CudaWaveField ライブラリ
リファレンスマニュアル

関西大学
電気電子情報工学科
光情報システム研究室

寺口　功

2012 年 3 月 22 日

CudaWaveField ライブラリ　Rel 1.0.0
リファレンスマニュアル　Rel 1.0

- CudaWaveField ライブラリ (以下 cwfl)，およびそのマニュアルの著作権は寺口 功，松島恭治および関西大学システム理工学部光情報システム研究室が保有しています．
- 寺口，松島および関西大学はこれらのライブラリ/プログラムの使用によって生じたいかなる損害についても，それを補償する責を負うものではありません．
- 商用目的でない研究・教育に自由に利用することができます．cwfl ライブラリを用いて行った研究成果を発表する場合は，謝辞等に cwfl を利用した旨を記載していただけるとありがたく思いますが，これは強制ではありません．
- バグ報告や質問等は下記の配布サイトで受け付けています．バグ報告の際には，問題が生じたソースを示していただけますようお願いします．
  http://www.laser.ee.kansai-u.ac.jp/WaveFieldTools
- 現時点では再配布等はご遠慮下さい．

Note

- このマニュアルはハイパーテキストになっています．このマニュアルを効率良く参照するためには Acrobat Reader のツールボタンの「前の画面」と「次の画面」を表示することを強く奨めます．
- これらのツールボタンは，デフォルトではツールバーに表示されないので，「ツール」⇒「ツールバーのカスタマイズ」から設定を行ってください．

目次

第 I 部

CudaWaveField ライブラリ

第 1 章

開発環境

1.1 CudaWaveField ライブラリを用いる開発環境

1.1.1 必須環境

このライブラリは NVIDIA 社の GPU を用いて WaveField の計算支援を行うものである．そのため，以下の環境が必須である．

Windows x64 環境

このライブラリは現在 64bit 環境のみを対象としている．32bit の Windows では動作しない．

NVIDIA 社の GPU

Compute Capability 2.0 以上の GPU．1.3 以下の GPU では動作しない．

CUDA

NVIDIA 社の GPU 開発環境として，このライブラリが対応したバージョンの CUDA が必要である．指定以外のバージョンの CUDA では動作しない．本ライブラリが対応する CUDA のバージョンはダウンロードファイル中あるいはダウンロードサイトに明記してある．

なお，CUDA 開発環境として，下記のインストールが必要である．

- CUDA Toolkit
- GPU Computing SDK

WaveFieldLib3

このライブラリは WaveFieldLib3(以下，WFL3) を拡張するものである．そのため，WFL3 がインストールされている必要がある．

1.1.2 インストール

インストーラを起動するだけでインストールが完了するが，WFL3 および本ライブラリが対応したバージョンの CUDA がインストールされている必要がある．WFL のサンプルソースと CUDA のサンプル

ソースのいずれもが正常にビルドできるようにインストールされていなければならない.

プログラム作成には次節に示す開発環境のいずれかが必要である.

1.1.3 開発環境とコンパイラ

WaveField ライブラリでは以下の開発環境とコンパイラで用いることができる.

`MS Visual Studio .NET (Visual Studio 2003)`

未検証

`MS Visual Studio 2005`

未検証.

`MS Visual Studio 2008`

推奨開発環境．IntelliSense が正常に機能する．ただし，64 ビット CPU 用コードのコンパイルが可能でなければならない (64 ビットコンパイラを使用するためには Visual Studio のインストール時に指定が必要．デフォルトではインストールされない).

`Intel C++ Compiler`

未検証.

1.1.4 ディレクトリ構造とファイル

WaveField ライブラリのインストーラは次のディレクトリに必要なファイルをインストールする．なおデフォルトでは < インストールディレクトリ >=`c:\WaveFieldTools` である.

ヘッダ

< インストールディレクトリ >`\include\`

< インストールディレクトリ >`\include\cwfl\`

ファイル：`*.h`

インポートライブラリ

WaveField ライブラリを呼び出すプログラムのビルド時には以下のフォルダにあるインポートライブラリが必要である.

< インストールディレクトリ >`\lib\`

ファイル：`cwfl.lib`

ダイナミックリンクライブラリ

WaveField ライブラリを呼び出すプログラムの実行時には以下のフォルダにあるダイナミックリンクライブラリが必要である．これは `PATH` の通ったフォルダーか実行ファイル (`.exe` ファイル) と同じフォルダーに置かれている必要がある.

< インストールディレクトリ >`\bin\`

ファイル：`cwfl.dll`

第 2 章

CudaWaveFieldLib チュートリアル

2.1 はじめに

CudaWaveField ライブラリは WaveField ライブラリを GPU を用いて支援するライブラリである．そのため，ベースには WaveField ライブラリが存在し，WaveField ライブラリが無いと使用する事はできない．また，ユーザビリティの観点から，本ライブラリのメインとなるクラスである CudaWaveField クラスと WaveField クラスは同じような関数を持ち，引数等もあまり違わないように考慮しているため，WaveField に慣れ親しんだ人には特に説明を読まなくても使える程使いやすく出来ている．この章では，そのコードの書き方の基本を説明する．

2.2 コーディングの基本

CudaWaveField ライブラリを用いたプログラムのソースコードの骨組みを次に示す．

Example 一般的なソースコードのスケルトン

```
#include    <cwfl.h>
using namespace wfl;
using namespace cwfl;

int main(void)
{
    StartCWFL();

    /*----- 以下に必要なプログラムを記述する -----*/

    return 0;
}
```

ヘッダファイル

ヘッダファイルとして cwfl.h をインクルードする．CudaWaveField ライブラリのヘッダファイルは多数のヘッダファイルによって構成されるが，cwfl.h をインクルードすることにより，関連するヘッダファイルがすべて読み込まれる．

また，cwfl.h は内部で wfl3.h をインクルードしているため，wfl3.h をインクルードする必要はない．

名前空間

CudaWaveField ライブラリの関数やクラスは全て cwfl 名前空間内で定義されている．従って，名前空間 cwfl の利用を宣言するのが一般的である．

初期化関数

CudaWaveField ライブラリで定義された関数やクラスを利用する場合は，まず初めに StartCWFL() 関数を実行する．

Note

- StartCWFL() 関数は内部で wfl::Start() を呼んでいるため，StartCWFL() 関数の前後に Start() を入れる必要は無い．
- ビルド時にはライブラリファイル cwfl.lib をリンクするが，これを明示的に設定する必要はない．(もちろん wfl3.lib も)
- 実行時には，PATH の通ったディレクトリか，実行ファイル (*.EXE) と同じディレクトリに cwfl.dll が必要である．(もちろん wfl3.dll も)

2.3　矩形関数の FFT

以下は，幅が縦横とも 1[cm] であるような矩形関数のフーリエ変換を求める例である．

Example 単純な FFT の例 (ソース: ExRectApertureFFT.cpp)

```
#include <cwfl.h>
using namespace std;
using namespace wfl;
using namespace cwfl;

int main(void)
{
    //CWFL開始
    StartCWFL("./FFT/FFT_test.log",1,true,0);

    //時間測定開始
    EventTimer timer;
    timer.Start();

    //512 x 512点，サンプリング間隔1[mm]の複素振幅分布をGPU上に生成
    CudaWaveField cwf(512 , 512 , 1e-3);

    //1[cm] x 1[cm]の矩形関数を設定
    cwf.SetRect(0.01,0.01);

    //フーリエ変換
    cwf.Fft(-1);

    //正規化処理
    cwf.Normalize();

    //測定終了
    timer.Stop();
    Printf("GPU Elapsed time = %f[ms]\n",timer.GetTime());

    //GPU上のデータをCPU側へ転送
    WaveField wf;
    cwf >> wf;

    //bmpとしてセーブ
    wf.SaveAsBmp("./FFT/[GPU]RectFFT_AMPLITUDE.bmp",AMPLITUDE,COLOR);
```

```
    return 0;
}
```

この例で用いている WaveField の関数については，WaveField ライブラリのマニュアルを参照して頂きたい．この例では初期化を行った後，初めに**コンストラクタ**で CudaWaveField クラスのオブジェクトを生成し，SetRect() メンバー関数でサンプリング領域の中心に 1cm × 1cm の矩形関数を設定している．その後，Fft() メンバー関数でフーリエ変換を行い，Normalize() メンバー関数で正規化処理を行っている．最終的に**転送演算子**で wfl::WaveField オブジェクトに転送し，セーブを行っている．この結果を示す．左が GPU により計算した例，右が CPU により計算した例である．どちらも相違が無いため，GPU を用いた場合でも正しく計算できている事がわかる．

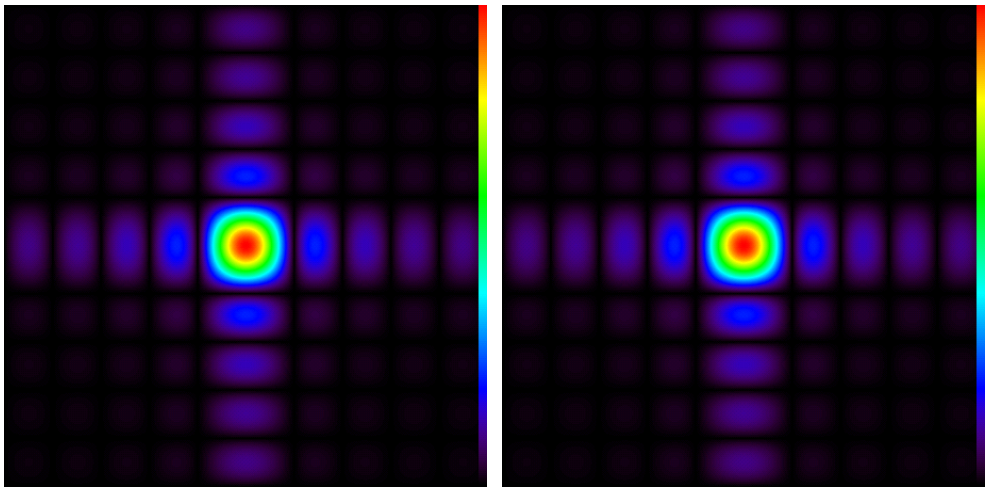

図 2.1　矩形関数のフーリエ変換

2.4　角スペクトル法による円形開口からの伝搬計算

2.4.1　短い距離の伝搬

CudaWaveField クラスを用いて角スペクトル法で円形開口からの伝搬計算を行った例を以下に示す．

Example 短距離の回折伝搬計算 (ソース: ExShortProp.cpp)

```
#include <cwfl.h>

using namespace std;
using namespace wfl;
using namespace cwfl;

int main(void)
{
    //CWFLの初期化
    StartCWFL("./AsmProp/AsmProp.log",1,true,0);

    //オブジェクト生成
```

```
    CudaWaveField cwf(256,256,2e-6);

    //高次のガウシアンビームを設定
    cwf.SetGaussian(0.1e-3,50);

    //角スペクトル法により1mmの伝搬計算
    cwf.AsmProp(1e-3);

    //正規化
    cwf.Normalize();

    //転送処理
    WaveField saver;
    cwf >> saver;

    //画像としてセーブ
    saver.SaveAsBmp("./AsmProp/Diffraction.bmp",AMPLITUDE);
    return 0;
}
```

この例では SetGaussian() メンバー関数を用いて高次ガウシアンビームとして円形開口を設定している．その後，AsmProp() メンバー関数を用いて伝搬計算を行っている．開口画像と伝搬後の画像を図 2.2 に示す．

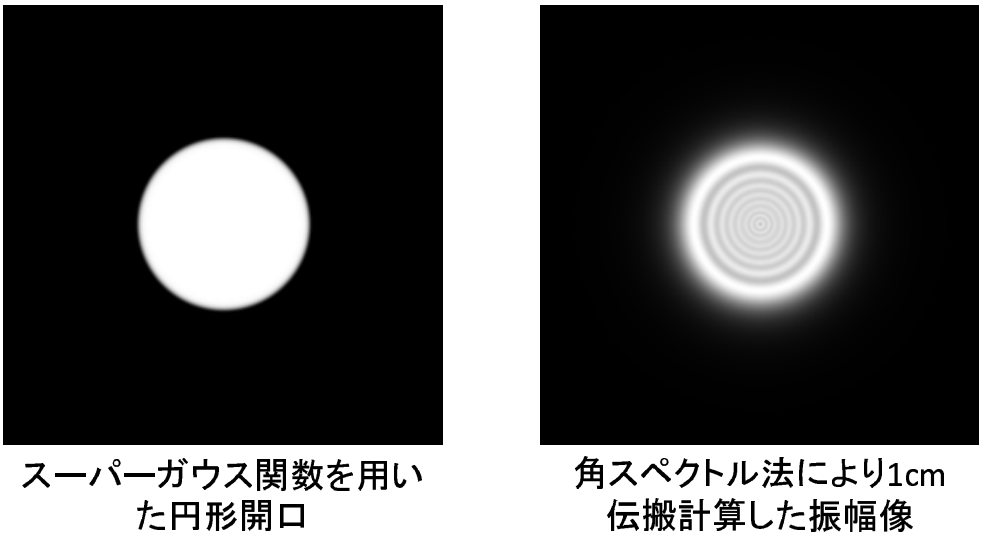

図 2.2　スーパーガウス関数を用いた円形開口と角スペクトル法による伝搬計算

2.4.2　長い距離の伝搬

先ほどの例では 1mm と短い伝搬であったが，今度は 10mm 〜 30mm と伝搬距離を長くした場合のサンプル及び結果を示す．

Example 長い距離の回折伝搬計算 (ソース: ExLongProp.cpp)

```
#include <cwfl.h>

using namespace std;
using namespace wfl;
using namespace cwfl;
```

```
int main(void)
{
    StartCWFL("./AsmProp/AsmProp2.log",1,true,0);
    CudaWaveField aperture(256,256,2e-6);
    aperture.SetGaussian(0.1e-3,50);

    for(int i = 1; i <= 3; i++)
    {
        CudaWaveField dest(aperture);   //WaveFieldオブジェクトで初期化
        double d = 10e-3 * i;
        dest.AsmProp(d);
        char filename[128];
        sprintf(filename,"Diffraction-%d.bmp",(int)(d / 1e-3));
        dest.Normalize();

        WaveField saver;
        dest >> saver;
        saver.SaveAsBmp(filename,AMPLITUDE);
    }
    return 0;
}
```

この場合の結果を図 2.3 に示す．このように長い伝搬計算を通常の AsmProp() メンバー関数を用いて計算すると．誤差が生じる．そこで，より正確に計算する事の出来る ExactAsmProp() メンバー関数を用いた例を次のセクションで示す．

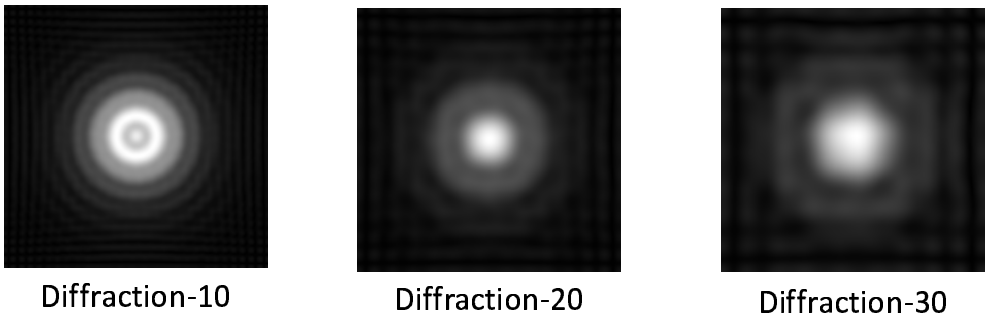

図 2.3 AsmProp() で伝搬距離を増加した場合の回折像．(a)10mm (b)20mm (3)30mm である．

2.4.3 ExactAsmProp を用いた伝搬計算

先ほどの例において，AsmProp() メンバー関数を ExactAsmProp() メンバー関数に置き換えた場合の結果を図 2.4 に示す．このように ExactAsmProp() メンバー関数を用いれば，非常に正確な伝搬計算を行う事が可能であるが，通常に比べて 4 倍のメモリを消費するため，注意が必要である．

2.5 レンズによる結像シミュレーション

応用としてレンズによる結像を CudaWaveField を用いて簡易的にシミュレーションした例を以下に示す．

Example レンズ結像シミュレーション (ソース: ExImageFormationByLens.cpp)

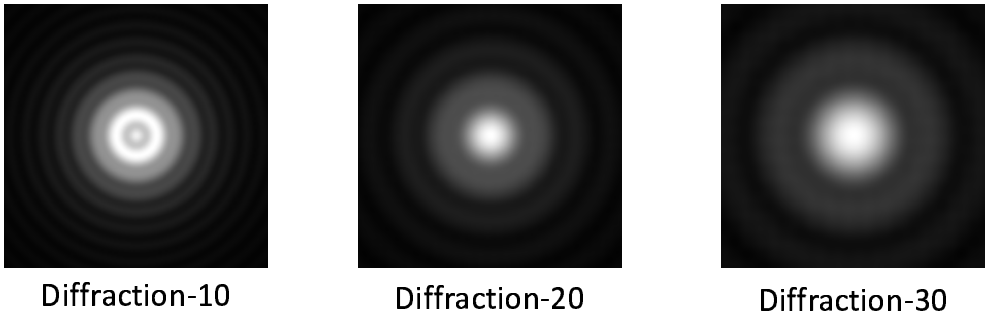

図 2.4 ExactAsmProp() を用いて計算した例

```
#include <cwfl.h>

using namespace std;
using namespace wfl;
using namespace cwfl;

int main(void)
{
    //////////////初期化///////////
    StartCWFL();                              //引数無しでスタート
    FieldParam::SetDefault(128,128,2e-6);     //生成されるフィールドのデフォルト値を設定
    double R = 1e-3;                          //レンズの半径
    WaveField saver;                          //ファイルセーブ用のWFオブジェクト

    /////////元画像の準備/////////
    WaveField temp;                           //画像ロード用のWFオブジェクト
    //↓bmpを振幅像として読み込み
    temp.LoadBmp("./EyeSimulation/lena128x128.bmp",AMPLITUDE);
    CudaWaveField source = temp;              //コピーコンストラクタで、データを渡す
    source.Embed(3);                          //縦横それぞれ8倍の64倍拡張を行う

    //////////レンズ設定//////////
    CudaWaveField lens;                       //レンズの役割をするCWFオブジェクト
    lens.Embed(3);                            //64倍拡張
    lens.SetGaussian(R,100);                  //開口をスーパーガウシアンとして生成
    lens.SetQuadraticPhase(10e-3);            //焦点距離10mmのレンズとして位相設定
    lens >> saver;                            //CWF -> WFに転送
    saver.SaveAsWf("./EyeSimulation/lens.wf");//WFデータとしてセーブ

    ////////////元位置 -> レンズ位置伝搬////////
    source.AsmProp(20e-3);                    //角スペクトル法で伝搬(20mm)

    ////////////レンズ乗算//////////////////
    lens *= source;                           //レンズにソースを乗算
    source.Dispose();                         //不要になったsourceを明示的に破棄

    //////////レンズ位置 -> スクリーン位置伝搬///////
    lens.AsmProp(20e-3);                      //レンズからスクリーンまで伝搬(20mm)

    //////////後処理////////////////
    CudaWaveField& screen = lens;             //わかりやすいように別名をつけた
    screen.Extract(3);                        //64倍縮小(元のサイズに戻す)
    screen.Normalize();                       //正規化


    //////////保存//////////
    screen >> saver;                          //最終結果をGPUからCPU側に転送
    saver.SaveAsBmp("./EyeSimulation/image.bmp",AMPLITUDE);//振幅画像として保存

    return 0;
}
```

ここでは，`FieldParam::SetDefault()` 関数を用いて，フィールドのデフォルトパラメータを設定している．この関数は `CudaWaveField` と `WaveField` に共通のデフォルト値を設定する．このプログラムの実行して得られる結果を図 2.5 に示す．

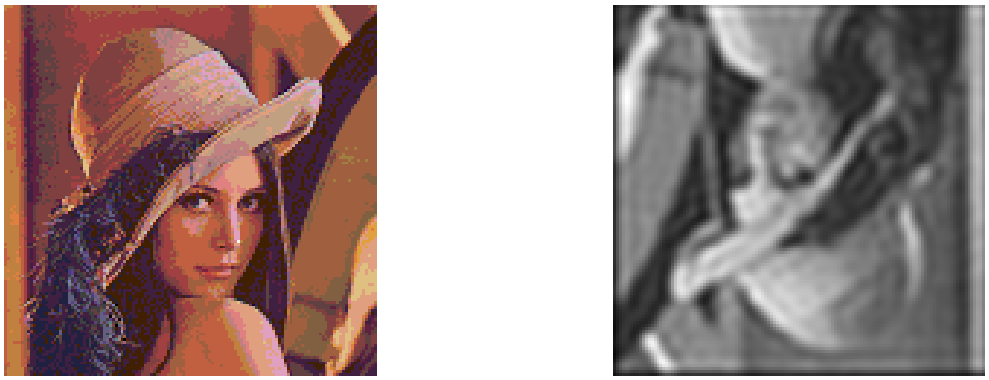

図 2.5　レンズによる結像のシミュレーション

第 3 章

リファレンス

3.1 名前空間内でグローバルな定義

3.1.1 初期化関数

WaveField の諸メソッド・関数を用いる前にまず実行しなければならない初期化関数，また長時間かかるシミュレーションにおいて，計算時間を計測したり，途中経過のメッセージをファイルに記録するための関数群．使用方法は下記の使用例を参照すること．

```
void StartCWFL(iconst char* fname = NULL,int msgLevel = 1,bool deviceProperty
= true,int deviceNumber = 0)
```

戻り値 なし

説　明 CudaWaveField の初期化のために必ず初めに実行する関数．`fname` はログを出力するファイル名，`msgLevel` は出力するメッセージのレベル，`deviceProperty` は GPU のデバイスプロパティを出力するか否か，`deviceNumber` は使用するデバイス (GPU) の ID である．

Note

- この関数より先に他の CudaWaveFieldLib 関数を用いた場合の結果は保証されない．
- メッセージレベルは 0 ～ 3 であり，0 では初期化メッセージは出力されない．
- ログファイル名を指定しない場合，ログ記録関数では，コンソールにのみ出力を表示するが，ログファイル名を指定した場合，コンソールに表示した内容と同じ内容がログファイルに追記される．

3.1.2 CUDA に関する関数

```
int GetCUDADeviceCount(void)
```

戻り値 存在する GPU の数

説　明 PC に搭載されている GPU(CUDA デバイス) の数を返す．

Note

- あくまで PC 内の CUDA デバイスの物理的な数であり，その全てが CWFL で使用可能とは限らない．

```
cudaDeviceProp GetCUDADeviceInformation(int deviceID)
```

戻り値　cudaDeviceProp 構造体

説　明　deviceID で指定したデバイスのプロパティ (モデルネーム，メモリ容量，ComputeCapability 等) をメンバに持つ cudaDeviceProp 構造体を返す．プロパティの詳細な内容は CUDA4.0 のリファレンスマニュアルを参照して下さい．

```
void CudaMallocHost(wfl::Complex* plPointer,long nx,long ny)
```

戻り値　なし

説　明　nx * ny * sizeof(Complex) サイズの CUDA から扱うことの出来る CPU 側メモリを確保し，plPointer にその領域を指すアドレスを入れる．CudaWaveField クラスの Init() メンバー関数と組み合わせて使う場合が多い．

Example

```
//ホスト側(CPU側)のメモリ領域を指し示すポインタを宣言
float2 *host_memory;

//GPU側から扱うことのできるホスト側メモリ領域を確保
CudaMallocHost(static_cast<wfl::Complex*>(host_memory),1024,1024);

//CWFオブジェクトを生成
CudaWaveField cwf(1024,1024);

//CWFオブジェクトが内部で指し示すデータ本体へのポインタを変更する
cwf.Init(host_memory);

//このようにする事でホスト側メモリをCWFオブジェクトに持たせる事が出来る．
```

3.2 CudaWaveField クラス

3.2.1 コンストラクタ・デストラクタ

```
CudaWaveField(void)
CudaWaveField(long nxy)
CudaWaveField(long nx, long ny)
CudaWaveField(long nx, long ny, double pxy)
CudaWaveField(long nx, long ny, double px, double py)
CudaWaveField(long nx,long ny,double px,double py, double waveLength)
CudaWaveField(const wfl::WaveField& wf)
CudaWaveField(const wfl::FieldParam& param)
```

戻り値 なし

説　明 コンストラクタ．1 番目の形式では全てデフォルト値のオブジェクトを生成する．2 番目の形式では x 方向サンプリング点数と y 方向サンプリング点数は同じ nxy となる．3 番目の形式では x 方向サンプリング点数 nx，y 方向サンプリング点数を nx とし，x，y 方向サンプリングが同じ間隔 p [単位:m] のオブジェクトを生成する．4 番目の形式では x 方向と y 方向サンプリング間隔を px と py で個別に設定する．5 番目の形式ではさらに，波長を wavelength [単位:m] に設定する．6 番目の形式は wfl::WaveField の全パラメータとデータをコピーするコンストラクタであり，暗黙的に CPU 側から GPU 側への転送処理が行われる．7 番目の形式は wfl::FieldParam によってパラメータを設定する．データは設定されない．

Note

- 初期状態では省略された引数のデフォルト値 (既定値) は次のとおりである．
  サンプリング点数　$N_x = N_y = 256$
  サンプリング間隔　$P_x = P_y = 1\mu\mathrm{m}$
  波長　$\lambda = 632.8\mathrm{nm}$
- これらのデフォルト値は wfl::FieldParam::SetDefault***を用いて変更できる．
- サンプリング点数は必ず，2 の累乗でなければならない．2 の累乗でないサンプリング点数を指定した場合は，指定したサンプリング点数が格納できる最小の 2 の累乗サイズでオブジェクトが生成される．

Example

```
//すべてデフォルト値のオブジェクトを宣言する
CudaWaveField cwf1;

//サンプリング点数を512x512としたオブジェクトを宣言する
CudaWaveField cwf2(512);

//1024x1024でサンプリング間隔がx，y方向で同じ10umのオブジェクトを宣言する
CudaWaveField cwf3(1024, 1024, 10e-6);
```

```
//2048x2048でサンプリング間隔がx方向5um，y方向10umのオブジェクトを宣言する
CudaWaveField cwf3(2048, 2048, 5e-6, 10e-6);

//既存のWaveFieldオブジェクトを用いてCWFオブジェクトを作る
wfl::WaveField wf1(1024,1024);
CudaWaveField cwf4(wf1);

//wfl::FieldParamオブジェクトを用いてCWFオブジェクトを作る
wfl::FieldParam param(4096,4096);
CudaWaveField cwf5(param);
```

```
CudaWaveField(const CudaWaveField& temp)
```

戻り値　なし

説　明　コピーコンストラクタ．データもコピーされる．

```
~CudaWaveField(void)
```

戻り値　なし

説　明　データが残っており，なおかつそれが外部メモリでない場合，データ領域を破棄する．そのため，一般に GPU 上のメモリ容量を節約したい場合を除いて明示的に破棄を行う必要はない．

3.2.2　初期化等の基本関数

```
void Init(float2* d = NULL)
```

戻り値　なし

説　明　CudaWaveField オブジェクトの持つデータ本体を収めるメモリ領域を GPU 上に確保する．既存の領域は消去される．d にポインタを与えた場合，そのポインタが指し示す領域をデータ格納先とする．

```
Dispose(void)
```

戻り値　なし

説　明　オブジェクトがデータ本体を収めた GPU 上のメモリ領域を持っている場合，その領域を明示的に解放する．データ本体を収めた領域が外部である場合 (Init() メンバー関数の引数に NULL 以外を与えた場合)，解放は行われない．

```
void Clear(void)
```

戻り値　なし

説　明　領域の 0 クリアを行う．実行にはコストがかかる．

```
void ThreadsOptimization(void)
void ThreadsOptimization(long nx,long ny)
```

戻り値 なし

説　明 スレッド数の最適化を行う．一般に，CudaWaveField は生成された時や，メンバ関数の実行時にそれぞれスレッド数・ブロック数の最適化を行っているため，一般に使う必要は無い．何らかの理由でスレッド数・ブロック数を明示的に調整したい場合にのみ使用すること．

1 番目の形式は引数をとらない．そのため，CudaWaveField オブジェクトが持っている nx,ny に合わせて最適化される．

2 番目の形式は nx,ny に合わせて最適化される．

3.2.3 コピー関数

```
void CopyParam(const wfl::FieldParam& param)
void CopyParam(const CudaWaveField& param)
void CopyParam(const wfl::WaveField& param)
```

戻り値 なし

説　明 それぞれの形式で，引数に指定したオブジェクトから nx,ny を除いたパラメータのみをコピーする．

```
void CopyParamAll(const wfl::FieldParam& param)
void CopyParamAll(const CudaWaveField& param)
void CopyParamAll(const wfl::WaveField& param)
```

戻り値 なし

説　明 それぞれの形式で，引数に指定したオブジェクトから nx,ny を含めて全てのパラメータをコピーする．

```
void CopyWinAreaData(CudaWaveField& dest)
```

戻り値 なし

説　明 dest にこのオブジェクトの Window エリア内のデータのみをコピーする．dest とこのオブジェクトの Window エリアのサイズが一致しない場合，エラーとなる．

3.2.4 領域の拡大・縮小に関わる関数

```
CudaWaveField& Embed(int nxy)
CudaWaveField& Embed(int nx,int ny)
```

戻り値　自身への参照

説　明　領域を拡大する．1 番目の形式では縦横をそれぞれ 2 の nxy 乗倍する．2 番目の形式では横方向に 2 の nx 倍，縦方向に 2 の ny 倍する．元のサンプル点の周辺は 0 パディングされる．

```
CudaWaveField& Extract(int nxy)
CudaWaveField& Extract(int nx,int ny)
```

戻り値　自身への参照

説　明　領域を縮小する．1 番目の形式では縦横のサンプリング点数をそれぞれ 2 の 1 / nxy 倍する．2 番目の形式では横方向に 2 の 1 / nx，縦方向に 2 の 1 / ny 倍する．

```
void AdjustNumPixel(void)
```

戻り値　なし

説　明　現在の領域を含む最小の 2 乗数倍の大きさになるようにサンプリング点数を変更する．

3.2.5 役割を設定する関数

```
CudaWaveField& SetRect(double wx,double wy,float amp = 1.0)
```

戻り値　自身への参照

説　明　幅 wx，高さ wy，振幅 amp の矩形をサンプリング領域の中央に設定する．

```
CudaWaveField& SetGaussian(double r,double nn = 2.0,double amp = 1.0)
```

戻り値　自身への参照

説　明　サンプリング領域の中心に半値幅 r，次数 nn，振幅 amp のガウス関数を設定する．

```
CudaWaveField& SetPlaneWave(wfl::Vector v,wfl::Phase phs = 0.0)
```

戻り値　自身への参照

説　明　方向ベクトル v，初期位相 phs の平面波を設定する．

```
CudaWaveField& SetSphericalWave(wfl::Point sphWCenter, double amp = 1.0)
```

戻り値 自身への参照

説　明 球面波の中心 sphWCenter，振幅 amp である球面波がこのオブジェクトの座標位置 (CudaWaveField::GetOrigin() にて取得) における波面を設定する．

3.2.6 光波計算関数

```
CudaWaveField& MultiplyPlaneWave(Vector v,wfl::Phase phs = 0.0)
CudaWaveField& MultiplyPlaneWave(double CosA,double CosB, wfl::Phase phs =
0.0)
```

戻り値 自身への参照

説　明 1 番目の形式は方向ベクトル v，初期位相 phs の平面波をこのオブジェクトに乗算する．
2 番目の形式は x 軸に対する方向余弦 CosA，y 軸に対する方向余弦 CosB，初期位相 phs の平面波をこのオブジェクトに乗算する．

```
CudaWaveField& AddSphericalWave(wfl::Point p)
CudaWaveField& AddSphericalWave(wfl::Point p, wfl::Phase phs,double a)
```

戻り値 なし

説　明 1 番目の形式は初期位相 0，振幅 1 で光源位置 p からの光波を加算する．
2 番目の形式は初期位相 phs，振幅 a で光源位置 p の点光源からの光波を加算する．

3.2.7 位相に関する関数

```
CudaWaveField& SetRandomPhase(void)
```

戻り値 自身への参照

説　明 ランダム位相を設定する．

```
CudaWaveField& SetQuadraticPhase(double f)
```

戻り値 自身への参照

説　明 焦点距離 f のレンズに相当する 2 次位相を設定する．振幅は変化しない．

3.2.8 ゲッター・セッター

```
float2* GetDataPointer(void) const
```

戻り値　float2 型のポインタ

説　明　データ本体を格納している領域へのポインタを返す．一般に CudaWaveField オブジェクトはデータ領域を GPU 上に持っているため，返されたポインタを CPU 側と同様に使う事はできない．

```
size_t GetDataSize(void) const
```

戻り値　size_t

説　明　size_t 型でデータ領域の大きさを返す．単位は [byte] である．

```
void SetParam(wfl::FieldParam& param)
```

戻り値　なし

説　明　wfl::FieldParam を引数にとり、そのパラメータをこのオブジェクトにセットする．

```
void SetBlocks(int x,int y,int z)
```

戻り値　なし

説　明　ブロック数を設定する．通常は自動的に最適な値が設定されているので使用する必要は無い．

```
dim3 GetBlocks(void)
```

戻り値　dim3 型

説　明　CUDA の組み込み型である dim3 型で，現在設定されてるブロック数を返却する．

```
void SetThreads(int x,int y,int z)
```

戻り値　なし

説　明　スレッド数を設定する．通常は自動的に最適な値が設定されているので使用する必要は無い．

```
dim3 GetThreads(void)
```

戻り値　dim3 型

説　明　CUDA の組み込み型である dim3 型で，現在設定されてるスレッド数を返却する．

3.2.9 FFT に関する関数

```
void FakeFft(void)
```

戻り値 なし

説　明 パラメータだけをフーリエ変換したものと同等にする．データはフーリエ変換されない

```
void Fft(int s, bool beforeShift = true,bool afterShift = true)
void Fft(int s,CudaFFTPlan cuPlan,bool beforeShift = true,bool afterShift =
true)
```

戻り値 なし

説　明 1 番目の形式はフーリエ変換の方向 s(-1 ならばフーリエ変換，1 ならばフーリエ逆変換), beforeShift,afterShift はそれぞれフーリエ変換を行う前と後にシフト処理 (象限交換) を行うか否かである．2 番目の形式もほぼ同じであるが，CudaFFTPlan クラスを使用する．FFT を頻繁に行う場合に CudaFFTPlan クラスを使いまわす事で通常のものより高速化が図れる場合がある．

```
void SwitchFs_(void)
```

戻り値 なし

説　明 象限交換を行う．

```
CudaWaveField& RawScaledFft(double s,double t)
```

戻り値 自身への参照

説　明 ScaledFFT によりオブジェクトを離散フーリエ変換する．

3.2.10 回転変換

```
CudaWaveField& Rotate(const CudaWaveField& source, wfl::RMatrix& crmat,
wfl::SFrequency* c)
```

戻り値 自身への参照

説　明 source オブジェクトを回転変換し，対象オブジェクトに代入する．wfl::RMatrix 型の crmat が回転行列である．キャリア周波数がゼロになるように，回転変換後は変換前の光波のキャリヤ信号成分が取り除かれている．取り除いたキャリア信号成分の周波数は，wfl::SFrequency 型のポインタ c が NULL でない場合に *c に設定される

```
CudaWaveField& RotateFs(wfl::Rmatrix& crmat,CudaWaveField& cpfb,double2
offset)
```

戻り値　自身への参照

説　明　フーリエ空間上で回転変換を行う．回転変換については上記参照

3.2.11　伝搬計算

```
void AsmProp(double d)
```

戻り値　なし

説　明　角スペクトル法による距離 d[m] の伝搬計算を行う．帯域制限を行っている

```
void AsmPropFs(double d)
```

戻り値　なし

説　明　フーリエ空間において角スペクトル法による距離 d[m] の伝搬計算を行う．帯域制限を行っている

```
void RawAsmProp(double d)
```

戻り値　なし

説　明　角スペクトル法による距離 d[m] の伝搬計算を行う．帯域制限を行っていない

```
void RawAsmPropFs(double d)
```

戻り値　なし

説　明　フーリエ空間において角スペクトル法による距離 d[m] の伝搬計算を行う．帯域制限を行っていない

```
CudaWaveField& ExactAsmProp(double d)
```

戻り値　自身への参照

説　明　角スペクトル法で距離 d[m] の伝搬計算を行う．内部で 4 倍拡張をおこなっている．

```
CudaWaveField& ShiftedAsmProp(const CudaWaveField& source,int precision = 1)
```

戻り値　自身への参照

説　明　source から自身にシフト角スペクトル法により伝搬計算を行う．この際，source と自身の座標がズレていても構わない．

```
CudaWaveField& FourierProp(double f)
```

戻り値 自身への参照

説　明 距離 f のフーリエ回折伝搬計算を行う.

```
CudaWaveField& BackFourierProp(double f)
```

戻り値 自身への参照

説　明 距離 f の逆フーリエ回折伝搬計算を行う.

```
CudaWaveField& ShiftedFresnelProp(const CudaWaveField& source)
CudaWaveField& ShiftedFresnelProp(CudaWaveField& source,
CudaShiftedFresnelPropDesc& csfpd)
```

戻り値 自身への参照

説　明 1 番目の形式は source から自身へシフテッドフレネル法で伝搬計算を行う．2 番目の形式も同様であるが，CudaShiftedFresnelPropDesc クラスを用いる事で，何度もこの関数を呼ぶ場合に高速化を図る事が出来る.

```
CudaWaveField& ShiftedFresnelPropAdd(CudaWaveField& source,
CudaShiftedFresnelPropDesc& csfpd)
```

戻り値 自身への参照

説　明 source オブジェクトをシフテッドフレネル法で伝搬計算を行い，このオブジェクトに加算する.

```
CudaWaveField& ShiftedFresnelPropEx(CudaWaveField& source)
```

戻り値 自身への参照

説　明 source オブジェクトをシフテッドフレネル法を用いて回折伝搬計算し，このオブジェクトに代入する．サンプリング数可変型.

```
CudaWaveField& ShiftedFresnelPropAddEx(CudaWaveField& source)
```

戻り値 自身への参照

説　明 source オブジェクトをシフレッドフレネル法を用いて回折伝搬計算し，このオブジェクトに加算する．サンプリング数可変型.

3.2.12 描画

```
void PaintTriangle(const wfl::PointArray& p, float2 amp,bool memset)
```

戻り値　なし

説　明　p で表される三角形を振幅 amp で描画する．memset は描画前に領域をゼロクリアするか否かのフラグである．

```
PaintPolygonShape(const wfl::PointArray& p ,float2 amp)
```

戻り値　なし

説　明　p で表されるポリゴンを振幅 amp で描画する．

3.2.13 変換・リダクション

```
double MaxReduction(void)
```

戻り値　double 型で表される最大値

説　明　自身の持つ要素の中から，振幅の最大値を探し出して返却する．

```
double2 MaxMinRedution(void)
```

戻り値　double2 型で表される最大・最小値

説　明　自身の持つ要素の中から，振幅の最大値および最小値を探し出して返却する．double2 型は CUDA の組み込み型である．double.x に最大値，double.y に最小値が入る．

```
void Normalize(double amp = 1.0)
```

戻り値　なし

説　明　要素を振幅 amp で規格化する．

3.2.14 特殊演算

```
void AddFrom(CudaWaveField& target)
```

戻り値　なし

説　明　target と自身のローカル座標を考慮し，重なり合っている部分のみを自身に加算する．

3.2.15 演算子オーバーロード

```
CudaWaveField& operator=(const CudaWaveField& cwf)
CudaWaveField& operator+(const CudaWaveField& cwf)
CudaWaveField& operator+=(const CudaWaveField& cwf)
CudaWaveField& operator-(const CudaWaveField& cwf)
CudaWaveField& operator-=(const CudaWaveField& cwf)
CudaWaveField& operator∗(const CudaWaveField& cwf)
CudaWaveField& operator∗=(const CudaWaveField& cwf)
CudaWaveField& operator/(const CudaWaveField& cwf)
CudaWaveField& operator/=(const CudaWaveField& cwf)
```

戻り値 自身への参照

説　明 CWF 同士の四則演算及びコピー代入.

```
CudaWaveField& operator∗= (double val)
CudaWaveField& opeartor∗= (const wfl::ComplexDouble& val)
CudaWaveField& operator/= (double val)
CudaWaveField& operator/= (const wfl::ComplexDouble& val)
CudaWaveField& operator+= (const wfl::ComplexDouble& val)
CudaWaveField& operator-=(const wfl::ComplexDouble& val)
```

戻り値 自身への参照

説　明 double 型を引数に取る演算子は，自身の全要素の Real と Imaginary に val を乗算・除算する．wfl::ComplexDouble 型を引数に取る演算子は自身の全要素と複素数的に四則演算を行う.

```
void operator>> (wfl::WaveField& target)
CudaWaveField& operator<< (const wfl::WaveField& wf)
```

戻り値 なし・自身への参照

説　明 1 番目の形式は CudaWaveField オブジェクトから WaveField オブジェクトへデータを転送する際に使用する．全てのデータが転送される．2 番目の形式は WaveField オブジェクトから CudaWaveField オブジェクトにデータを転送する際に使用する．一般に CudaWaveField オブジェクトはデータのセーブ・ロード等の関数を持たないため，CudaWaveField オブジェクトが持つデータをセーブしたい場合や HDD から読み込んだデータを CudaWaveField オブジェクトにロードしたい場合に用いる.

Example

```
CudaWaveField cwf1;
```

```
WaveField temp;

//cwf1のデータをセーブしたい場合，以下のように用いる
cwf1 >> temp;
temp.SaveAsWf("cwf1.wf");

//ロードを行う場合は次のように使用する(ex.画像のロード)
temp.LoadBmp("lena.bmp",AMPLITUDE);
cwf1 << temp;
```

Note

- 一般に WaveField オブジェクトは CPU 側のメモリ領域，CudaWaveField オブジェクトは GPU 側のメモリ領域を使用しているため，転送の際には PCI-E バスを通る必要がある．PCI-E バスはメモリ内部の転送速度に比べて大幅に遅いため，オーバーヘッドが生じる．効率を上げるためには，無闇に転送を行わない (デバッグ時を除く) 事が肝要である．