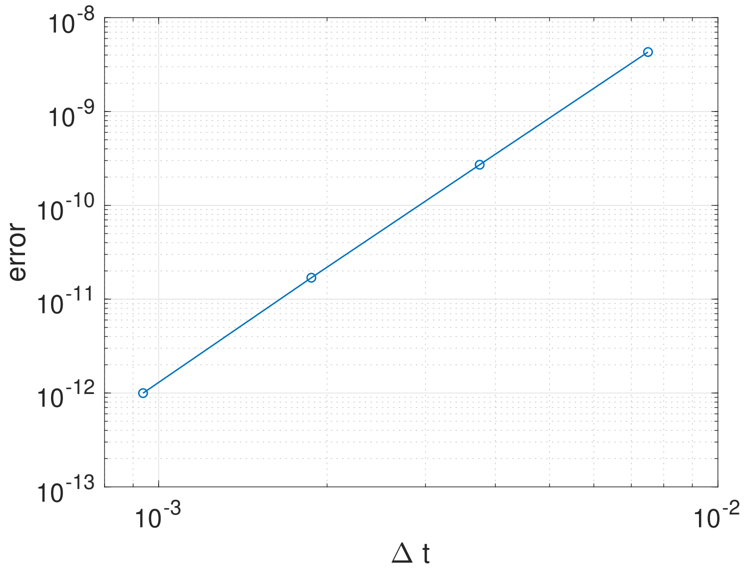


Figure 3.16: Log-log plot of errors at $t = 0.15$ versus Δt when $\mu = 10^{-2}$, $N = 8192$ with scheme (2.1)

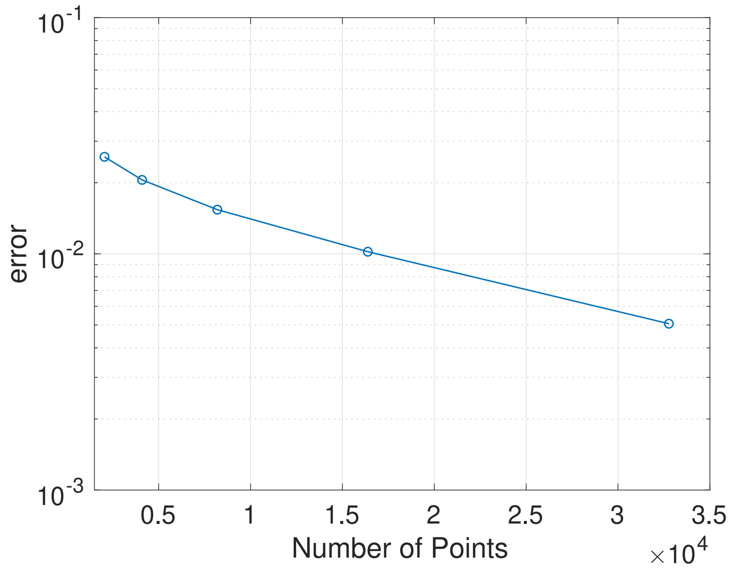


Figure 3.17: Errors in log scale versus N when $\mu = 10^{-5}$, $t = 0.15$ with scheme (2.1)

Chapter 4

Unbounded case : Tent-shaped Vortex Filament

In this chapter, we will give another case study of the proposed GPU computation to unbounded curves, a tilted hyperbola with two branches called tent-shaped, which has been proposed by Y. Kimura and H. K. Moffatt [4]. It consists of two smooth unbounded curves, and its initial shape is depicted in Fig. 4.1, which shows the two branches $\boldsymbol{x}^{\mathbf{1}}$ in blue and $\boldsymbol{x}^{\mathbf{2}}$ in orange of a tent-shaped vortex filament.

$$\boldsymbol{x}^{\mathbf{1}} : \begin{cases} x(p) = c\cosh p\cos\theta \\ y(p) = (c/m)\sinh p \\ z(p) = -c\cosh p\sin\theta \end{cases} \quad \boldsymbol{x}^{\mathbf{2}} : \begin{cases} x(p) = -c\cosh p\cos\theta \\ y(p) = (c/m)\sinh p \\ z(p) = -c\cosh p\sin\theta \end{cases} \tag{4.1}$$

with $c = 0.1,\ m = 0.35,\ \theta = \pi/4$ and $p \in \mathbb{R}$.

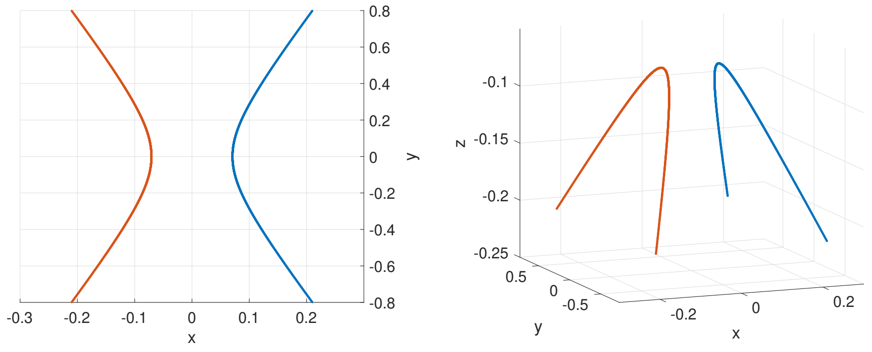


Figure 4.1: Initial shape of the tent model

Let us recall the regularized Biot–Savart law for multiple vortex filaments

$$\boldsymbol{v}_\mu(t, \boldsymbol{x}) = -\sum_{k=1}^{M} \frac{\Gamma_k}{4\pi} \int_{I_k} \frac{\left(\boldsymbol{x} - \boldsymbol{r}_\mu^k(t,\xi)\right) \times \frac{\partial \boldsymbol{r}_\mu^k}{\partial \xi}\ g(|\boldsymbol{x} - \boldsymbol{r}_\mu^k(t,\xi)|/\sigma_k)}{|\boldsymbol{x} - \boldsymbol{r}_\mu^k(t,\xi)|^3} \mathrm{d}\xi, \tag{4.2}$$

where $g(y) = y^3/(y^2 + \alpha)$ and $\alpha = (\sigma_i/\mu)^2$ give Rosenhead regularization. In the numerical computation, a finite number of points are distributed on the initial shape of the tent model (4.1), and the velocity of each point given by (4.2) is approximately calculated to find the evolution of the curve. Let M be a positive integer, and $h = \pi/(2M)$. Using the distribution appeared in the double exponential rule [26], we adopt parameterization

$$p_i = \sinh\left(\frac{\pi}{2}\sinh(ih)\right) \quad -M \le i \le M,\ h > 0, \tag{4.3}$$

and Δp_i defined by

$$\Delta p_i = \frac{\pi}{2} h \cosh(ih) \cosh\left(\frac{\pi}{2}\sinh(ih)\right). \tag{4.4}$$

Note that the number of total points on $\boldsymbol{x^1}_\mu$ and $\boldsymbol{x^2}_\mu$ is $N = 4M + 2$. The velocity at discretized point is calculated by

$$\boldsymbol{v^1}_{t,i} = -\frac{\Gamma_1}{4\pi}\sum_{j=-M}^{M}\frac{(\boldsymbol{x^1}_{t,i} - \boldsymbol{x^1}_{t,j}) \times \boldsymbol{\tau^1}_{t,j}}{\left(|\boldsymbol{x^1}_{t,i} - \boldsymbol{x^1}_{t,j}|^2 + \mu^2\right)^{3/2}}\Delta p_j - \frac{\Gamma_2}{4\pi}\sum_{j=-M}^{M}\frac{(\boldsymbol{x^1}_{t,i} - \boldsymbol{x^2}_{t,j}) \times \boldsymbol{\tau^2}_{t,j}}{\left(|\boldsymbol{x^1}_{t,i} - \boldsymbol{x^2}_{t,j}|^2 + \mu^2\right)^{3/2}}\Delta p_j, \tag{4.5}$$

$$\boldsymbol{v^2}_{t,i} = -\frac{\Gamma_2}{4\pi}\sum_{j=-M}^{M}\frac{(\boldsymbol{x^2}_{t,i} - \boldsymbol{x^2}_{t,j}) \times \boldsymbol{\tau^2}_{t,j}}{\left(|\boldsymbol{x^2}_{t,i} - \boldsymbol{x^2}_{t,j}|^2 + \mu^2\right)^{3/2}}\Delta p_j - \frac{\Gamma_1}{4\pi}\sum_{j=-M}^{M}\frac{(\boldsymbol{x^2}_{t,i} - \boldsymbol{x^1}_{t,j}) \times \boldsymbol{\tau^1}_{t,j}}{\left(|\boldsymbol{x^2}_{t,i} - \boldsymbol{x^1}_{t,j}|^2 + \mu^2\right)^{3/2}}\Delta p_j, \tag{4.6}$$

where $-M \le i \le M$, $\boldsymbol{v^1}_{t,i}$ and $\boldsymbol{v^2}_{t,i}$ are velocities at $\boldsymbol{x^1}_{t,i} = \boldsymbol{r}^1_\mu(t, p_i)$ and $\boldsymbol{x^2}_{t,i} = \boldsymbol{r}^2_\mu(t, p_i)$ respectively, and $\boldsymbol{\tau^k} = \partial \boldsymbol{x^k}_\mu / \partial p$ is the tangent vector and it is calculated by the fourth-order finite difference. Note that for $f \in C^6(\mathbb{R})$, we have

$$\begin{aligned} f'(x) - \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} \\ = \frac{1}{30} f^{(5)}(x)h^4 + O(h^6) \end{aligned} \tag{4.7}$$

and

$$\begin{aligned} f'(x) - \frac{-3f(x+4h) + 16f(x+3h) - 36f(x+2h) + 48f(x+h) - 25f(x)}{12h} \\ = \frac{1}{5} f^{(5)}(x)h^4 + O(h^5) \end{aligned} \tag{4.8}$$

as $h \to 0$. Moreover, two branches perform as symmetrical to the plane $x = 0$, and we adopt replicating algorithm to reduce the computation. Throughout this chapter, $\Gamma_1 = -\Gamma_2 = 4\pi/50$ and $\mu = 10^{-5}$ are used unless otherwise stated.

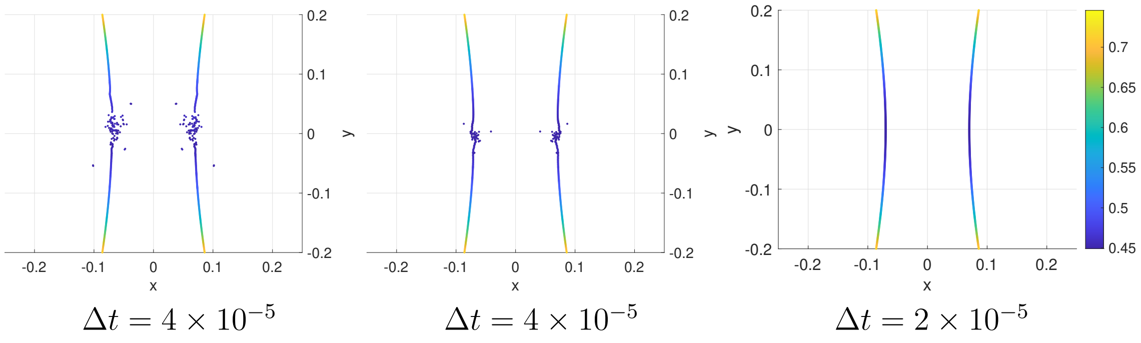
Note that the supremum norm is not applicable to the tent-shaped vortex filament due to its unboundedness. In fact, maximum errors defined by (3.11) are attained almost at the edge of truncated curves, and rounding errors are dominant in double precision arithmetic.

4.1 Reliability Criteria

In this section, we discuss the choice of reliable parameter pairs N and Δt to reproduce the numerical reconnection.

Fig. 4.2 shows the results for at $t = 0.006$ with different sizes of Δt. Fig. 4.2a and Fig. 4.2b are numerical solutions calculated with $\Delta t = 4 \times 10^{-5}$ by double and multiple precision arithmetic respectively, and they have disturbance. However in Fig. 4.2c, the curve is smooth under $\Delta t = 2 \times 10^{-5}$, and we can reproduce the reconnection.

Fig. 4.3 depicts *Type S* and *Type U* pairs of N and Δt we have tested by the red circles, and blue crosses, respectively. The green line is the estimated interface of the *Type S* and *Type U* region by the least-square method. We state that the interface of reliable parameter pairs under the tent-shaped vortex is proportional to $1/N^2$.



(a) Type U, double precision (b) Type U, multiple precision (c) Type S, double precision

Figure 4.2: Numerical results by Type U and Type S parameter pairs ($t = 0.006$, $N = 3202$). The color bar corresponds to the distance calculated by (2.6).

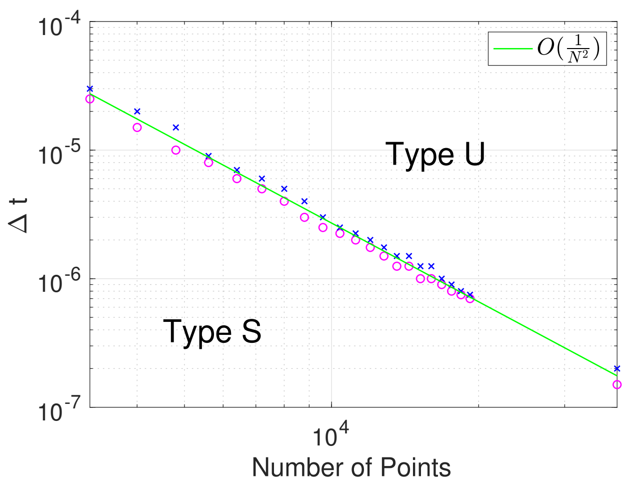


Figure 4.3: Reliable regions for the tent model

4.2 Estimate of Reconnection Time

As we discussed in Section 3.2, we introduce the minimum separation $D_{\min}$ and examine a scaling property (3.6). Fig. 4.4 depicts the development of the square of the minimum distance $(D_{\min})^2$. Computational times are listed in Table 4.1, where pairs of N and Δt are chosen from *Type S* examined in Fig. 4.3.

Fig. 4.5 depicts the regression line and the estimated reconnection time for $N = 9602$. For the profile before and after the estimated reconnection time $t^* = 0.4691$ with $N = 9602$ is shown in Fig. 4.6. The orange curve touches the blue one and the collision happens after $t = 0.4691$. As in Section 3.2, we set the fitting range by the time interval $[0.40, 0.42]$.

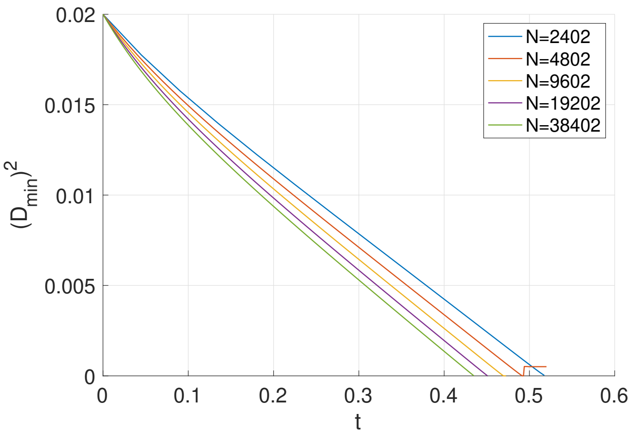


Figure 4.4: Development of the tent-shaped vortex along the curve of the square of minimum distance

Table 4.1: Elapse times for tent-shaped vortex filament under GPU computation $0 \le t \le 0.52$

N	Δt	Total
2402	4.5×10^{-5}	33s
4802	1×10^{-5}	5m 04s
9602	3×10^{-6}	35m 22s
19202	7×10^{-7}	5h 34m 32s
38402	1.5×10^{-7}	2d 23h 05m

We assume the convergence relation (3.7) and verify the assumption based on the numerical computation. Similarly to Section 3.2, the estimated reconnection time $t^*(N)$ shown in Table 4.2 gives $t_{\text{erc}} = 0.3467$, and $\alpha = 0.2381$. As define in Section 3.2, coefficient of determination $R^2 = 0.99998$. Fig. 4.7 depicts the regression line and $|t^* - t_{\text{erc}}|$. Moreover, we estimate reconnection time by Richardson extrapolation with $t^*(2402), t^*(19202)$, and $t^*(38402)$ which are similar to (3.8). We have the estimate with $t_{\text{erc}}^{(\text{Richardson})} = 0.3433$, $\alpha = 0.2324$, and $R^2 = 0.99994$. The results derived by Richardson extrapolation consist with those by the golden section search, and thus both results are validated.

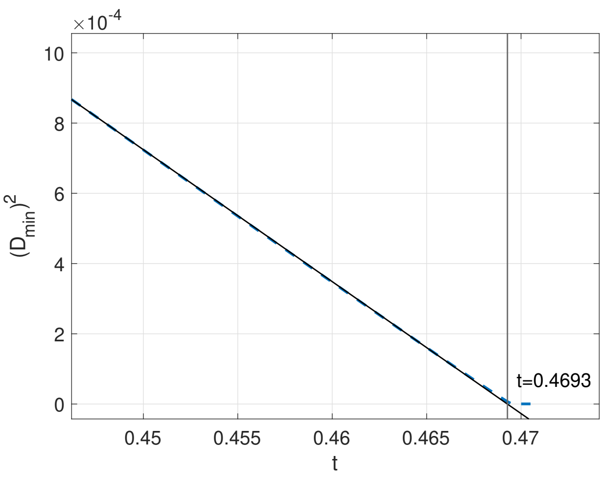
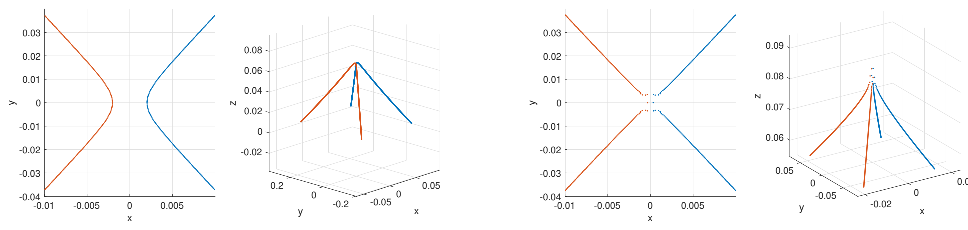


Figure 4.5: The regression line and estimated reconnection time $t = 0.4691$ is obtained for the tent-shaped vortex ($N = 9602$)



(a) $t = 0.4690$ (b) $t = 0.4696$

Figure 4.6: Reconnection of tent-shaped vortex near the estimated Reconnection Time $t = 0.4691$ ($N = 9602$)

In Fig. 4.8, we plot the estimated t_{erc} by the red cross and the decay of minimum separations $(D_{\text{min}})^2$. In the case of tent model, the convergence of reconnection time is far from our computed largest result $N = 38402$.

Table 4.2: Estimate of reconnection time t^* in different N

N	t^*	N	t^*
2402	0.5171	21602	0.4476
4802	0.4907	24002	0.4451
7202	0.4775	26402	0.4429
9602	0.4689	28802	0.4409
12002	0.4627	31202	0.4391
14402	0.4578	33602	0.4374
16802	0.4538	36002	0.4359
19202	0.4505	38402	0.4345

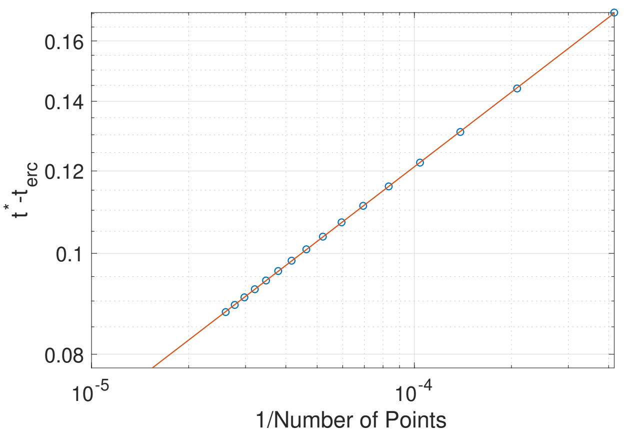


Figure 4.7: The error $|t^*(N) - t_{erc}|$ versus $1/N$ under tent-shaped vortex filaments where $t_{\mathrm{erc}} = 0.3466$

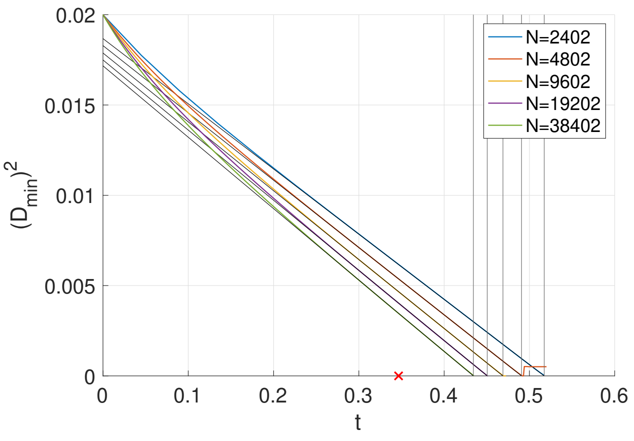


Figure 4.8: Comparison of the $t_{\mathrm{erc}} = 0.3466$ indicated by the red cross with the estimated reconnection time $t^* = 0.4346$ for $N = 38402$

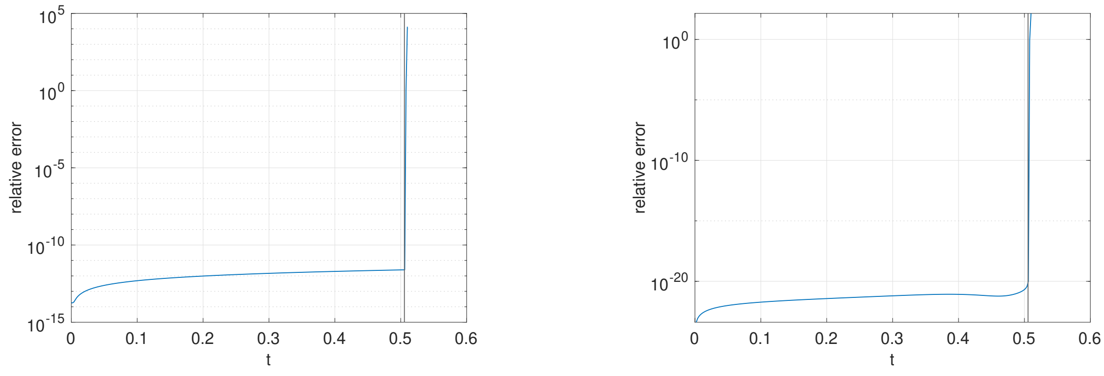
4.3 Estimate of Accumulation of Rounding Errors

In this section, we measure the rounding errors by changing the precisions of arithmetic. In Table 4.3, the maximum relative errors similar to (3.9) and (3.10) under tent-shaped vortex filament are presented. For the relative error in $0 \leq t \leq 0.52$, we see the rapid growth of rounding errors after the reconnection time illustrated in Fig. 4.9. In the tent-shaped case, $N = 3202$ and $\Delta t = 2 \times 10^{-5}$ are used. The results also show that the rounding errors are not significant near the estimated reconnection time $t^* = 0.5055$. Thus we conclude that the numerical simulation, including reconnection, is reliable in terms of rounding errors.

Table 4.4 shows the computational times with different arithmetic for $0 \leq t \leq 0.52$ $(= 26000\Delta t)$ where $N = 3202$ and $\Delta t = 2 \times 10^{-5}$ are used.

Table 4.3: Maximum relative errors of tent-shaped vortex filaments under double, quadruple precision with multiple precision, where $t^* = 0.5055$. ($N = 3202,\ \Delta t = 2 \times 10^{-5}$)

	Double (GPU)	Quadruple (CPU)	
0.4500	2.20×10^{-12}	6.21×10^{-22}	before t^*
0.5040	2.46×10^{-12}	3.85×10^{-21}	near t^*
0.5060	2.47×10^{-12}	9.95×10^{-21}	after t^*
0.5100	1.36×10^{4}	1.41×10^{2}	after t^*



(a) Double precision

(b) Quadruple precision

Figure 4.9: Maximum relative errors of tent-shaped vortex filaments under double, quadruple precision with multiple precision ($N = 3202,\ \Delta t = 2 \times 10^{-5}$) for $0 \leq t \leq 0.51$. The vertical line is the estimated reconnection time $t^* = 0.5055$.

Table 4.4: Elapse times of different precision with tent-shaped vortex filaments with $N = 3202$ and $\Delta t = 2 \times 10^{-5}$

unit: sec.

	Total
double (16 digits, GPU, 5120 cores)	1m 42s
double (16 digits, 32 cores)	3m 36s
quadmath (34 digits, 32 cores)	5h 6m 54s
exflib (58 digits, 32 cores)	7h 53m 48s

Chapter 5

Conclusion

In this thesis, we discuss fast and reliable numerical computation for simulation of vortex filament evolution described by (1.1). We propose the efficient use of Graphics Processing Units (GPUs), which enable us to verify the reliability of numerical results by performing and comparing computations with various computation parameters.

Firstly, we have introduced design and implementation of vortex filament evolutions on GPU, which is equipped with thousands of cores and is suitable for parallel computations. In this thesis, we employ the numerical scheme (2.1) for the numerical computation of vortex filament evolution. It has $O(N^2)$ computational complexity when the vortex filament is approximated by N points. We optimize computation performance by using shared memory. We also optimize the choice of warp sizes by measuring computational times with several combinations of possible warp sizes. As a result, we achieve 8.58 times faster than supercomputer computation, and we push forward twice the nodal points N than the previous studies by Kimura and Moffatt.

Secondly, we study the reliability of the employed scheme (2.1) with figure-of-eight vortex and tent-shaped vortex. Numerical stability of schemes is essential in time evolution problems. However, it is challenging to evaluate the reliability of schemes by mathematical analysis. Thus, we investigate reliability parameter pairs with the aid of massive GPU computations. Moreover, a rigorous estimate of reconnection time is also hard. We estimate the reconnection time by examining numerical solutions precisely. Finally, we compare numerical solutions under GPU computation with standard double precision arithmetic with quadruple precision and multiple precision by exflib. From the results, we verify the reliability of numerical solutions in terms of discretization and rounding errors.

Bibliography

[1] L. Rosenhead, “The spread of vorticity in the wake behind a cylinder,” *Proc. R. Soc. London. Ser. A, Contain. Pap. a Math. Phys. Character*, vol. 127, no. 806, pp. 590–612, 1930.

[2] L. C. Berselli and H. Bessaih, “Some results for the line vortex equation,” *Nonlinearity*, vol. 15, no. 6, pp. 1729–1746, 2002.

[3] Y. Kimura and H. K. Moffatt, “Scaling properties towards vortex reconnection under Biot-Savart evolution,” *Fluid Dyn. Res.*, vol. 50, no. 1, 2018.

[4] ——, “A tent model of vortex reconnection under Biot-Savart evolution,” *J. Fluid Mech.*, vol. 834, pp. 1–12, 2018.

[5] NVIDIA Corporation, “CUDA C++ Programming Guide, release: 10.2.89,” 2017. https://docs.nvidia.com/cuda/archive/10.2/cuda-c-programming-guide/index.html

[6] H. Nguyen, *GPU Gems 3*, 1st ed. Addison-Wesley Professional, 2007.

[7] L. Greengard and V. Rokhlin, “A fast algorithm for particle simulations,” *J. Comput. Phys.*, vol. 73, no. 2, pp. 325–348, 1987.

[8] A. J. Majda and A. L. Bertozzi, *Vorticity and Incompressible Flow.* Cambridge University Press, nov 2001.

[9] H. Helmholtz, “Über Integrale der hydrodynamischen Gleichungen, welche den Wirbelbewegungen entsprechen.” *J. für die reine und Angew. Math.*, vol. 55, pp. 25–55, 1858.

[10] K. Moffatt, “Vortex Dynamics: The Legacy of Helmholtz and Kelvin,” in *IUTAM Symp. Hamiltonian Dyn. Vor. Struct. Turbul.* Dordrecht: Springer Netherlands, 2007, pp. 1–10.

[11] H. Lamb, *Hydrodynamics.* University Press, 1895.

[12] L. S. Da Rios, “Sul moto d’un liquido indefinito con un filetto vorticoso di forma qualunque,” *Rend. del Circ. Mat. di Palermo*, vol. 22, no. 1, pp. 117–135, 1906.

[13] A. Leonard, “Vortex methods for flow simulation,” *J. Comput. Phys.*, vol. 37, no. 3, pp. 289–335, 1980.

[14] C. Greengard, “Convergence of the Vortex Filament Method,” *Math. Comput.*, vol. 47, no. 176, p. 387, 1986.

[15] J. Soler, “Vortex filament method,” *IMA J. Numer. Anal.*, vol. 10, no. 1, pp. 75–102, 1990.

[16] W. Thomson, “VI.—On Vortex Motion,” *Trans. R. Soc. Edinburgh*, vol. 25, no. 1, pp. 217–260, 1868.

[17] NVIDIA Corporation, “Nvidia Tesla V100 GPU Volta Architecture,” 2017.

[18] ——, “Cufft Library User’s Guide,” 2013. https://developer.nvidia.com/cuda-toolkit-55-archive

[19] “ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic,” 1985.

[20] “IEEE 754–2008 Standard for Floating-Point Arithmetic,” 2008.

[21] N. Whitehead and A. Fit-Florea, “Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs,” 2011.

[22] “GCC libquadmath.” https://gcc.gnu.org/onlinedocs/libquadmath/

[23] H. Fujiwara, “exflib: multiple-precision arithmetic library.” http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/

[24] E. D. Siggia, “Collapse and amplification of a vortex filament,” *Phys. Fluids*, vol. 28, no. 3, pp. 794–805, 1985.

[25] A. T. De Waele and R. G. Aarts, “Route to vortex reconnection,” *Phys. Rev. Lett.*, vol. 72, no. 4, pp. 482–485, 1994.

[26] H. Takahasi and M. Mori, “Double exponential formulas for numerical integration,” *Publ. Res. Inst. Math. Sci.*, vol. 9, no. 3, pp. 721–741, 1974.