令和7年度 卒業論文

# CUDAを用いた超伝導体内の磁束線運動の並列高速シミュレーション

九州工業大学情報工学部

物理情報工学科 電子物理工学コース

学生番号 222C3058

杉本 陸

指導教員：小田部 荘司

## 要旨

本研究では、超伝導体内の磁束線運動を記述する時間依存 Ginzburg-Landau (TDGL) 方程式のシミュレーションにおいて、CUDA を用いた GPGPU による高速化を実現した。従来、計算コストの高さが課題であったが、GPU の並列計算能力を活用することで、2 次元シミュレーションにおいて約 99 倍、3 次元シミュレーションにおいて約 191 倍の高速化を達成した。開発においては、まず Processing で各格子における物理量を CSV ファイルとして書き出し、Java による CPU 実装で CSV ファイルと値を比較しつつ数値的な正しさを検証し、その後 GPU 実装へ移行する段階的な手法を採用し、信頼性の高いシミュレーションシステムを確立した。この高速化により、シミュレーションのパラメータを変化させながら現象を観察するような試行錯誤的な研究プロセスや、リアルタイムで直感的な可視化が可能となるなど、研究や理解の効率化が期待できる。

# 目 次

# 第1章　序論

## 1.1　研究の背景と目的

超伝導体における渦糸の生成・消滅や相転移といった動的現象の解明は、基礎物理学のみならず、次世代超伝導デバイスの設計においても極めて重要である。これらの非平衡・非線形現象を記述する有効な理論として時間依存 Ginzburg-Landau（TDGL）方程式が知られている。TDGL 方程式を用いることで、磁場中での超伝導体の振る舞いを詳細に解析することが可能となる。

しかし、TDGL 方程式の数値シミュレーションは膨大な計算コストを伴うという問題がある。特に、大規模な格子系を扱う場合や、より現実的な 3 次元モデルへの拡張を行う場合、計算量は格子点数の増加に伴い急激に増大する。従来の CPU を用いた逐次計算では、十分な時間発展や大規模なパラメータ探索を行うために非現実的な計算時間を要することが、研究を進める上での大きな障壁となっていた。

実際に、2 次元や 3 次元のシミュレーションは既に実装されて行われていたが、計算量が膨大であるため、シミュレーション結果が得られるまで何十時間という長時間を要していた。これではシミュレーションのパラメータを変化させながら現象を観察するような試行錯誤的な研究プロセスが困難である。本研究ではその問題に注目し、根本から解決するためのシステムを構築した。

本研究の目的は、CUDA (Compute Unified Device Architecture) を用いた GPGPU (General-Purpose computing on Graphics Processing Units) 技術により、この計算ボトルネックを抜本的に解消することである。GPU が持つ大規模な並列計算能力を活用し、2 次元および 3 次元の TDGL 方程式シミュレーションを飛躍的に高速化することを目指す。

## 1.2　本論文の構成

本論文は全 5 章で構成される。第 2 章では、本研究の基礎となる Ginzburg-Landau 方程式の理論、および GPGPU と CUDA の基本原理、シミュレーションの可視化技術について述べる。第 3 章では、システム設計と実装の詳細について説明する。開発アプローチ、シミュレーションアルゴリズムの並列化手法、およびリアルタイム可視化システムの構築について詳述する。第 4 章では、数値的正確性の検証とパフォーマンス評価の結果を述べ、AI を活用した開発プロセスについても考察を行う。最後に第 5 章で本研究の結論を述べ、今後の課題について展望する。

# 第2章　基礎理論

## 2.1　Ginzburg-Landau 方程式

超伝導現象を記述する現象論的な理論として、Ginzburg-Landau (GL) 方程式が広く知られている。本研究では、この GL 方程式を時間発展させた時間依存 Ginzburg-Landau (TDGL: Time Dependent Ginzburg-Landau) 方程式を用いる。

### 2.1.1　TDGL 方程式

TDGL 方程式は、超伝導電子対の密度を表す複素オーダーパラメータ $\Psi$ と、電磁場を表すベクトルポテンシャル $\mathbf{A}$ の時間発展を記述する。無次元化された TDGL 方程式は以下の通りである。

$$\gamma\frac{\partial\Psi}{\partial t} = (\nabla - \mathrm{i}\mathbf{A})^2\Psi - \alpha\Psi - \beta|\Psi|^2\Psi \tag{2.1}$$

$$\tau_A\frac{\partial\mathbf{A}}{\partial t} = \mathrm{Im}[\bar{\Psi}(\nabla - \mathrm{i}\mathbf{A})\Psi] - \nabla\times\nabla\times\mathbf{A} \tag{2.2}$$

ここで、$\Psi$ は超伝導の状態を表す複素オーダーパラメータ、$\mathbf{A}$ はベクトルポテンシャルである。$\gamma$ と $\tau_A$ はそれぞれ秩序変数とベクトルポテンシャルの時定数であり、$\alpha$ と $\beta$ は GL 自由エネルギーを $|\Psi|$ のべき級数で展開したときの係数である。$\alpha$ の符号が超伝導相転移において重要になり、負だと超伝導状態が安定となる。

これらの式を数値的に解くことで、磁場中における超伝導体内の磁束線の運動や配置をシミュレートすることができる。

### 2.1.2　数値解法と離散化

TDGL 方程式を計算機上で解くためには、空間および時間を離散化する必要がある。本研究では、空間方向の離散化にリンク変数（Link Variable）を用いた手法を採用し、時間積分には Lie-Trotter-Suzuki (LTS) 分解と AFI (Affine Integrator) を組み合わせた手法を用いる。

空間離散化において、ゲージ不変性を保つために「リンク変数 (Link Variable)」を用いる手法が一般的である。格子点 $(i,j,k)$ におけるオーダーパラメータを $\psi_{i,j,k}$ とし、格子点間をつなぐリンク変数を $U^{\mu}_{i,j,k}$ $(\mu = x,y,z)$ と定義する。リンク変数はベクトルポテンシャル $A_\mu$ と以下の関係にある。

$$U^{\mu}_{i,j,k} = \exp\left(-i\int_{x_i}^{x_{i+1}} A_\mu dx_\mu\right) \approx \exp(-iA_\mu\Delta x) \tag{2.3}$$

このリンク変数を用いることで、ベクトルポテンシャルのゲージ変換に対して方程式の形式が不変に保たれ、数値的に安定した解を得ることが可能となる。

## 2.2 シミュレーション関連技術

### 2.2.1 GPU コンピューティング

GPU (Graphics Processing Unit) は、元来画像処理や画面描画を高速に行うために開発されたプロセッサである。CPU (Central Processing Unit) が少数のコアで複雑な順次処理を得意とするのに対し、GPU は数千から数万の小さなコアを持ち、大量のデータを並列に処理することを得意としている。この GPU の並列計算能力を画面描画以外の一般計算に応用する技術を GPGPU (General-Purpose computing on Graphics Processing Units) と呼ぶ。特に、格子状に並んだデータに対して同一の演算を行う TDGL シミュレーションのような数値計算は、GPU のアーキテクチャと非常に相性が良く、これを活用することにより大幅な高速化が期待できる。

### 2.2.2 CUDA アーキテクチャ

CUDA (Compute Unified Device Architecture) は、NVIDIA 社が提供する並列コンピューティングプラットフォームおよびプログラミングモデルである。CUDA では、GPU を「デバイス」、CPU を「ホスト」と呼び、ホストからデバイスへカーネル関数と呼ばれる計算処理を指示することで並列計算が実行される。

CUDA の並列実行モデルは SIMT (Single Instruction, Multiple Threads) と呼ばれる。これは 1 つの命令を多数のスレッドで同時に実行する方式である。このスレッド分割を意識してシミュレーション空間の各格子点などの計算対象を各スレッドに割り当てることで、効率的な並列化を実現することができる。

### 2.2.3 JCuda

本研究では、シミュレーションの制御を Java 言語で記述し、計算集約的な部分のみを CUDA で記述している。そのため Java から CUDA を利用する必要があり、本研究ではそれを可能とする JCuda ライブラリを採用した。JCuda を用いることで、Java プログラムから CUDA ドライバ API やランタイム API を直接呼び出すことが可能となり、Java の生産性と CUDA のパフォーマンスを両立させることができる。

## 2.3 シミュレーションの可視化技術

リアルタイムのシミュレーションにおいては、計算処理と描画処理を同一スレッドで行うと、描画処理の負荷が計算速度を制限してしまう場合がある。また、本研究では既に Processing による CPU シミュレーションが行えているものについて、それを GPGPU を用いて加速することに焦点を当てているため、描画処理に関しては Processing をそのまま使用するのが好ましいと判断した。

しかし、Processing では相性的に GPU を直接利用することができない。そこで、計算をバックエンド (Java/CUDA)、可視化をフロントエンド (Processing) として明確に分離し、両者をネットワーク (TCP/IP) で連携させることで、計算性能を最大限に維持しつつ、スムーズな可視化を実現する手法を採用した。

なお本研究の実装では、理論上それぞれ別のコンピューターにおける演算と描画を行うことが可能であるが、シミュレーション結果の転送に使用されるネットワーク帯域が一般的なネットワー

クでは不足するため、これらの処理は同一デバイスで行うことが好ましい。本研究においても、全て同一デバイスで実行した場合について議論する。

# 第3章　システム設計と実装

## 3.1　開発アプローチとシステム構成

既存の Processing における CPU シミュレーションをそのまま高速化するには、シミュレーション結果が変わっていないことを厳密に確認する必要がある。そこで、本研究では以下の段階的な開発手法を採用した。

1. **CSV 出力用 Processing 実装**: 既存の Processing シミュレーションで得られる数値的な結果を、各ステップごとに CSV ファイルとして出力するコードを作成
2. **Java CPU 実装**: Processing コードを参考に Java による CPU 実装を行い、CSV と比較しつつ数値的正当性を確立
3. **Java GPU 実装**: CUDA を用いた GPU 実装を開発し、Java CPU 実装のシミュレーション結果と値が一致することを確認

この段階的な開発手法により、既存の Processing でのシミュレーション結果と、最終的な GPU を用いたシミュレーション結果が同一であることを保証することができる。

また、本研究では可能な限りシミュレーションを高速化しつつ、結果として得られる画像やリアルタイム可視化を残すために、Java と CUDA が計算した結果を Processing で描画する手法を採用している。そのため、シミュレーションを行うシステム全体は、以下の 3 レイヤーに分割することができる。

- **計算層**: Java/CUDA によるシミュレーション実行環境
- **通信層**: シミュレーション結果を Java から Processing に転送するための通信
- **描画層**: Processing によるシミュレーション結果のリアルタイム描画

これらを組み合わせて動作させることで、従来の Processing によるシミュレーションと同様にリアルタイムでシミュレーション結果を可視化しつつ、シミュレーションを高速化する。

なお、これから第 3 章で述べる具体的な実装コードは、特に言及がない限り 2 次元シミュレーションの実装のみを挙げている。これは CUDA 連携や非同期でのシミュレーション結果の転送など、高速化に関するコードは 2 次元シミュレーションと 3 次元シミュレーションで共通する部分が多いためである。

## 3.2　Processing におけるシミュレーション結果の CSV 出力

段階的開発手法の第 1 段階として、既存の Processing シミュレーションで得られる各ステップの物理量を CSV ファイルとして出力する機能を実装した。この CSV 出力により、後続の Java CPU 実装や GPU 実装との数値比較が可能となり、実装の正しさを検証する基盤を確立した。

### 3.2.1 CSV 出力の実装

既存の Processing シミュレーションのコードを編集し、新たに export.pde ファイルを作成する。このファイルにおいて新しい関数を定義し、シミュレーションの各ステップにおいて、各格子点の物理量を CSV 形式で出力するように実装する。

具体的に、2 次元のシミュレーションの場合は以下の物理量を出力するように記述した。

- **オーダーパラメータ** $\psi$: 格子点 $(i, j)$ における複素オーダーパラメータ。格子範囲は $[0, N_x + 1] \times [0, N_y + 1]$
- **リンク変数** $w_x$: $x$ 方向のリンク変数。格子範囲は $[0, N_x] \times [0, N_y + 1]$。
- **リンク変数** $w_y$: $y$ 方向のリンク変数。格子範囲は $[0, N_x + 1] \times [0, N_y]$。

出力される CSV ファイルは `output/step_{step}.csv` に保存される。ファイルの中身は 1 行目がヘッダー行、2 行目以降がデータ行となっており、各物理量について、ステップ番号、物理量の種類（`psi`, `wx`, `wy`）、格子座標 $(i, j)$、複素数の実部、虚部をカンマ区切りで記録する。

100 ステップにおける CSV 出力ファイルである `output/step_100.csv` の先頭 12 行を例として以下に示す。

```
step,type,i,j,real,imag
100,psi,0,0,1.0,0.0
100,psi,0,1,0.9888082850297936,0.001418746010547655
100,psi,0,2,0.9886541171435255,0.0014865124804608043
100,psi,0,3,0.9883855838253205,0.0016210426193348864
100,psi,0,4,0.9879829932036462,0.0018267990279420378
100,psi,0,5,0.987428733949633,0.0021011995559995582
100,psi,0,6,0.9866978145276131,0.002439241902727726
100,psi,0,7,0.9857565256063348,0.002833164710583308
100,psi,0,8,0.9845567339762301,0.003276398316959516
100,psi,0,9,0.983029983922229,0.0037601220773637724
100,psi,0,10,0.9810807231102961,0.00427559826781999
...
```

なお、当初は 1 ステップごとにシミュレーション結果を保存していたが、実行途中でこの仕様が大幅にディスク容量を圧迫することが確認された。後に行う Java の CPU 実装において、正確性を検証する際に必要なのは、ある程度のステップ間隔を空けた数値結果のみで良い。そのため、それらを加味して出力タイミングは初期状態と、そこから 20 ステップごとに制限するよう変更した。

出力された CSV ファイルは、後続の Java CPU 実装と比較することで、数値的正確性を段階的に検証するための基準データとして機能する。

## 3.3 Java による CPU でのシミュレーション実装

段階的開発手法の第 2 段階として、Processing コードを Java に移植し、CPU 上でシミュレーションを実行する実装を行った。この実装により、Processing と Java の間で数値的な一貫性を確認し、後続の GPU 実装の検証基盤を確立した。

### 3.3.1 実装の概要

Processing 4 で実装されていたシミュレーションコードを、標準的な Java アプリケーションとして再実装した。この時、Processing においては描画までを行っていたが、今回の Java CPU 実装においては数値計算が出来れば良いため、描画処理は実装していない。Processing と同様にシミュレーションが進行できることと、第 1 段階で出力した CSV データとの比較テストが行えることを主な実装要件とした。

### 3.3.2 シミュレーションに関する実装

#### 複素数演算の実装

オーダーパラメータ $\psi$ やリンク変数 $w_x, w_y$ は複素数であるため、複素数演算をサポートする `Complex` クラスを実装した。主要な機能をソースコードとして以下に示す。

ソースコード 3.1: Complex クラスの主要なメソッド

```
public class Complex {
    public double x, y;  // 実部と虚部

    // 複素数の乗算
    public Complex mul(Complex z) {
        return new Complex(
            this.x * z.x - this.y * z.y,
            this.x * z.y + this.y * z.x
        );
    }

    // 複素共役
    public Complex conj() {
        return new Complex(this.x, -this.y);
    }

    // 絶対値の二乗
    public static double abssq(Complex z) {
        return z.x * z.x + z.y * z.y;
    }
}
```

#### シミュレーションループの実装

`GinzburgLandauCpu` クラスでは、Processing コードの `draw()` 関数に相当する `step()` メソッドを実装し、各時間ステップでの物理量の更新を行う。なお先述した通り Java 側での描画は行わないため、以下のように数値計算のみを行う関数としている。

ここで、`updateLinkAFI()` はリンク変数の時間発展を、`updatePsiAFI()` はオーダーパラメータの時間発展をそれぞれ実装している。

ソースコード 3.2: step() メソッドの実装

```
public void step() {
    updateLinkAFI(tau);       // リンク変数の更新
    boundaryPsiNBC();         // 境界条件の適用
```

```
    updatePsiAFI(tau);          // オーダーパラメータの更新
    tcount++;
}
```

### 3.3.3 CSV ファイルと比較するテストの実装

Java CPU 実装の正確性を保証するため、Processing から出力された CSV データとの比較を行うコードを別途実装した。具体的に、`compareWithReference()` メソッドを作成し、以下のように記述した。

ソースコード 3.3: compareWithReference() メソッドの実装

```
public boolean compareWithReference(int step, String referenceFile)
        throws IOException {
    final double EPSILON = 1e-10;
    int mismatchCount = 0;
    int totalLines = 0;

    // psi配列の比較
    for (int i = 0; i <= Nx + 1; i++) {
        for (int j = 0; j <= Ny + 1; j++) {
            // CSVから参照値を読み込み
            double refReal = Double.parseDouble(parts[4]);
            double refImag = Double.parseDouble(parts[5]);
            double genReal = psi[i][j].x;
            double genImag = psi[i][j].y;

            // 許容誤差内で一致するかチェック
            if (Math.abs(refReal - genReal) >= EPSILON ||
                Math.abs(refImag - genImag) >= EPSILON) {
                mismatchCount++;
            }
        }
    }
    // wx, wy配列についても同様に比較
    ...
}
```

このテストでは、オーダーパラメータ $\psi$、リンク変数 $w_x, w_y$ の全ての格子点について、実部と虚部を個別に比較する。許容誤差は浮動小数点の精度を考慮して $1 \times 10^{-10}$ と設定した。

### 3.3.4 Java における CPU 実装の正確性検証

先述したテストを 0、20、100 ステップにおいて実行すると、2 次元シミュレーションにおいては計 69,008 個の値を比較することになる。内訳を以下に示す。

- オーダーパラメータ $\psi$ ($[N_x + 2] \times [N_y + 2] = 152 \times 152 = 23,104$ 個)
- リンク変数 $w_x$ ($[N_x + 1] \times [N_y + 2] = 151 \times 152 = 22,952$ 個)
- リンク変数 $w_y$ ($[N_x + 2] \times [N_y + 1] = 152 \times 151 = 22,952$ 個)

これらの値について、CSV ファイルに記述されている値と許容誤差 $1 \times 10^{-10}$ 以内で一致するかをテストし、全て有効であることを確認した。検証結果の一例として、ステップ 100 での検証出力を以下に示す。

```
Running CPU test mode...
Grid size: 150x150
Time step: 0.001
Test steps: 100

Completed step: 20
Completed step: 40
Completed step: 60
Completed step: 80
Completed step: 100

Comparing with reference file: src_processing/output/step_100.csv

Verification Summary for step 100:
  Total values compared: 69008
  Mismatches: 0
  All values match
```

以上の結果から、Java CPU 実装が Processing 実装と数値的に等価であることが確認できた。この結果により、Java CPU 実装を信頼できる検証基盤として使用し、GPU 実装の正確性を段階的に確認することが可能となった。

## 3.4 CUDA を用いた GPU 実装

段階的開発手法の第 3 段階として、CUDA を用いたシミュレーション実装を行った。Java CPU 実装を基に、計算集約的な部分を CUDA カーネルとして実装し、GPU の並列計算能力を活用することで大幅な高速化を実現した。

### 3.4.1 実装の概要

GPU 実装では、Java から JCuda ライブラリを用いて CUDA を呼び出し、計算処理を GPU 上で実行する。具体的に、`GinzburgLandauGpu.java` では GPU メモリの確保、カーネルのコンパイルと実行、データ転送などの管理を行う。CUDA で実行するカーネルである `superconductor_kernels.cu` と `boundary_kernels.cu` では、それぞれリンク変数とオーダーパラメータの更新処理や、境界条件の処理を行う。

### 3.4.2 CUDA カーネルの実装

本研究で実装した CUDA カーネルは、各格子点の計算を個別のスレッドに割り当てることで並列化を実現している。主要な CUDA カーネルを以下に示す。

#### リンク変数の更新カーネル

リンク変数の更新には、以下のカーネルを実装した。

- `shift_link_x_kernel`, `shift_link_y_kernel`: 角度変数 $\theta_x, \theta_y$ からリンク変数 $w_x, w_y$ を計算
- `update_theta_x_kernel`, `update_theta_y_kernel`: リンク変数の時間発展を計算
- `calculate_thetax_from_wx_kernel`, `calculate_thetay_from_wy_kernel`: リンク変数から角度変数を再計算

これらのカーネルは内部格子点を並列に処理し、各スレッドが対応する格子点の計算を担当する。

#### オーダーパラメータの更新カーネル

オーダーパラメータの更新には、`update_psi_kernel` を実装した。このカーネルでは、チェッカーボードパターンによる並列化を採用している。具体的には、条件 $(i+j+k) \bmod 2 = 1$ を満たす格子点のみを更新することで、隣接格子点間の依存性を回避し、全格子点の約 50%を同時に並列更新することができる。

カーネルの実装例を以下に示す。

ソースコード 3.4: update_psi_kernel の実装

```
extern "C" __global__ void update_psi_kernel(
    double *p_r, double *p_i,
    const double *wx_r, const double *wx_i,
    const double *wy_r, const double *wy_i,
    const double *alpha_arr,
    int N, int M, double h_sq, double dt_step,
    double g_0, double g_1, double beta_val, int k_flag) {
    int i = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int j = blockIdx.y * blockDim.y + threadIdx.y + 1;
    if (i > N || j > M || (i + j + k_flag) % 2 != 1) return;
    // オーダーパラメータの更新処理
    ...
}
```

### 3.4.3 データ転送の最適化

GPU における計算処理は並列で非常に高速であるが、その結果を保存、描画するために CPU で処理しようとすると、GPU デバイスからホスト (CPU) へデータを転送する必要がある。この転送処理は大幅な遅延を生むものであり、Processing でのシミュレーションと同様に 20 ステップごとにデータ転送を行っていると、大幅なシミュレーション速度の低下を招くことが確認された。

実際に 20 ステップごとにデータを転送するようにしてシミュレーションを行ってみると、2 次元シミュレーションの場合では GPU の使用率がおよそ 80%程度となる一方で、500 ステップごとのように十分に間隔を大きくすると 95%程度まで活用できることが確認された。

シミュレーションが十分に高速でなかった Processing における処理では、特にリアルタイムでの可視化において 20 ステップごとに描画するのは合理的であった。しかし、GPU でのシミュレー

ションは高速なため、20 ステップごとに描画するとおよそ 290 フレーム毎秒となり、これは過剰な描画頻度となる。また、先述した通りデータ転送は遅延を生むため、リアルタイム描画の利点を損ねない程度にこの間隔を大きくすることで、より高速なシミュレーションを達成した。

具体的にシミュレーションの進行速度を考慮し、2 次元シミュレーションでは 500 ステップごと、3 次元シミュレーションでは 20 ステップごとに GPU から CPU へデータを転送するように変更した。これにより、可視化に必要なデータは適切な間隔で取得しつつ、転送オーバーヘッドを最小限に抑えることができた。

また、可能な限りデータ転送による遅延を最小化するために、データ転送は非同期転送（cuMemcpyDtoHAsync）を用いて実装し、計算処理と並行して実行できるようにした。実装例を以下に示す。

ソースコード 3.5: 非同期データ転送の実装

```
// GPU_TO_CPU_TRANSFER_INTERVAL(=500)ステップごとにCPUに転送
if (tcount % GPU_TO_CPU_TRANSFER_INTERVAL == 0) {
    transferToCpuAsync();
}

private void transferToCpuAsync() {
    // 非同期転送を実行
    cuMemcpyDtoHAsync(Pointer.to(temp_psi_real),
                      psi_real_d, psiSize, transferStream);
    // ストリームの同期を待つ
    cuStreamSynchronize(transferStream);
    // CPU配列に変換
    DataTransformUtils.unflattenComplex(...);
}
```

### 3.4.4 GPU 実装の正確性検証

GPU 実装の正しさを確認するため、Java CPU 実装の結果と比較検証を実施した。検証方法は、CPU 実装と同様に、特定のステップで全格子点の物理量（オーダーパラメータ $\psi$、リンク変数 $w_x, w_y$）を比較するものである。

検証の結果、GPU 実装と CPU 実装の間で、浮動小数点演算の精度範囲内で一致することを確認した。GPU の浮動小数点演算による微小な差異は存在するが、物理的に意味のある範囲内であり、シミュレーション結果の信頼性に影響を与えないことを確認した。

この検証により、GPU 実装が CPU 実装と数値的に等価であることが保証され、高速化を実現しつつ、シミュレーション結果の正確性を維持できることが確認された。

## 3.5 TCP 通信による描画機構

GPU 実装により高速化されたシミュレーション結果をリアルタイムで可視化するため、Java 側のシミュレーション実行環境と Processing 側の描画環境を TCP/IP 通信で連携させる機構を実装した。この機構により、計算処理と描画処理を分離し、GPU 計算の性能を最大限に維持しつつ、スムーズな可視化を実現している。実際に可視化を行った様子を図 3.1 と図 3.2 に示す。

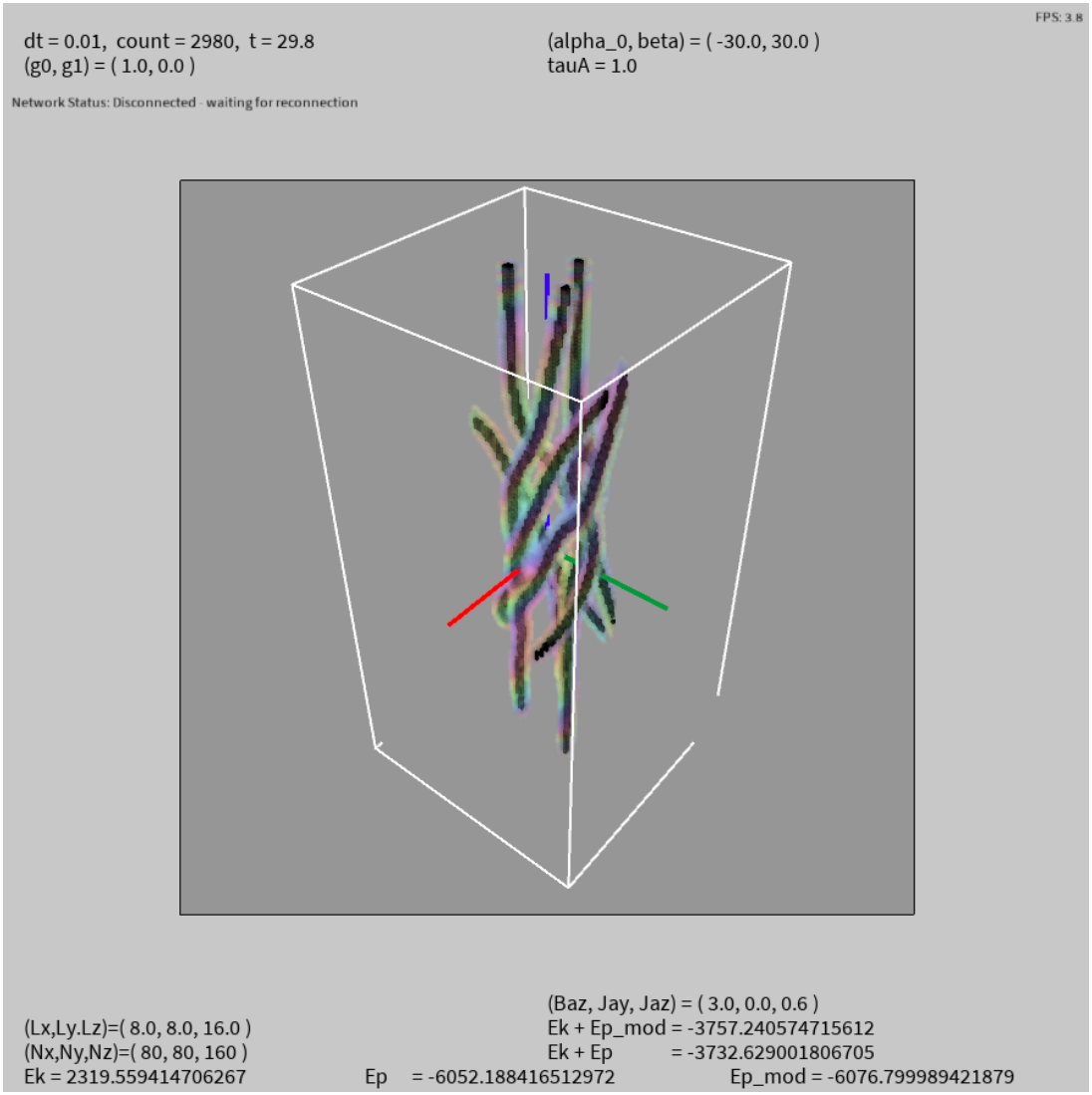


図 3.1: TCP 通信によるリアルタイム可視化（3 次元シミュレーションの描画例）

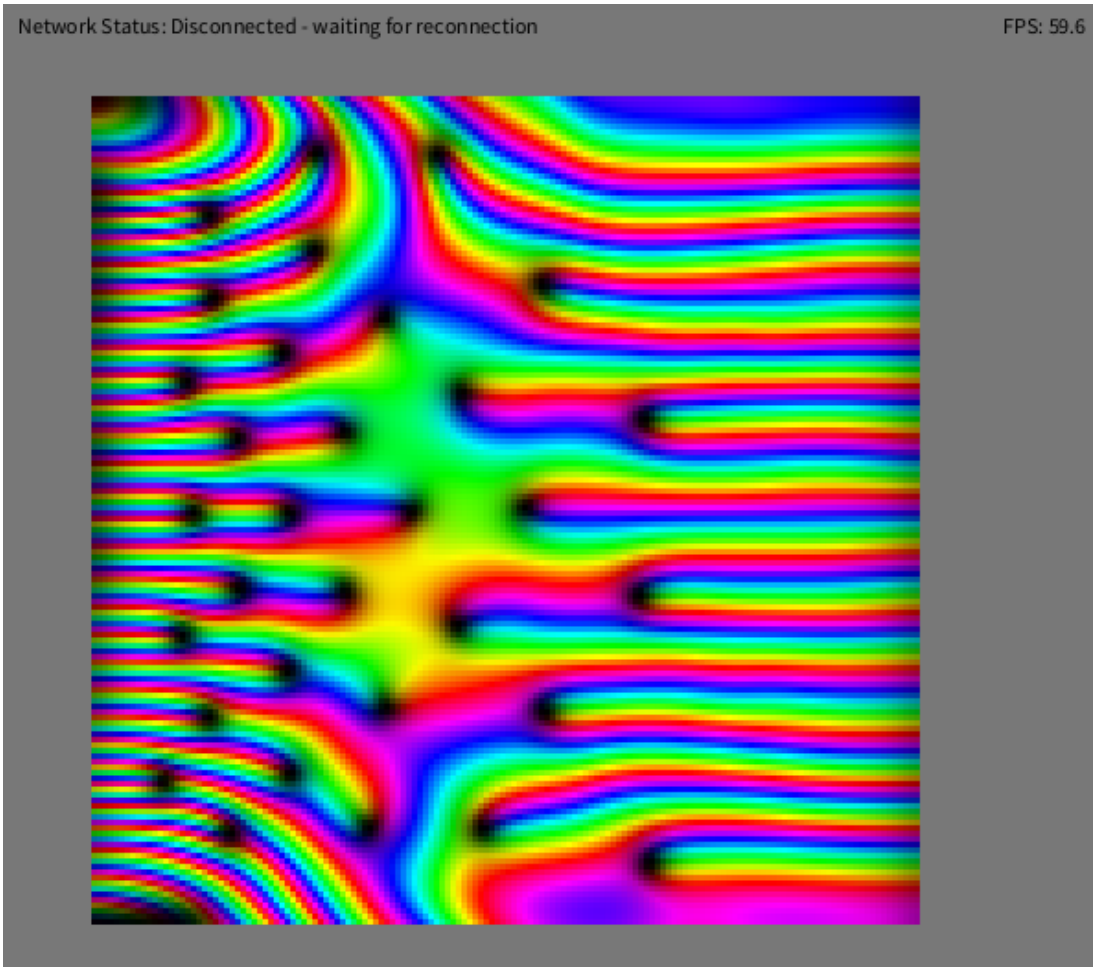


図 3.2: TCP 通信によるリアルタイム可視化（2 次元シミュレーションの描画例）

### 3.5.1 通信機構の概要

本システムでは、Java 側のシミュレーション実行環境（計算層）と Processing 側の描画環境（描画層）を、TCP/IP ソケット通信で接続している。Java 側はクライアントとして動作し、Processing 側はサーバーとして待機する。シミュレーションの各ステップにおいて、描画に必要な物理量（オーダーパラメータ $\psi$、リンク変数 $w_x, w_y$）をバイナリ形式で送信し、Processing 側で受信してリアルタイムに描画する。

### 3.5.2 非同期送信とデータ破棄機構

シミュレーションの計算速度が非常に高速であるため、ネットワーク帯域がボトルネックとなって送信が追いつかない場合がある。この問題に対処するため、送信処理は非同期で実行するようにし、もし遅延が発生してキューが滞るようであれば、最新データのみを保持して古いデータを破棄するようにした。これにより、必ずリアルタイムでシミュレーション結果が描画されることになる。

具体的には、`SimulationDataSender` クラスにおいて、`AtomicReference<DataSnapshot>` を用いて送信待ちデータを管理している。新しいシミュレーションデータが生成された際、`getAndSet()`

メソッドにより古いデータを置き換えることで、常に最新のシミュレーション結果のみが送信されるようになっている。これにより、ネットワーク帯域が不足している場合でも、最新の状態を可視化することが可能となる。

### 3.5.3 ローカル実行環境での動作

本システムは、基本的に同一デバイス上で Java 側のシミュレーションと Processing 側の描画を実行することを想定している。理論上は異なるデバイス間での通信も可能であるが、シミュレーション結果の転送に必要なデータ量が大きいため、一般的なネットワーク環境では帯域が不足する。そのため、同一デバイス上での実行により、ローカルループバックインターフェース（127.0.0.1）を経由した高速な通信を実現している。

### 3.5.4 ソースコード例

Java 側の送信実装の主要部分を以下に示す。

ソースコード 3.6: SimulationDataSender クラスの主要メソッド

```
public class SimulationDataSender {
    // 送信待ちデータを保持（最新のもののみ）
    private final AtomicReference<DataSnapshot> pendingData =
        new AtomicReference<>(null);
    private Thread transmissionThread;
    private volatile boolean running = false;

    /**
     * シミュレーションデータを送信キューに追加（非ブロッキング）
     * 既にデータがある場合は古いものを破棄して最新のものに置き換え
     */
    public boolean sendData(int step, Complex[][] psi,
                            Complex[][] wx, Complex[][] wy) {
        if (!shouldSend(step)) {
            return true;
        }

        // 内部格子点のみをコピーしてスナップショットを作成
        DataSnapshot snapshot = new DataSnapshot(step, Nx, Ny,
                                                 psi, wx, wy);
        // 古いデータを破棄して最新のデータに置き換え
        DataSnapshot oldSnapshot = pendingData.getAndSet(snapshot);

        return true;
    }

    /**
     * 送信スレッド：バックグラウンドでデータを送信
     */
    private void transmissionLoop() {
        while (running && connected) {
            try {
                // 送信待ちデータを取得（取得後はnullにリセット）
                DataSnapshot snapshot = pendingData.getAndSet(null);
```

```
                if (snapshot != null) {
                    // TCPでデータを送信
                    if (transmitData(snapshot)) {
                        lastSentStep = snapshot.step;
                    }
                    // 送信頻度を制限（最大20回/秒）
                    Thread.sleep(50);
                } else {
                    Thread.sleep(1);
                }
            } catch (InterruptedException e) {
                break;
            }
        }
    }
}
```

Processing 側の受信実装の主要部分を以下に示す。

ソースコード 3.7: NetworkReceiver.pde の主要関数

```
/**
 * ネットワークからデータを受信して処理
 */
void checkForNetworkData() {
  if (!networkClientConnected) {
    // クライアント接続を待機
    Client newClient = networkServer.available();
    if (newClient != null) {
      networkClient = newClient;
      networkClientConnected = true;
    }
    return;
  }

  // 利用可能なデータを読み込み
  if (networkClient != null && networkClient.available() > 0) {
    receiveData();
  }
}

/**
 * 完全なメッセージを受信したら解析して配列を更新
 */
void parseCompleteMessage() {
  // メタデータを解析
  double time = bytesToDouble(receiveBuffer, 0);
  // シミュレーションパラメータを解析
  double r_h = bytesToDouble(receiveBuffer, 40);
  double r_tau = bytesToDouble(receiveBuffer, 48);
  // ... その他のパラメータ

  // フィールドデータを解析して配列を更新
  int offset = METADATA_SIZE;
  for (int i = 1; i <= Nx; i++) {
    for (int j = 1; j <= Ny; j++) {
      // 各格子点のpsi, wx, wyを読み込み
```

```
      double psi_real = bytesToDouble(receiveBuffer, offset);
      double psi_imag = bytesToDouble(receiveBuffer, offset + 8);
      // ... wx, wyも同様に読み込み

      // 配列を更新
      psi[i][j] = new Complex(psi_real, psi_imag);
      wx[i][j] = new Complex(wx_real, wx_imag);
      wy[i][j] = new Complex(wy_real, wy_imag);

      offset += POINT_SIZE;
    }
  }
}
```

この実装により、GPU による高速なシミュレーション計算と、Processing によるリアルタイム可視化を両立させることができた。非同期送信とデータ破棄機構により、ネットワーク帯域がボトルネックとなる場合でも、常に最新のシミュレーション結果を可視化できるようになっている。

# 第4章　評価と考察

## 4.1　数値的正確性の検証

実装の信頼性を確保するため、段階的な開発手法に基づき、以下の多段階検証を実施した。具体的には、まず既存の Processing のコードと Java での CPU 実装において、精度に差が無いことを確認した。その後、Java での CPU 実装と GPU 実装において、許容される範囲の誤差に収まることを確認した。これにより、間接的に Processing での実装と GPU 実装が許容される範囲内の正確性を有していることを確認した。

### 4.1.1　Processing-Java CPU 間検証

既存の Processing 実装と Java CPU 実装の間で数値的一致を確認した際のログを以下に示す。

**2 次元シミュレーションにおける検証**

```
Running CPU test mode...
Grid size: 150x150
Time step: 0.001
Test steps: 100

Completed step: 20
Completed step: 40
Completed step: 60
Completed step: 80
Completed step: 100

Comparing with reference file: src_processing/output/step_100.csv

Verification Summary for step 100:
  Total values compared: 69008
  Mismatches: 0
  All values match
```

**3 次元シミュレーションにおける検証**

```
[INFO] Running jp.ac.kyutech.cse.aquarius20.PinSimulationTest
=== Pinning Simulation Accuracy Test ===
Checking step 1... Pass Rate: 100.00%, Max Error: 1.14e-09 at (1,80,3)
```

```
Checking step 2... Pass Rate: 100.00%, Max Error: 1.87e-09 at (1,39,80)
Checking step 3... Pass Rate: 100.00%, Max Error: 3.13e-09 at (1,39,80)
Checking step 4... Pass Rate: 100.00%, Max Error: 4.57e-09 at (1,39,80)
Checking step 5... Pass Rate: 100.00%, Max Error: 6.18e-09 at (1,39,80)
Checking step 6... Pass Rate: 100.00%, Max Error: 7.95e-09 at (1,39,80)
Checking step 7... Pass Rate: 100.00%, Max Error: 1.21e-08 at (1,80,53)
Checking step 8... Pass Rate: 100.00%, Max Error: 1.71e-08 at (1,80,53)
Checking step 9... Pass Rate: 100.00%, Max Error: 2.72e-08 at (1,80,53)
Checking step 10... Pass Rate: 100.00%, Max Error: 3.89e-08 at (1,80,53)
=== Test Completed ===
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 16.09 s
-- in jp.ac.kyutech.cse.aquarius20.PinSimulationTest
```

2 次元および 3 次元両者ともに、CPU でのシミュレーション結果と Processing でのシミュレーション結果が一致していることが確認できる。また、シミュレーションが進行するにつれて最大誤差が大きくなっていることが確認できるものの、その値は十分に小さいため、シミュレーションの正確性に影響を与えていないことが確認できる。

### 4.1.2 CPU-GPU 間検証

CPU でのシミュレーション結果と GPU でのシミュレーション結果が一致していることを確認した際のログを以下に示す。

**2 次元シミュレーションにおける検証**

```
Running GPU test mode (CPU vs GPU comparison)...
Grid size: 150x150
Time step: 0.001
Test steps: 100
Compare interval: 20 steps

Initializing GPU resources...
GPU resources initialized successfully.
Checking step 20... Pass Rate: 100.00%, Max Error: 2.78e-15 at (2,127,34)
Checking step 40... Pass Rate: 100.00%, Max Error: 5.33e-15 at (2,126,15)
Checking step 60... Pass Rate: 100.00%, Max Error: 7.44e-15 at (2,126,65)
Checking step 80... Pass Rate: 100.00%, Max Error: 8.99e-15 at (2,126,65)
Checking step 100... Pass Rate: 100.00%, Max Error: 1.07e-14 at (2,125,8)
Cleaning up GPU resources...
GPU resources cleaned up.

All steps passed!
```

**3 次元シミュレーションにおける検証**

```
[INFO] Running jp.ac.kyutech.cse.aquarius20.PinCPUGPUComparisonTest
=== CPU vs GPU Comparison Test (Pinning Simulation) ===
[2026-01-26 05:34:33.705] Initializing GPU resources...
[2026-01-26 05:34:33.705] Using GPU: NVIDIA GeForce RTX 4070 Ti SUPER
Log size: 1
[2026-01-26 05:34:34.220] GPU resources initialized successfully.
Checking step 1... Pass Rate: 100.00%, Max Error: 4.44e-16 at (78,5,2)
Checking step 2... Pass Rate: 100.00%, Max Error: 4.44e-16 at (76,80,0)
Checking step 3... Pass Rate: 100.00%, Max Error: 5.55e-16 at (66,80,26)
Checking step 4... Pass Rate: 100.00%, Max Error: 5.55e-16 at (66,81,26)
Checking step 5... Pass Rate: 100.00%, Max Error: 5.55e-16 at (72,72,0)
Checking step 6... Pass Rate: 100.00%, Max Error: 5.55e-16 at (4,8,23)
Checking step 7... Pass Rate: 100.00%, Max Error: 5.55e-16 at (73,81,26)
Checking step 8... Pass Rate: 100.00%, Max Error: 5.55e-16 at (75,68,0)
Checking step 9... Pass Rate: 100.00%, Max Error: 5.55e-16 at (0,79,2)
Checking step 10... Pass Rate: 100.00%, Max Error: 5.55e-16 at (62,66,0)
Checking step 11... Pass Rate: 100.00%, Max Error: 6.11e-16 at (1,80,19)
Checking step 12... Pass Rate: 100.00%, Max Error: 7.22e-16 at (1,80,19)
Checking step 13... Pass Rate: 100.00%, Max Error: 9.71e-16 at (1,80,19)
Checking step 14... Pass Rate: 100.00%, Max Error: 9.99e-16 at (1,80,19)
Checking step 15... Pass Rate: 100.00%, Max Error: 1.01e-15 at (0,80,19)
Checking step 16... Pass Rate: 100.00%, Max Error: 9.44e-16 at (0,80,19)
Checking step 17... Pass Rate: 100.00%, Max Error: 9.44e-16 at (1,81,78)
Checking step 18... Pass Rate: 100.00%, Max Error: 8.33e-16 at (2,78,9)
Checking step 19... Pass Rate: 100.00%, Max Error: 9.44e-16 at (2,78,9)
Checking step 20... Pass Rate: 100.00%, Max Error: 1.08e-15 at (2,78,9)
=== Test Completed ===
[2026-01-26 05:34:49.963] Cleaning up GPU resources...
[2026-01-26 05:34:49.983] GPU resources cleaned up.
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 16.48 s
-- injp.ac.kyutech.cse.aquarius20.PinCPUGPUComparisonTest
```

2 次元および 3 次元両者ともに、CPU でのシミュレーション結果と GPU でのシミュレーション結果が一致していることが確認できる。

### 4.1.3 シミュレーション精度に関する考察

検証結果を踏まえ、段階的検証手法の有効性と、各実装間の数値精度の関係について考察する。

#### 段階的検証手法の有効性

本研究で採用した段階的検証手法は、得られた成果から有効なものであったと結論づけられる。具体的に、精度を検証していく上で以下のような利点があった。

まず、既存の Processing 実装と Java CPU 実装の間で数値的一致を確認することで、Java CPU 実装が Processing 実装と同じ結果を出力することを確認した。直接 Processing 実装から GPU 実装を比較しないことにより、Processing 基盤から Java へランタイムを移行したことによる原因を切り分けることができる。加えて、Processing で出力する CSV ファイルは容量が大きいため、数百ステップ先の結果を保持するのが非効率という問題もあった。先に Java CPU 実装を行ってその正確性を検証することで、これらの問題を解決することができた。

さらに、後述する通り本研究では生成 AI によるプログラミングを積極的に行っている。AI を活用したプログラミングを行うにあたって、このような段階的検証手法は非常に有効かつ、正確性を大きく向上させるものであった。

### 各実装間の数値精度の比較

検証結果から、各実装間の数値精度について以下のことが明らかになった。

2 次元シミュレーションにおける Processing 実装から Java CPU 実装への移植では、CSV 参照値との比較において、許容誤差 $1 \times 10^{-10}$ 以内で全て一致することが確認された。これは両実装が倍精度浮動小数点数を使用しており、同一のアルゴリズムを実装していることから期待される結果である。3 次元シミュレーションにおける同様の検証では、最大誤差が $10^{-8}$ オーダーで推移したものの、いずれも許容範囲内であり、実装の正確性に影響を与えないことを確認した。

Java CPU 実装から GPU 実装への移行では、2 次元シミュレーションにおいて最大誤差が $10^{-15}$ から $10^{-14}$ のオーダーで推移することが確認された。具体的には、ステップ 20 で $2.78 \times 10^{-15}$、ステップ 100 で $1.07 \times 10^{-14}$ 程度の誤差が観測された。3 次元シミュレーションでは、ステップ 1 で $4.44 \times 10^{-16}$、ステップ 20 で $1.08 \times 10^{-15}$ 程度の誤差となり、2 次元よりも小さい誤差に留まった。

これらの誤差はいずれも倍精度浮動小数点数の機械イプシロン（約 $2.2 \times 10^{-16}$）に近い値であり、数値計算における理論的な限界に近い精度が達成されていることを示している。ステップ数の増加に伴い誤差が累積する傾向が見られるが、これは浮動小数点演算の丸め誤差が各ステップで蓄積される通常の挙動であると推察される。

### 実用性と精度のバランス

超伝導シミュレーションにおいて、$10^{-14}$ オーダーの数値誤差は物理的な解釈に影響を与えない。オーダーパラメータ $\psi$ の絶対値は 0 から 1 の範囲で変化し、物理的に意味のある変化は $10^{-3}$ 程度のスケールで生じる。したがって、観測された $10^{-14}$ の誤差は有効数字の 11 桁以上が一致していることを意味し、シミュレーション結果の信頼性は十分に担保されている。

本研究では、GPU 実装の許容誤差として $10^{-6}$ を設定した。これは GPU 浮動小数点演算の特性を考慮した保守的な閾値であったが、実際の誤差はこれを 8 桁も下回る結果となった。この結果は、CUDA による GPU 実装においても、適切なアルゴリズム設計と同期処理を行うことで、CPU 実装と同等の数値精度を維持できることを実証している。

### 今後の精度向上の方向性

前節の検証ログでは、CPU 実装と GPU 実装の差は小さく留まる一方、ステップ数の増加に伴って最大誤差が徐々に増大する傾向が見られた。これは、各時間ステップの更新に含まれる浮動小数点演算の丸め誤差が累積することに加え、CPU と GPU で演算の順序や最適化の適用の違いが

ある場合、その微小差が長時間の反復で拡大し得るためである。長時間シミュレーションにおいてこの誤差拡大を抑えるためには様々な手法が考えられるが、最も有効と考えられるのは演算の順序を固定し、決定性を向上させることにある。

GPU 側で並列化に起因する演算順序の揺らぎが生じると、加算の順序で結果が微小に変化することで、その差は増幅しやすくなる。そのため累積和が存在する場合は演算順序を固定し、必要に応じて最適化オプションを調整して CPU/GPU 間で丸め規則の差を小さくすることが有効だと考えられる。

## 4.2 パフォーマンスの評価と考察

### 4.2.1 2 次元シミュレーションにおける性能比較

2 次元シミュレーションにおいて、既存の Processing 実装と本研究で開発した GPU 実装の性能を比較した。Processing 実装の性能測定では、30 秒間シミュレーションを実行し、その間に進行したステップ数から性能を算出した。その結果、約 59.0 steps/sec の性能が得られた。一方、GPU 実装では 100,000 ステップのシミュレーションを実行し、その平均性能を測定した。以下に GPU 実装の実行ログを示す。

```
Grid size: 150x150
Time step: 0.001
Max steps: 100000
Visualization: localhost:5555
Send interval: every 20 steps

Initializing GPU resources...
GPU resources initialized successfully.
Connecting to Processing visualization server at localhost:5555...
Connected to Processing visualization server (buffer size: 1024 KB)
Starting GPU simulation...
Step: 1000 / 100000 (5154.6 steps/sec, Recent Avg: 5154.6, Total Avg: 5154.6)
Step: 2000 / 100000 (5464.5 steps/sec, Recent Avg: 5309.6, Total Avg: 5309.6)
Step: 3000 / 100000 (5780.3 steps/sec, Recent Avg: 5466.5, Total Avg: 5466.5)
Step: 4000 / 100000 (5747.1 steps/sec, Recent Avg: 5536.6, Total Avg: 5536.6)
Step: 5000 / 100000 (5917.2 steps/sec, Recent Avg: 5612.8, Total Avg: 5612.8)
...
Step: 95000 / 100000 (5848.0 steps/sec, Recent Avg: 5875.7, Total Avg: 5807.0)
Step: 96000 / 100000 (5917.2 steps/sec, Recent Avg: 5896.4, Total Avg: 5808.1)
Step: 97000 / 100000 (5882.4 steps/sec, Recent Avg: 5889.4, Total Avg: 5808.9)
Step: 98000 / 100000 (6024.1 steps/sec, Recent Avg: 5917.7, Total Avg: 5811.1)
Step: 99000 / 100000 (5882.4 steps/sec, Recent Avg: 5910.8, Total Avg: 5811.8)
Step: 100000 / 100000 (5848.0 steps/sec, Recent Avg: 5910.8, Total Avg: 5812.2)
Simulation completed!
Disconnected from Processing visualization server
Cleaning up GPU resources...
```

```
GPU resources cleaned up.
```

GPU 実装では、最終的な 1 秒あたりの平均生成ステップ数が約 5,812 steps/sec となった。これは Processing 実装と比較して、約 98.5 倍の高速化を達成したことを示している。

この大幅な性能向上により、従来は 10 分以上を要していたシミュレーションが数秒で完了するようになり、パラメータを変化させながらの試行錯誤的な研究が実現可能となった。

### 4.2.2 3 次元シミュレーションにおける性能比較

3 次元シミュレーションにおいても、CPU 実装と GPU 実装の性能比較を行った。3 次元では計算量が大幅に増加するため、GPU による並列化の効果がより顕著に現れることが期待される。

CPU 実装の性能測定では、120 フレームのシミュレーションを実行し、完了までに 4 分 16 秒（256 秒）を要した。これはフレームレートに換算すると約 0.469 FPS に相当する。一方、GPU 実装では 3,000 ステップのシミュレーションを実行し、その平均性能を測定した。以下に GPU 実装の実行ログを示す。

```
=== NAFI 3D TDGL Simulation ===
Total steps: 3000
Using GPU
GPU batch size: 20 (boundary conditions processed on GPU)
Network: localhost:5555
================================
Initializing GPU...
[2026-02-02 07:28:46.516] Initializing GPU resources...
[2026-02-02 07:28:46.516] Using GPU: NVIDIA GeForce RTX 4070 Ti SUPER
[2026-02-02 07:28:46.936] GPU resources initialized successfully.
[2026-02-02 07:28:47.013] [NETWORK_CONNECT] Attempting to connect to localhost:5555...
[2026-02-02 07:28:47.038] [NETWORK_CONNECT_SUCCESS] Connected to localhost:5555 (64MB buffer
Sending initial state (step 0)...
Step: 100 / 3000 (85.4 steps/sec, Recent Avg: 85.4, Total Avg: 85.4)
Step: 200 / 3000 (88.9 steps/sec, Recent Avg: 87.1, Total Avg: 87.1)
Step: 300 / 3000 (89.8 steps/sec, Recent Avg: 88.0, Total Avg: 88.0)
Step: 400 / 3000 (89.4 steps/sec, Recent Avg: 88.4, Total Avg: 88.4)
Step: 500 / 3000 (90.0 steps/sec, Recent Avg: 88.7, Total Avg: 88.7)
...
Step: 2500 / 3000 (89.6 steps/sec, Recent Avg: 89.7, Total Avg: 89.5)
Step: 2600 / 3000 (90.2 steps/sec, Recent Avg: 89.8, Total Avg: 89.6)
Step: 2700 / 3000 (89.8 steps/sec, Recent Avg: 89.8, Total Avg: 89.6)
Step: 2800 / 3000 (89.0 steps/sec, Recent Avg: 89.6, Total Avg: 89.6)
Step: 2900 / 3000 (89.6 steps/sec, Recent Avg: 89.7, Total Avg: 89.6)
Step: 3000 / 3000 (89.5 steps/sec, Recent Avg: 89.6, Total Avg: 89.6)
[2026-02-02 07:29:20.997] [NETWORK_DISCONNECT] Closing connection...
[2026-02-02 07:29:20.997] [NETWORK_DISCONNECT] Connection closed
================================
```

```
Simulation completed!
Total time: 33.63 seconds
Average time per step: 11.21 ms
[2026-02-02 07:29:20.998] Cleaning up GPU resources...
[2026-02-02 07:29:21.014] GPU resources cleaned up.
```

GPU実装では、最終的な平均フレームレートが約89.6FPSとなった。これはCPU実装の約0.469FPSと比較して、約191倍の高速化を達成したことを示している。

2次元シミュレーションでの約98.5倍と比較して、3次元ではさらに大きな高速化率が得られた。これは、3次元への拡張により計算量が増加したことで、GPUの並列計算能力がより効果的に活用されたためと考えられる。また、89.6 FPSという性能は、リアルタイムでの可視化を伴う対話的なシミュレーションを十分に実現できる水準である。

なお、ここでは簡略化のためにFPSと表現しているが、実際には1秒あたりの生成ステップ数を示したものである。先述した通り全てのステップが描画されるわけではなく、効率的な可視化のために一定間隔で描画されるため、描画フレームレートはこれよりも低くなる。これを加味したとしても、対話的なシミュレーションを実現できることに変わりはない。

## 4.3 AIを活用した開発プロセスに関する考察

本研究では、アルゴリズムの移植支援やエラー解析において生成AIを活用した。AIは実装速度を向上させる強力なツールであるが、誤ったコードを生成することがある。本研究では当初、精度の検証機構を設けずにGPU実装を行うように指示したが、2ステップの時点で全く違う値を出力するような、実用性に乏しいものであった。この問題を解消するため、第3章で述べた「CPU版による検証基盤」がAI生成コードの品質を保証する上で不可欠であった。

また、可視化処理をJavaで実装する試みも行われたが、数値を一致させるタスクと違い視覚情報が必要な可視化実装の移植は、コードのみを頼りに実装するAIにとっては困難なタスクであった。そこで本研究ではProcessingに対してネットワーク経由で情報を送るというアプローチを取ることによりこれを解決した。

これらの経験から、AI時代のソフトウェア開発においても、人間の設計による堅牢な検証システムや、開発を促進する仕組みを策定することの重要性が再確認された。

# 第5章　結論

## 5.1　本研究のまとめ

本研究では、超伝導体内の磁束線運動を記述する時間依存 Ginzburg-Landau（TDGL）方程式の数値シミュレーションを対象として、CUDA を用いた GPGPU 計算による高速化を行った。TDGL シミュレーションは格子点ごとに同種の計算を繰り返す構造を持つ一方で、計算規模の増加に伴い CPU 逐次計算では実行時間がボトルネックとなり、十分な時間発展やパラメータ探索の実施が困難であった。本研究はこの問題に対し、GPU の並列計算能力を活用することで計算時間を抜本的に短縮し、研究プロセスの試行錯誤性と可視化の即時性を高めることを目的とした。

開発にあたっては、既存の Processing 実装を起点に、Processing の CSV 出力を参照値として Java CPU 実装の数値的一致を確認し、Java CPU 実装と GPU 実装を直接比較することで GPU 実装の正確性を保証する、という段階的検証手法を採用した。この手法により、実装の移植に伴う原因の切り分けが容易となり、また大量の参照データを扱う上での運用上の非効率も低減された。検証結果として、Processing と Java CPU 実装の一致、ならびに CPU と GPU の一致が確認され、CPU-GPU 間の差は大きいものでも $10^{-10}$ という極めて小さな値に留まることを示した。

性能評価では、2 次元シミュレーションにおいて従来の Processing 実装に対し約 98.5 倍、3 次元シミュレーションにおいて CPU 実装に対し約 191 倍の高速化を達成した。特に 3 次元では計算量の増大に伴い GPU 並列化の効果がより顕著となり、リアルタイム可視化を伴う対話的なシミュレーションを実現できる性能水準に到達した。さらに、可視化については GPU 計算結果をネットワーク経由で Processing へ送信する構成を採ることで、計算部分と描画部分を分離しつつ、既存の可視化コードを活用することができるシステムを構築した。

また、本研究では生成 AI を開発の補助として利用したが、AI が生成するコードは誤りを含み得ること、そしてそのリスクを実用上許容可能な範囲に抑えるためには、設計段階で検証基盤を組み込むことが不可欠であることが明らかになった。すなわち、AI の活用は実装速度の向上に寄与する一方で、数値計算のように「正しさ」が本質となる領域では、段階的検証や決定的な比較手順といった人間の設計による品質保証が成果の信頼性を支える。

## 5.2　今後の課題

本研究で構築した GPU シミュレーションは、精度と性能の両面で有効性を示した一方、さらなる発展の余地がある。

第一に、長時間シミュレーションにおける誤差の累積や再現性の観点から、ステップ数が増加することに伴う誤差の拡大を防ぐ施策が挙げられる。シミュレーションとして活用する現実的な範囲でのステップ数ではこれらの誤差による問題は確認されなかったが、長時間シミュレーションにおいてはこれらの誤差による問題が顕著になる可能性がある。第二に、他のシミュレーションに対しても本研究で行った GPGPU による加速を行うことが挙げられる。これにより、多くの物理シミュレーションが高速かつ対話的に実行できるようになり、研究の効率化が期待できる。

以上より、本研究は超伝導体内の磁束線運動を TDGL 方程式でシミュレーションするコードに対し、CUDA による高速化と、その正確性を保証する検証基盤、および実用的な可視化連携を一体として実現した。これにより、従来は計算時間の制約により困難であった試行錯誤的な研究を現実的な時間スケールで実施可能とし、超伝導体内の磁束線運動の理解と解析の効率化に寄与する基盤を提供した。

# 謝辞

本論文を作成するにあたり、熱心にご指導いただいた小田部荘司教授、ならびに研究室の皆様に深く感謝の意を表します。

小田部教授には、超伝導に関する基礎的な知識から、本研究における Ginzburg-Landau 方程式に関する理論的な背景、ならびに検証の進め方について、数多くのご助言をいただきました。また、日々の研究生活において、議論や技術的な相談に乗っていただいた研究室の皆様に心より感謝いたします。充実した研究環境のなかで本論文を執筆することができました。

本研究を遂行するにあたり、以上の方々をはじめ、多くの方に支えていただいたことに、改めて感謝の意を表します。

# 参考文献


[1] K. Arita et al., Physica C **662** (2024) 1354522.

[2] T. Matsuno, E.S. Otabe, Y. Mawatari, Journal of the Physical Society of Japan **89** (2020) 054006.

[3] JCuda, jcuda.org - JCuda, `http://www.jcuda.org/jcuda/JCuda.html` (accessed Feb. 10, 2026).

# 研究実績

- 杉本陸 他、「CUDA を用いた超伝導体動力学の並列高速シミュレーション」、2025 年度電子情報通信学会九州支部学生会講演会、福岡工業大学、令和 7 年 9 月 17 日