Pustaka GPU-Cuda Untuk Poisson Solver 3D Dalam Koordinat Silinder

LIPI

Rifki Sadikin
I Wayan Aditya Swardiana
Taufiq Wirahman
Arnida Lailatul Latifah

Kelompok Penelitian Komputasi Berkinerja Tinggi
Pusat Penelitian Informatika
Lembaga Ilmu Pengetahuan Indonesia

November 2018

Daftar Isi

Bab 1

PoissonSolver3DCylindricalGPU

PoissonSolver3D adalah pustaka yang dikembangkan untuk menyelesaikan persamaan Poisson 3 dimensi dalam sistem koordinat silinder. Persamaan Poisson secara umum berbentuk $\nabla^2(r, \phi, z) = \rho(r, \phi, z)$ dengan diketahui nilai tepi pada potensial $\mathbf{V}$ dan distribusi buatan ρ.

Untuk menyelesaikan persamaan tersebut digunakan metode multigrid yang diimplementasikan pada akselerator GPU.

Pengembang

Pustaka ini dikembangkan oleh Kelompok Penelitian Komputasi Kinerja Tinggi, Pusat Penelitian Informatika, Lembaga Ilmu Pengetahuan Indonesia. Tim pengembang terdiri dari:

- Rifki Sadikin (rifki.sadikin@lipi.go.id), koordinator
- I Wayan Aditya Swardiana (i.wayan.aditya.swardiana@lipi.go.id), anggota
- Taufiq Wirahman (taufiq.wirahman@lipi.go.id), anggota
- Arnida L. Latifah (arnida.lailatul.lattifah@lipigo.id), anggota

Kontak

Untuk pertanyaan, komentar atau diskusi dapat menghubungi tim pengembang di atas atau mengunjungi situs kami di: http://grid.lipi.go.id

Dokumentasi

Petunjuk penggunaan dapat dilihat pada berkas PETUNJUKPENGGUNAAN.md

Bab 2

Petunjuk Penggunaan

```
pembuat : Rifki Sadikin (rifki.sadikin@lipi.go.id)
tanggal : 6 November 2018
```

Petunjuk penggunaan ini berisi:

- Cara **membuat** dan **menginstal** Pustaka Pemercepat Komputasi berbasis GPU CUDA untuk Penyelesaian Persamaan *Poisson* dalam Koordinat Silindrikal.
- Cara **memakai** Pustaka Pemercepat Komputasi berbasis GPU CUDA untuk Penyelesaian Persamaan *Poisson* dalam Koordinat Silindrikal dalam C++.

Struktur Berkas

Struktur berkas pustaka Pemercepat Komputasi berbasis GPU CUDA untuk Penyelesaian Persamaan *Poisson* dalam Koordinat Silindrikal adalah sebagai berikut:

```
|-- CMakeLists.txt
|-- docs
|   `-- Doxyfile.in
|-- example
|   |-- CMakeLists.txt
|   |-- PoissonSolver3DGPUTest.cpp
|   `-- PoissonSolver3DGPUTest.h
|-- interface
|   |-- PoissonSolver3DCylindricalGPU.cxx
|   `-- PoissonSolver3DCylindricalGPU.h
|-- kernel
|   |-- PoissonSolver3DGPU.cu
|   `-- PoissonSolver3DGPU.h
|-- PETUNJUKPENGGUNAAN.md
`-- README.md
```

Modul Program yang Dibutuhkan

Untuk menginstall pustaka dibutuhkan program pengembang di lingkungan sistem operasi Linux (saat ini masih di Linux) yaitu:

1. CMAKE versi minimal 3.10.3
2. GCC versi minimal 6.2.0
3. CUDA versi minimal 8.0
4. GIT versi minimal 1.8.3.1

Jika disediakan module program pada lingkungan sistem HPC maka dapat menjalankan perintah berikut ini:

```
module load cmake/3.10.3
module load gcc/6.2.0
module load cuda
```

Instalasi

Untuk menginstall pustaka PoissonSolver3D lakukan langkah-langkah berikut:

1. Jalankan git clone untuk menyalin kode sumber ke direktori local. Perintah ini akan menyalin kode sumber ke direktori **PoissonSolver3D**

```
$ git clone http://github.com/rsadikin/PoissonSolver3D.git
```

2. Pindah ke direktori PoissonSolver3D, dan lihat daftar direktori

```
$ cd PoissonSolver3D
$ tree .
.
|-- CMakeLists.txt
|-- docs
|   `-- Doxyfile.in
|-- example
|   |-- CMakeLists.txt
|   |-- PoissonSolver3DGPUTest.cpp
|   `-- PoissonSolver3DGPUTest.h
|-- interface
|   |-- PoissonSolver3DCylindricalGPU.cxx
|   `-- PoissonSolver3DCylindricalGPU.h
|-- kernel
|   |-- PoissonSolver3DGPU.cu
|   `-- PoissonSolver3DGPU.h
|-- PETUNJUKPENGGUNAAN.md
`-- README.md
```

3. Buatlah folder **buildpoissonsolver** di luar direktori kode sumber

```
$ mkdir $HOME/buildpoissonsolver
```

4. Pindah ke folder **buildpoissonsolver**

```
$ cd $HOME/buildpoissonsolver
```

1. Jalankan **cmake** di folder **buildpoissonsolver** dengan menyatakan di mana pustaka akan diinstal dengan perintah sebagai berikut

```
$ export  PSLIB=/home/usertest/trypoissonsolver/PoissonSolver3D
$ export PSLIB_INSTALL=/home/usertest/buildpoissonsolver
$ cmake $PSLIB/ -DCMAKE_INSTALL_PREFIX=$PSLIB_INSTALL
-- The C compiler identification is GNU 6.2.0
-- The CXX compiler identification is GNU 6.2.0
-- Check for working C compiler: /apps/tools/gcc-6.2.0/bin/gcc
-- Check for working C compiler: /apps/tools/gcc-6.2.0/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /apps/tools/gcc-6.2.0/bin/c++
-- Check for working CXX compiler: /apps/tools/gcc-6.2.0/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Found CUDA: /apps/tools/cuda-9.1 (found version "9.1")
-- Building SpaceCharge distortion framework with CUDA support
-- Found Doxygen: /usr/bin/doxygen (found version "1.8.5") found components:  doxygen dot
Doxygen build started
-- Configuring done
-- Generating done
-- Build files have been written to: /home/usertest/buildpoissonsolver
```

2. Kompail kode sumbernya

```
$ make
```

3. Install modul PoissonSolver3D di folder **buildpoissonsolver**

```
$ make install
..
-- Install configuration: ""
-- Installing: /home/usertest/trypoissonsolver/buildpoissonsolver/lib/libPoissonSolver3DCylindricalGPU.so
-- Installing: /home/usertest/trypoissonsolver/buildpoissonsolver/include/PoissonSolver3DCylindricalGPU.h
-- Installing: /home/usertest/trypoissonsolver/buildpoissonsolver/include/PoissonSolver3DGPU.h
```

Hasil instalasi adalah sebuah pustaka yang dapat digunakan (**shared library**) yaitu libPoissonSolver3DCylindrical←↩ GPU.so pada direktori **lib** dan 2 berkas header yang mengandung definisi kelas /fungsi sehingga pengguna pustaka dapat menggunakannya.

Penggunaan Pustaka

Penggunaan Dalam CMAKE

Dalam folder **example** di struktur direktori sumber terdapat contoh penggunaan pustaka libpoissonsolvergpu.so dengan menggunakan **cmake**. Berikut ini langkah-langkahnya:

1. Buat file **CMakeLists.txt** dengan struktur sebagai berikut untuk menambahkan pustaka cuda (**libcuda.←↩ so**) dan pustaka PoissonSolver3DCylindricalGPU (**libPoissonSolver3DCylindricalGPU.so**) pada proyek cmake:

```
cmake_minimum_required (VERSION 2.8.11)
project (3DPoissonSolverGPUTest)

find_package(CUDA)
if(NOT CUDA_FOUND)
    message( FATAL_ERROR "NVIDIA CUDA package not found" )
else()
    find_library(LIBCUDA_SO_PATH libcuda.so)
    string(FIND ${LIBCUDA_SO_PATH} "-NOTFOUND" LIBCUDA_SO_PATH_NOTFOUND )
endif(NOT CUDA_FOUND)
message( STATUS "Building Poisson Solver  with CUDA support" )

if(LIBCUDA_SO_PATH_NOTFOUND GREATER -1)
  message( FATAL_ERROR "NVIDIA CUDA libcuda.so not found" )
endif(LIBCUDA_SO_PATH_NOTFOUND GREATER -1)


#set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/CMake/cuda" ${CMAKE_MODULE_PATH})
#find_package(CUDA QUIET REQUIRED)
set(PSLIBNAME libPoissonSolver3DCylindricalGPU.so)

find_library(PSLIB ${PSLIBNAME})
string(FIND ${PSLIB} "-NOTFOUND" PSLIB_NOTFOUND )

if(PSLIB_NOTFOUND GREATER -1)
  message( FATAL_ERROR "Poisson Solver Cuda Library libPoissonSolver3DCylindricalGPU.o not found" )
endif(PSLIB_NOTFOUND GREATER -1)
```

Setelah itu, baru tambahkan perintah pada CMakeLists.txt (dilanjutkan) untuk mengikut sertakan kode sumber user yaitu:

```
# tambah disini kode sumber user
set(CPP_SOURCE PoissonSolver3DGPUTest.cpp)
set(HEADERS PoissonSolver3DGPUTest.h)

set(TARGET_NAME poissonsolvergputest)

add_executable(${TARGET_NAME}
  PoissonSolver3DGPUTest.cpp
)


# ikut sertakan shared library cuda dan poisson solver
target_link_libraries(${TARGET_NAME} ${PSLIB} ${LIBCUDA_SO_PATH})
```

1. Pada kode sumber include header file sehingga definisi fungsi dan kelas dapat dipanggil di badan kode.

```
{c++}
#include "PoissonSolver3DCylindricalGPU.h"

...
// create poissonSolver
PoissonSolver3DCylindricalGPU *poissonSolver = new PoissonSolver3DCylindricalGPU();
PoissonSolver3DCylindricalGPU::fgConvergenceError = 1e-8;
poissonSolver->SetExactSolution(VPotentialExact,kRows,kColumns, kPhiSlices);
poissonSolver->SetStrategy(PoissonSolver3DCylindricalGPU::kMultiGrid);
poissonSolver->SetCycleType(PoissonSolver3DCylindricalGPU::kFCycle);
poissonSolver->PoissonSolver3D(VPotential,RhoCharge,kRows,kColumns,kPhiSlices, kIterations,kSymmetry) ;
```

1. Buat folder terpisah untuk mebangun proyek yang menggunakan pustaka misal **buildexamplepoissonsolver** lalu jalankan **cmake** dengan flag spesial yaitu ∗∗-DCMAKE_PREFIX_PATH∗∗ yang ditetapkan dengan absolute path tempat direktori PoissonSolver3DCylindricalGPU dibangun.

```
$ mkdir buildexamplepoissonsolver
$ cd buildexamplepoissonsolver
$ cmake ../PoissonSolver3D/example/ -DCMAKE_PREFIX_PATH=/home/usertest/trypoissonsolver/buildpoissonsolver
...
-- Found CUDA: /apps/tools/cuda-9.1 (found version "9.1")
-- Building Poisson Solver  with CUDA support
-- Configuring done
-- Generating done
-- Build files have been written to: /home/usertest/trypoissonsolver/buildexamplepoissonsolver
```

1. Jalankan **make** untuk membuat *executable file*

```
$ make

-- Building Poisson Solver  with CUDA support
-- Configuring done
-- Generating done
-- Build files have been written to: /home/usertest/trypoissonsolver/buildexamplepoissonsolver
[ 50%] Linking CXX executable poissonsolvergputest
[100%] Built target poissonsolvergputest
```

1. Hasil dari **make** adalah program yang dapat dieksekusi yang berjalan di GPU card

```
$ ./poissonsolvergputest
Poisson Solver 3D Cylindrical GPU test
Ukuran grid (r,phi,z) = (17,17,18)
Waktu komputasi:            0.55 s
Jumlah iterasi siklus multigrid:    5
Iterasi Error Convergen     Error Absolut
[0]:    8.362587e-01    2.430797e-03
[1]:    5.126660e-02    4.434380e-04
[2]:    9.490424e-03    1.892288e-04
[3]:    2.084086e-03    1.314385e-04
[4]:    1.141812e-03    1.217592e-04

Poisson Solver 3D Cylindrical GPU test
Ukuran grid (r,phi,z) = (33,33,36)
Waktu komputasi:            1.000000e-02 s
Jumlah iterasi siklus multigrid:    10
Iterasi Error Convergen     Error Absolut
[0]:    1.724319e-03    1.315177e-04
[1]:    8.652242e-05    1.266415e-04
[2]:    5.450314e-05    1.256187e-04
[3]:    5.081671e-05    1.249848e-04
[4]:    4.951065e-05    1.244460e-04
[5]:    4.904093e-05    1.241379e-04
[6]:    4.890266e-05    1.239894e-04
[7]:    4.885175e-05    1.239208e-04
[8]:    4.883604e-05    1.238959e-04
[9]:    4.883501e-05    1.238844e-04

Poisson Solver 3D Cylindrical GPU test
Ukuran grid (r,phi,z) = (65,65,72)
Waktu komputasi:            5.000000e-02 s
Jumlah iterasi siklus multigrid:    3
Iterasi Error Convergen     Error Absolut
[0]:    1.084726e-04    1.291784e-04
```

```
[1]:    5.680057e-06    1.283386e-04
[2]:    1.501479e-06    1.278558e-04

Poisson Solver 3D Cylindrical GPU test
Ukuran grid (r,phi,z) = (129,129,144)
Waktu komputasi:            1.200000e-01 s
Jumlah iterasi siklus multigrid:    3
Iterasi Error Convergen     Error Absolut
[0]:    1.373293e-05    1.302576e-04
[1]:    2.154564e-06    1.299626e-04
[2]:    1.048384e-06    1.297261e-04

Poisson Solver 3D Cylindrical GPU test
Ukuran grid (r,phi,z) = (257,257,288)
Waktu komputasi:            6.900000e-01 s
Jumlah iterasi siklus multigrid:    5
Iterasi Error Convergen     Error Absolut
[0]:    5.469890e-06    1.388922e-04
[1]:    1.783695e-06    1.384964e-04
[2]:    1.345343e-06    1.381199e-04
[3]:    1.156555e-06    1.377524e-04
[4]:    1.050234e-06    1.373983e-04
```

Bab 3

Indeks Kelas

3.1 Daftar Kelas

Berikut ini daftar kelas, struct, union, dan interface, dengan penjelasan singkat:

Bab 4

Indeks File

4.1 Daftar File

Berikut ini daftar seluruh file yang didokumentasikan, dengan penjelasan singkat:

Bab 5

Dokumentasi Kelas

5.1 Referensi Struct PoissonSolver3DCylindricalGPU::MGParameters

Atribut Publik

- bool isFull3D

  TRUE: full coarsening, FALSE: semi coarsening.
- CycleType cycleType

  cycleType follow CycleType
- GridTransferType gtType

  gtType grid transfer type follow GridTransferType
- RelaxType relaxType

  relaxType follow RelaxType
- int gamma

  number of iteration at coarsest level
- int nPre

  number of iteration for pre smoothing
- int nPost

  number of iteration for post smoothing
- int nMGCycle

  number of multi grid cycle (V type)
- int maxLoop

  the number of tree-deep of multi grid

5.1.1 Keterangan Lengkap

Definisi pada baris 51 dalam file PoissonSolver3DCylindricalGPU.h.

Dokumentasi untuk struct ini dibangkitkan dari file berikut:

- /home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/interface/PoissonSolver3DCylindrical←
  GPU.h

5.2 Referensi Kelas PoissonSolver3DCylindricalGPU

Kelas ini merupakan interface PoissonSolver 3D dalam koordinat silindrikal yang diterapkan pada NVDIA Cuda.

```
#include <PoissonSolver3DCylindricalGPU.h>
```

Kelas

- struct MGParameters

Tipe Publik

- enum StrategyType { kRelaxation = 0, kMultiGrid = 1, kFastRelaxation = 2 }

  < Enumeration of Poisson Solver Strategy Type
- enum CycleType { kVCycle = 0, kWCycle = 1, kFCycle = 2 }
- enum GridTransferType { kHalf = 0, kFull = 1 }
- enum RelaxType { kJacobi = 0, kWeightedJacobi = 1, kGaussSeidel = 2 }
- enum InterpType { kHalfInterp = 0, kFullInterp = 1 }

Fungsi Anggota Publik

- void **SetExactSolution** (float ∗exactSolution, const int fPhiSlices)
- void **SetCycleType** (PoissonSolver3DCylindricalGPU::CycleType cycleType)
- PoissonSolver3DCylindricalGPU ()
- **PoissonSolver3DCylindricalGPU** (int nRRow, int nZColumn, int nPhiSlice)
- **PoissonSolver3DCylindricalGPU** (const char ∗name, const char ∗title)
- virtual ∼PoissonSolver3DCylindricalGPU ()

  destructor
- void PoissonSolver3D (float ∗matricesV, float ∗matricesChargeDensities, int nRRow, int nZColumn, int phi↩ Slice, int maxIterations, int symmetry)
- void **SetStrategy** (StrategyType strategy)
- StrategyType **GetStrategy** ()
- void SetExactSolution (float ∗exactSolution, int nRRow, int nZColumn, int phiSlice)
- float **GetErrorConv** (int iteration)
- float **GetErrorExact** (int iteration)

Atribut Publik

- int fIterations

  number of maximum iteration
- MGParameters fMgParameters

  parameters multi grid
- StrategyType fStrategy

  strategy used default multiGrid
- float ∗ **fExactSolutionF**

Atribut Publik Statis

- static const float fgkTPCZ0 = 249.7

  nominal gating grid position
- static const float fgkIFCRadius = 83.5

  Mean Radius of the Inner Field Cage (82.43 min, 83.70 max) (cm)
- static const float fgkOFCRadius = 254.5

  Mean Radius of the Outer Field Cage (252.55 min, 256.45 max) (cm)
- static const float fgkZOffSet = 0.2

  Offset from CE: calculate all distortions closer to CE as if at this point.
- static const float fgkCathodeV = -100000.0

  Cathode Voltage (volts)
- static const float fgkGG = -70.0

  Gating Grid voltage (volts)
- static const float fgkdvdE = 0.0024

  [cm/V] drift velocity dependency on the E field (from Magboltz for NeCO2N2 at standard environment)
- static const float fgkEM = -1.602176487e-19 / 9.10938215e-31

  charge/mass in [C/kg]
- static const float fgke0 = 8.854187817e-12

  vacuum permittivity [A·s/(V·m)]
- static float fgExactErr = 1e-4

  Error tolerated.
- static float fgConvergenceError = 1e-3

  Error tolerated.

5.2.1 Keterangan Lengkap

Kelas ini merupakan interface PoissonSolver 3D dalam koordinat silindrikal yang diterapkan pada NVDIA Cuda.

Penulis

Rifki Sadikin `rifki.sadikin@cern.ch`, Indonesian Institute of Sciences

Tanggal

Nov 8, 2018

Penulis

Rifki Sadikin `rifki.sadikin@cern.ch`, Indonesian Institute of Sciences

Tanggal

Nov 20, 2017

Definisi pada baris 15 dalam file PoissonSolver3DCylindricalGPU.h.

5.2.2 Dokumentasi Anggota: Enumerasi

5.2.2.1 CycleType

```
enum PoissonSolver3DCylindricalGPU::CycleType
```

Nilai enumerasi

kVCycle	V Cycle.
kWCycle	W Cycle (TODO)
kFCycle	Full Cycle.

Definisi pada baris 25 dalam file PoissonSolver3DCylindricalGPU.h.

```
25                {
26     kVCycle = 0,
27     kWCycle = 1,
28     kFCycle = 2
29   };
```

5.2.2.2 GridTransferType

`enum PoissonSolver3DCylindricalGPU::GridTransferType`

Nilai enumerasi

kHalf	Half weighting.
kFull	Full weighting.

Definisi pada baris 32 dalam file PoissonSolver3DCylindricalGPU.h.

```
32                       {
33     kHalf = 0,
34     kFull = 1,
35   };
```

5.2.2.3 InterpType

`enum PoissonSolver3DCylindricalGPU::InterpType`

Nilai enumerasi

kHalfInterp	Half bi linear interpolation.
kFullInterp	Full bi linear interpolation.

Definisi pada baris 45 dalam file PoissonSolver3DCylindricalGPU.h.

```
45                 {
46     kHalfInterp = 0,
47     kFullInterp = 1
48   };
```

5.2.2.4 RelaxType

`enum PoissonSolver3DCylindricalGPU::RelaxType`

Nilai enumerasi

kJacobi	Jacobi (5 Stencil 2D, 7 Stencil 3D_.
kWeightedJacobi	(TODO)
kGaussSeidel	Gauss Seidel 2D (2 Color, 5 Stencil), 3D (7 Stencil)

Definisi pada baris 38 dalam file PoissonSolver3DCylindricalGPU.h.

```
38               {
39     kJacobi = 0,
40     kWeightedJacobi = 1,
41     kGaussSeidel = 2
42   };
```

5.2.2.5 StrategyType

`enum PoissonSolver3DCylindricalGPU::StrategyType`

< Enumeration of Poisson Solver Strategy Type

Nilai enumerasi

kRelaxation	S.O.R Cascaded MultiGrid.
kMultiGrid	Geometric MG.
kFastRelaxation	Spectral (TODO)

Definisi pada baris 18 dalam file PoissonSolver3DCylindricalGPU.h.

```
18                  {
19     kRelaxation = 0,
20     kMultiGrid = 1,
21     kFastRelaxation = 2
22   };
```

5.2.3 Dokumentasi Konstruktor & Destruktor

5.2.3.1 PoissonSolver3DCylindricalGPU()

`PoissonSolver3DCylindricalGPU::PoissonSolver3DCylindricalGPU ( )`

constructor

Definisi pada baris 48 dalam file PoissonSolver3DCylindricalGPU.cxx.

```
48                                                            {
49
50     fErrorConvF = new float [fMgParameters.nMGCycle];
51     fErrorExactF = new float [fMgParameters.nMGCycle];
52
53 }
```

5.2.4 Dokumentasi Anggota: Fungsi

5.2.4.1 PoissonSolver3D()

```
void PoissonSolver3DCylindricalGPU::PoissonSolver3D (
            float * matricesV,
            float * matricesCharge,
            int nRRow,
            int nZColumn,
            int phiSlice,
            int maxIteration,
            int symmetry )
```

Menyediakan solusi iteratif terhadap poisson solver pada koordinat silindrikal 3D

Disediakan algoritma iteratif

- Geometric MultiGrid
  - Cycles: V, W, Full
  - Relaxation: Gauss-Seidel
  - Grid transfer operators: Full, Half

Parameter

matricesV	float ∗ potential dalam array 1D berukuran nRRow∗nZColumn ∗phiSlice
matricesCharge	float ∗ charge dalam array 1D berukuran nRRow∗nZColumn ∗phiSlice
nRRow	int jumlah titik grid pada arah radial
nZColumn	int jumlah titik grid pada arah z
phiSlice	int jumlah titik grid pada arah sudut (phi)
maxIteration	int jumlah iterasi maksimum pada multigrud
symmetry	int nilai simetri (tidak dipakai)

Definisi pada baris 85 dalam file PoissonSolver3DCylindricalGPU.cxx.

```
                                                        {

  fNRRow = nRRow;
  fNZColumn = nZColumn;
  fPhiSlice = phiSlice;

  PoissonMultiGrid3D2D(matricesV, matricesCharge, nRRow, nZColumn, phiSlice, symmetry);


}
```

5.2.4.2 SetExactSolution()

```
void PoissonSolver3DCylindricalGPU::SetExactSolution (
            float * exactSolution,
            int nRRow,
            int nZColumn,
            int phiSlice )
```

Helper untuk menset nilai V dari fungsi analitik (diperlukan untuk memastikan implementasi benar

Parameter

exactSolution	float ∗ array 1D sebesar nRRow ∗ nZColumn ∗ phiSlice
nRRow	int jumlah titik grid pada arah radial
nZColumn	int jumlah titik grid pada arah z
phiSlice	int jumlah titik grid pada arah sudut (phi)

Definisi pada baris 177 dalam file PoissonSolver3DCylindricalGPU.cxx.

```
{
    fNRRow = nRRow;
    fNZColumn = nZColumn;
    fPhiSlice = phiSlice;
    fExactSolutionF = new float[fNRRow * fPhiSlice * fNZColumn];
    fExactPresent = true;
    fMaxExact = 0.0;;
    for (int i=0;i<nRRow*nZColumn*phiSlice;i++) {
        fExactSolutionF[i] = exactSolution[i];
        if (abs(fExactSolutionF[i]) > fMaxExact) fMaxExact = abs(fExactSolutionF[i]);
    }
}
```

5.2.5 Dokumentasi Anggota: Data

5.2.5.1 fgkIFCRadius

`const float PoissonSolver3DCylindricalGPU::fgkIFCRadius = 83.5 [static]`

Mean Radius of the Inner Field Cage (82.43 min, 83.70 max) (cm)

radius which renders the "18 rod manifold" best -> compare calc. of Jim Thomas

Definisi pada baris 80 dalam file PoissonSolver3DCylindricalGPU.h.

Dokumentasi untuk kelas ini dibangkitkan dari file-file berikut:

- /home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/interface/PoissonSolver3DCylindrical↩ GPU.h
- /home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/interface/PoissonSolver3DCylindrical↩ GPU.cxx

Bab 6

Dokumentasi File

6.1 Referensi File /home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3↩ D/example/PoissonSolver3DGPUTest.h

Berkas ini berisi definisi fungs untuk memakai pustaka libPoissonSolver3DCylindrialGPU.so.

```
#include <iostream>
#include "PoissonSolver3DCylindricalGPU.h"
```

Fungsi

- void DoPoissonSolverExperiment (const int kRows, const int kColumns, const int kPhiSlices, const int k↩ Iterations, const int kSymmetry)

  Lakukan eksperimen poisson solver.
- void InitVoltandCharge3D (float ∗VPotentialExact, float ∗VPotential, float ∗RhoCharge, const int kRows, const int kColumns, const int kPhiSlices, float gridSizeR, float gridSizeZ, float gridSizePhi)

  Inisisasi nilai volt dan charge.
- float TestFunction1PotentialEval (double a, double b, double c, float radius0, float phi0, float z0)

  analytic function untuk potensial
- float TestFunction1ChargeEval (double a, double b, double c, float radius0, float phi0, float z0)

  analytic function untuk charge

Variabel

- const double fgkTPCZ0 = 249.7

  nominal gating grid position
- const double fgkIFCRadius = 83.5

  radius which renders the "18 rod manifold" best -> compare calc. of Jim Thomas
- const double fgkOFCRadius = 254.5

  Mean Radius of the Outer Field Cage (252.55 min, 256.45 max) (cm)
- const double fgkZOffSet = 0.2

  Offset from CE: calculate all distortions closer to CE as if at this point.
- const double fgkCathodeV = -100000.0

  Cathode Voltage (volts)

- const double fgkGG = -70.0

  Gating Grid voltage (volts)
- const double fgkdvdE = 0.0024

  [cm/V] drift velocity dependency on the E field (from Magboltz for NeCO2N2 at standard environment)
- const double fgkEM = -1.602176487e-19 / 9.10938215e-31

  charge/mass in [C/kg]
- const double fgke0 = 8.854187817e-12

  vacuum permittivity [A·s/(V·m)]
- const double fgConvergenceError = 1e-6

  vacuum permittivity [A·s/(V·m)]

6.1.1 Keterangan Lengkap

Berkas ini berisi definisi fungs untuk memakai pustaka libPoissonSolver3DCylindrialGPU.so.

Penulis

Rifki Sadikin rifki.sadikin@lipi.go.id, Pusat Penelitian Informatika, Lembaga Ilmu Pengetahuan Indonesia
I Wayan Aditya Swardiana i.wayan.aditya.swardiana@lipi.go.id, Pusat Penelitian Informatika, Lembaga Ilmu Pengetahuan Indonesia

Tanggal

November 8, 2018

6.1.2 Dokumentasi Fungsi

6.1.2.1 DoPoissonSolverExperiment()

```
void DoPoissonSolverExperiment (
            const int kRows,
            const int kColumns,
            const int kPhiSlices,
            const int kIterations,
            const int kSymmetry )
```

Lakukan eksperimen poisson solver.

```
const float  ratioPhi    =  gridSizeR*gridSizeR / (gridSizePhi*gridSizePhi) ;  // ratio_{phi} = gridsize_{r} /
const float  ratioZ      =  gridSizeR*gridSizeR / (gridSizeZ*gridSizeZ) ; // ratio_{Z} = gridsize_{r} / gridsi
const float  convErr       =  fgConvergenceError;
const float  IFCRadius   =  fgkIFCRadius;

const int  fparamsize = 8;
float * fparam = new float[fparamsize];

fparam[0] = gridSizeR;
fparam[1] = gridSizePhi;
fparam[2] = gridSizeZ;
fparam[3] = ratioPhi;
fparam[4] = ratioZ;
fparam[5] = convErr;
fparam[6] = IFCRadius;
```

```
int  iparamsize = 4;
int  * iparam = new int[iparamsize];

iparam[0] = 2;//nPre
iparam[1] = 2;//nPost;
iparam[2] = 6;//maxLoop;
iparam[3] = 200; //nMGCycle;
for (int k=0;k<kPhiSlices;k++) {
for (int i=0;i<kRows;i++) {
        for (int j=0;j< kColumns;j++) printf("%.3f\t",RhoCharge[k * (kRows * kColumns) + i * kColumns + j]);
        printf("\n");
}
printf("\n");
}
```

VCycle PoissonMultigrid3DSemiCoarseningGPUError(VPotential, RhoCharge, kRows, kColumns ,kPhiSlices, 0 , fparam, iparam, true, errorConv,errorExact, VPotentialExact); Call poisson solver

for (int k=0;k<1;k++) { for (int i=0;i<kRows;i++) { for (int j=0;j< kColumns;j++) printf("%.3f\t",VPotentialExact[k∗ (kRows ∗ kColumns) + i ∗ kColumns + j]); printf("\n"); } printf("\n"); }

for (int k=0;k<1;k++) { for (int i=0;i<kRows;i++) { for (int j=0;j< kColumns;j++) printf("%.3f\t",VPotential[k∗ (kRows ∗ kColumns) + i ∗ kColumns + j]); printf("\n"); } printf("\n"); }

Definisi pada baris 15 dalam file PoissonSolver3DGPUTest.cpp.

```
15                                      {
16
17
18       int kPhiSlicesPerSector = kPhiSlices/18;
19
20       const float gridSizeR   =  (fgkOFCRadius-fgkIFCRadius) / (kRows-1) ;
21       const float gridSizeZ   =  fgkTPCZ0 / (kColumns-1) ;
22
23       const float gridSizePhi =  (M_PI * 2)/ ( 18.0 * kPhiSlicesPerSector);
24
25       int size = kRows * kColumns * kPhiSlices;
26
27       float * VPotential = new float[size];
28       float * VPotentialExact  = new float[size];
29       float * RhoCharge  = new float[size];
30       float * errorConv  = new float[200];
31       float * errorExact  = new float[200];
32
33       InitVoltandCharge3D(VPotentialExact,VPotential,RhoCharge,kRows,kColumns,kPhiSlices,
        gridSizeR,gridSizeZ,gridSizePhi);
34
35
82       // create poissonSolver
83       PoissonSolver3DCylindricalGPU *poissonSolver = new
        PoissonSolver3DCylindricalGPU();
84
85       PoissonSolver3DCylindricalGPU::fgConvergenceError = 1e
        -6;
86       // zeroring array of error
87       poissonSolver->SetExactSolution(VPotentialExact,kRows,kColumns, kPhiSlices);
88
89
90
91       // Case 1. Set the strategy as multigrid, fullmultigrid, and full 3d
92       poissonSolver->SetStrategy(PoissonSolver3DCylindricalGPU::kMultiGrid
        );
93       poissonSolver->SetCycleType(PoissonSolver3DCylindricalGPU::kFCycle
        );
94
95       // TStopwatch w;
96       std::clock_t start,stop;
97       double duration;
98
99           start = std::clock();
100
101       //w.Start();
102       poissonSolver->PoissonSolver3D(VPotential,RhoCharge,kRows,kColumns,kPhiSlices,
        kIterations,kSymmetry) ;
103       //w.Stop();
```

```
    stop = std::clock();
        duration = ( stop - start ) / (double) CLOCKS_PER_SEC;

    std::cout<<"Poisson Solver 3D Cylindrical GPU test" << '\n';
    std::cout<<"Ukuran grid (r,phi,z) = (" << kRows << "," << kColumns << "," << kPhiSlices <<") \n";
        std::cout<<"Waktu komputasi: \t\t\t"<< duration <<" s \n";
    std::cout<<"Jumlah iterasi siklus multigrid: \t"<< poissonSolver->fIterations << '\n';
    std::cout<<"Iterasi\tError Convergen\t\tError Absolut \n";
    std::cout<< std::scientific;
    for (int i=0;i<poissonSolver->fIterations;i++) {
        std::cout<<"[" <<  i  << "]:\t" << poissonSolver->GetErrorConv(i) << "\t" << poissonSolver->GetErrorExact(i) << '\n' << std::scientific;
    }
    delete poissonSolver;
    delete VPotential;
    delete VPotentialExact;
    delete RhoCharge;
}
```

6.2 Referensi File /home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/interface/PoissonSolver3DCylindricalGPU.h

```
#include "PoissonSolver3DGPU.h"
```

Kelas

- class PoissonSolver3DCylindricalGPU

  Kelas ini merupakan interface PoissonSolver 3D dalam koordinat silindrikal yang diterapkan pada NVDIA Cuda.
- struct PoissonSolver3DCylindricalGPU::MGParameters

6.3 Referensi File /home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/kernel/PoissonSolver3DGPU.cu

Berkas ini berisi implementasi kernel dalam cuda untuk PoissonSolver Cylindrical berbasis Multigrid.

```
#include "PoissonSolver3DGPU.h"
#include <cuda.h>
#include <math.h>
```

Fungsi

- __global__ void relaxationGaussSeidelRed (float ∗VPotential, float ∗RhoChargeDensity, const int RRow, const int ZColumn, const int PhiSlice, float ∗coef1, float ∗coef2, float ∗coef3, float ∗coef4)
- __global__ void relaxationGaussSeidelBlack (float ∗VPotential, float ∗RhoChargeDensity, const int RRow, const int ZColumn, const int PhiSlice, float ∗coef1, float ∗coef2, float ∗coef3, float ∗coef4)
- __global__ void residueCalculation (float ∗VPotential, float ∗RhoChargeDensity, float ∗DeltaResidue, const int RRow, const int ZColumn, const int PhiSlice, float ∗coef1, float ∗coef2, float ∗coef3, float ∗icoef4)
- __global__ void restriction2DFull (float ∗RhoChargeDensity, float ∗DeltaResidue, const int RRow, const int ZColumn, const int PhiSlice)
- __global__ void **zeroingVPotential** (float ∗VPotential, const int RRow, const int ZColumn, const int PhiSlice)
- __global__ void **zeroingBoundaryTopBottom** (float ∗VPotential, int RRow, int ZColumn, int PhiSlice)
- __global__ void **zeroingBoundaryLeftRight** (float ∗VPotential, int RRow, int ZColumn, int PhiSlice)

- __global__ void **prolongation2DHalf** (float ∗VPotential, const int RRow, const int ZColumn, const int Phi←Slice)
- __global__ void **prolongation2DHalfNoAdd** (float ∗VPotential, const int RRow, const int ZColumn, const int PhiSlice)
- __global__ void **errorCalculation** (float ∗VPotentialPrev, float ∗VPotential, float ∗EpsilonError, const int R←Row, const int ZColumn, const int PhiSlice)
- float **GetErrorNorm2** (float ∗VPotential, float ∗VPotentialPrev, const int rows, const int cols, float weight)
- float **GetAbsMax** (float ∗VPotentialExact, int size)
- void Restrict_Boundary (float ∗VPotential, const int RRow, const int ZColumn, const int PhiSlice, const int Offset)
- void PrintMatrix (float ∗Mat, const int Row, const int Column)
- void **VCycleSemiCoarseningGPU** (float ∗d_VPotential, float ∗d_RhoChargeDensity, float ∗d_DeltaResidue, float ∗d_coef1, float ∗d_coef2, float ∗d_coef3, float ∗d_coef4, float ∗d_icoef4, float gridSizeR, float ratioZ, float ratioPhi, int RRow, int ZColumn, int PhiSlice, int gridFrom, int gridTo, int nPre, int nPost)
- void PoissonMultigrid3DSemiCoarseningGPUError (float ∗VPotential, float ∗RhoChargeDensity, const int R←Row, const int ZColumn, const int PhiSlice, const int Symmetry, float ∗fparam, int ∗iparam, bool isExact←Present, float ∗errorConv, float ∗errorExact, float ∗VPotentialExact)
- void PoissonMultigrid3DSemiCoarseningGPUErrorWCycle (float ∗VPotential, float ∗RhoChargeDensity, const int RRow, const int ZColumn, const int PhiSlice, const int Symmetry, float ∗fparam, int ∗iparam, float ∗errorConv, float ∗errorExact, float ∗VPotentialExact)
- void PoissonMultigrid3DSemiCoarseningGPUErrorFCycle (float ∗VPotential, float ∗RhoChargeDensity, const int RRow, const int ZColumn, const int PhiSlice, const int Symmetry, float ∗fparam, int ∗iparam, bool is←ExactPresent, float ∗errorConv, float ∗errorExact, float ∗VPotentialExact)

Variabel

- __device__ __constant__ int **d_coef_StartPos**
- __device__ __constant__ int **d_grid_StartPos**
- __device__ __constant__ float **d_h2**
- __device__ __constant__ float **d_ih2**
- __device__ __constant__ float **d_tempRatioZ**

6.3.1 Keterangan Lengkap

Berkas ini berisi implementasi kernel dalam cuda untuk PoissonSolver Cylindrical berbasis Multigrid.

Penulis

Rifki Sadikin rifki.sadikin@lipi.go.id, Pusat Penelitian Informatika, Lembaga Ilmu Pengetahuan Indonesia
I Wayan Aditya Swardiana i.wayan.aditya.swardiana@lipi.go.id, Pusat Penelitian Informatika, Lembaga Ilmu Pengetahuan Indonesia

Tanggal

November 8, 2018

6.3.2 Dokumentasi Fungsi

6.3.2.1 PoissonMultigrid3DSemiCoarseningGPUError()

```
void PoissonMultigrid3DSemiCoarseningGPUError (
            float * VPotential,
            float * RhoChargeDensity,
            const int RRow,
            const int ZColumn,
            const int PhiSlice,
            const int Symmetry,
            float * fparam,
            int * iparam,
            bool isExactPresent,
            float * errorConv,
            float * errorExact,
            float * VPotentialExact )
```

Fungsi ini menghitung solusi terhadap Persamaan Poisson

$$\nabla^2(r,\phi,z) = \rho(r,\phi,z)$$

dengan diketahui nilai tepi (Boundary Value) pada V (potensial) dan distribusi buatan ρ

Parameter

in,out	*VPotential*	float[nrows∗ncols∗nphi] distribusi potensial. Input: hanya nilai tepi. Output: hasil perhitungan penyelesaian persamaan Poisson
in	*RhoChangeDensity*	float[nrows∗ncols∗nphi] distributsi muatan listrik.

Mengembalikan

A fixed number that has nothing to do with what the function does

Definisi pada baris 893 dalam file PoissonSolver3DGPU.cu.

```
{
    // variables for CPU memory
    float *temp_VPotential;
    float *VPotentialPrev;
    float *EpsilonError;

    // variables for GPU memory
    float *d_VPotential;
    float *d_RhoChargeDensity;
    float *d_DeltaResidue;
    float *d_VPotentialPrev;
    float *d_EpsilonError;

    float *d_coef1;
    float *d_coef2;
    float *d_coef3;
    float *d_coef4;
    float *d_icoef4;

    // variables for coefficent calculations
    float *coef1;
    float *coef2;
    float *coef3;
    float *coef4;
```

```
    float *icoef4;
    float tempRatioZ;
    float tempRatioPhi;
    float radius;

    int gridFrom;
    int gridTo;
    int loops;


    // variables passed from ALIROOT
    float gridSizeR      = fparam[0];
    float gridSizePhi    = fparam[1];
    float gridSizeZ      = fparam[2];
    float ratioPhi       = fparam[3];
    float ratioZ         = fparam[4];
    float convErr        = fparam[5];
    float IFCRadius      = fparam[6];
    int nPre         = iparam[0];
    int nPost        = iparam[1];
    int maxLoop      = iparam[2];
    int nCycle       = iparam[3];

    // variables for calculating GPU memory allocation
    int grid_RRow;
    int grid_ZColumn;
    int grid_PhiSlice = PhiSlice;
    int grid_Size = 0;
    int grid_StartPos;
    int coef_Size = 0;
    int coef_StartPos;
    int iOne, jOne;
    float h, h2, ih2;

    // variables for calculating multigrid maximum depth
    int depth_RRow = 0;
    int depth_ZColumn = 0;
    int temp_RRow = RRow;
    int temp_ZColumn = ZColumn;

    // calculate depth for multigrid
    while (temp_RRow >>= 1) depth_RRow++;
    while (temp_ZColumn >>= 1) depth_ZColumn++;

    loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
    loops = (loops > maxLoop) ? maxLoop : loops;

    gridFrom = 1;
    gridTo = loops;

    // calculate GPU memory allocation for multigrid
    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
        grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;

        grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
        coef_Size += grid_RRow;
    }

    // allocate memory for temporary output
    temp_VPotential          = (float *) malloc(grid_Size * sizeof(float));
    VPotentialPrev = (float *) malloc(RRow * ZColumn * PhiSlice * sizeof(float));
    EpsilonError = (float *) malloc(1 * sizeof(float));


    // allocate memory for relaxation coefficient
    coef1 = (float *) malloc(coef_Size * sizeof(float));
    coef2 = (float *) malloc(coef_Size * sizeof(float));
     coef3 = (float *) malloc(coef_Size * sizeof(float));
     coef4 = (float *) malloc(coef_Size * sizeof(float));
     icoef4 = (float *) malloc(coef_Size * sizeof(float));

     // pre-compute relaxation coefficient
     coef_StartPos = 0;
     iOne = 1 << (gridFrom - 1);
     jOne = 1 << (gridFrom - 1);

     for (int step = gridFrom; step <= gridTo; step++)
     {
         grid_RRow = ((RRow - 1) / iOne) + 1;

         h = gridSizeR * iOne;
         h2 = h * h;
         ih2 = 1.0 / h2;

         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
```

```
        tempRatioPhi = ratioPhi * iOne * iOne;

        for (int i = 1; i < grid_RRow - 1; i++)
        {
            radius = IFCRadius + i * h;
            coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
            coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
            coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
            coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
            icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
        }
        coef_StartPos += grid_RRow;
        iOne = 2 * iOne;
        jOne = 2 * jOne;
    }

    // device memory allocation
    cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
    cudaMalloc( &d_VPotentialPrev, RRow * ZColumn * PhiSlice * sizeof(float) );
    cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );
    cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
    cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
    cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
    cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );

    // set memory to zero
    cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
    cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
    cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
    cudaMemset( d_VPotentialPrev, 0, RRow * ZColumn * PhiSlice * sizeof(float) );
    cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );


    // copy data from host to device
    cudaMemcpy( d_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice
 ); //check
    cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyHostToDevice ); //check
    cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyHostToDevice );

    // max exact

    // float maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
    float maxAbsExact = 1.0;

    if (isExactPresent == true)
        maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
    dim3 error_BlockPerGrid((RRow < 16) ? 1 : (RRow / 16), (ZColumn < 16) ? 1 : (ZColumn / 16), PhiSlice);
    dim3 error_ThreadPerBlock(16, 16);


    for (int cycle = 0; cycle < nCycle; cycle++)
    {
        cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
        if (isExactPresent == true) errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact,
RRow * PhiSlice,ZColumn, maxAbsExact);


        VCycleSemiCoarseningGPU(d_VPotential, d_RhoChargeDensity, d_DeltaResidue, d_coef1, d_coef2, d_coef3
, d_coef4, d_icoef4, gridSizeR, ratioZ, ratioPhi, RRow, ZColumn, PhiSlice, gridFrom, gridTo, nPre, nPost);


        errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev, d_VPotential,
d_EpsilonError, RRow, ZColumn, PhiSlice);

        cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );


        errorConv[cycle] = *EpsilonError  / (RRow * ZColumn * PhiSlice);

        if (errorConv[cycle] < convErr)
        {
            //errorConv
            nCycle = cycle;
            break;
        }

```

```
        cudaMemcpy( d_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );
        cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );

    }
    iparam[3] = nCycle;

//  for (int cycle = 0; cycle < nCycle; cycle++)
//  {
//      cudaMemcpy( temp_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );


//      VCycleSemiCoarseningGPU(d_VPotential, d_RhoChargeDensity, d_DeltaResidue, d_coef1, d_coef2, d_coef3, d_coef4, d_icoef4, gridSizeR, ratioZ, ratioPhi, RRow, ZColumn, PhiSlice, gridFrom, gridTo, nPre, nPost);

//      cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
//      errorConv[cycle] = GetErrorNorm2(temp_VPotential, temp_VPotentialPrev, RRow * PhiSlice, ZColumn, 1.0);
//      //errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact, RRow * PhiSlice,ZColumn, 1.0);
//  }


    // copy result from device to host
    cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );

    memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));

    // free device memory
    cudaFree( d_VPotential );
    cudaFree( d_DeltaResidue );
    cudaFree( d_RhoChargeDensity );
    cudaFree( d_VPotentialPrev );
    cudaFree( d_EpsilonError );
    cudaFree( d_coef1 );
    cudaFree( d_coef2 );
    cudaFree( d_coef3 );
    cudaFree( d_coef4 );
    cudaFree( d_icoef4 );

    // free host memory
    free( coef1 );
    free( coef2 );
    free( coef3 );
    free( coef4 );
    free( icoef4 );
    free( temp_VPotential );
    free( VPotentialPrev );
}
```

6.3.2.2 PoissonMultigrid3DSemiCoarseningGPUErrorFCycle()

```
void PoissonMultigrid3DSemiCoarseningGPUErrorFCycle (
            float * VPotential,
            float * RhoChargeDensity,
            const int RRow,
            const int ZColumn,
            const int PhiSlice,
            const int Symmetry,
            float * fparam,
            int * iparam,
            bool isExactPresent,
            float * errorConv,
            float * errorExact,
            float * VPotentialExact )
```

cudaMemcpy(temp_VPotential, d_RhoChargeDensity + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost);

Definisi pada baris 1837 dalam file PoissonSolver3DGPU.cu.

```
{
    // variables for CPU memory
    float *temp_VPotential;
    float *VPotentialPrev;
    float *EpsilonError;

    // variables for GPU memory
    float *d_VPotential;
    float *d_RhoChargeDensity;
    float *d_DeltaResidue;
    float *d_coef1;
    float *d_coef2;
    float *d_coef3;
    float *d_coef4;
    float *d_icoef4;
    float *d_VPotentialPrev;
    float *d_EpsilonError;


    // variables for coefficent calculations
    float *coef1;
    float *coef2;
    float *coef3;
    float *coef4;
    float *icoef4;
    float tempRatioZ;
    float tempRatioPhi;
    float radius;

    int gridFrom;
    int gridTo;
    int loops;

    // variables passed from ALIROOT
    float gridSizeR     = fparam[0];
    //float gridSizePhi = fparam[1];
    //float gridSizeZ       = fparam[2];
    float ratioPhi      = fparam[3];
    float ratioZ        = fparam[4];
    float convErr       = fparam[5];
    float IFCRadius     = fparam[6];
    int nPre    = iparam[0];
    int nPost   = iparam[1];
    int maxLoop = iparam[2];
    int nCycle  = iparam[3];

    // variables for calculating GPU memory allocation
    int grid_RRow;
    int grid_ZColumn;
    int grid_PhiSlice = PhiSlice;
    int grid_Size = 0;
    int grid_StartPos;
    int coef_Size = 0;
    int coef_StartPos;
    int iOne, jOne;
    float h, h2, ih2;

    // variables for calculating multigrid maximum depth
    int depth_RRow = 0;
    int depth_ZColumn = 0;
    int temp_RRow = RRow;
    int temp_ZColumn = ZColumn;

    // calculate depth for multigrid
    while (temp_RRow >>= 1) depth_RRow++;
    while (temp_ZColumn >>= 1) depth_ZColumn++;

    loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
    loops = (loops > maxLoop) ? maxLoop : loops;

    gridFrom = 1;
    gridTo = loops;

    // calculate GPU memory allocation for multigrid
    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
        grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;

        grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
        coef_Size += grid_RRow;
    }

    // allocate memory for temporary output
    temp_VPotential         = (float *) malloc(grid_Size * sizeof(float));
    VPotentialPrev = (float *) malloc(grid_Size * sizeof(float));
    EpsilonError = (float *) malloc(1 * sizeof(float));
```

```
    for (int i=0;i<grid_Size;i++) temp_VPotential[i] = 0.0;


    // allocate memory for relaxation coefficient
    coef1 = (float *) malloc(coef_Size * sizeof(float));
    coef2 = (float *) malloc(coef_Size * sizeof(float));
    coef3 = (float *) malloc(coef_Size * sizeof(float));
    coef4 = (float *) malloc(coef_Size * sizeof(float));
    icoef4 = (float *) malloc(coef_Size * sizeof(float));

    // pre-compute relaxation coefficient
    // restrict boundary
    coef_StartPos = 0;
    grid_StartPos = 0;

    iOne = 1 << (gridFrom - 1);
    jOne = 1 << (gridFrom - 1);

    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / iOne) + 1;
        grid_ZColumn = ((ZColumn - 1) / iOne) + 1;

        h = gridSizeR * iOne;
        h2 = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        for (int i = 1; i < grid_RRow - 1; i++)
        {
            radius = IFCRadius + i * h;
            coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
            coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
            coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
            coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
            icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
        }

        // call restrict boundary
        if (step == gridFrom) {
            // Copy original VPotential to tempPotential
            memcpy(temp_VPotential,     VPotential, RRow * ZColumn * PhiSlice * sizeof(float));

        }
        // else
        //{
        //  Restrict_Boundary(temp_VPotential, grid_RRow, grid_ZColumn, PhiSlice, grid_StartPos);
        //}


        coef_StartPos += grid_RRow;
        grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;


        iOne = 2 * iOne;
        jOne = 2 * jOne;
    }

    // device memory allocation
    cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
    cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
    cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
    cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
    cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );
    cudaMalloc( &d_VPotentialPrev, grid_Size * sizeof(float) );
    cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );


    // set memory to zero
    cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
    cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
    cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
    cudaMemset( d_VPotentialPrev, 0, grid_Size * sizeof(float) );
    cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );

    // set memory to zero
    cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
    cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
    cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
```

```

    // copy data from host to devicei
    // case of FCycle you need to copy all boundary for all
    cudaMemcpy( d_VPotential, temp_VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice ); //check
//  cudaMemcpy( d_VPotential, VPotential, grid_Size * isizeof(float), cudaMemcpyHostToDevice ); //check

    cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice ); //check
//  cudaMemcpy( d_RhoChargeDensity, temp_VPotentialPrev, grid_Size * sizeof(float), cudaMemcpyHostToDevice ); //check
    cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
//  cudaMemcpy( d_VPotentialPrev, temp_VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice );

//  cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice );

    // max exact

    float maxAbsExact = 1.0;

    if (isExactPresent == true)
        maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);



    // init iOne,grid_RRow, grid_ZColumn, grid_StartPos, coef_StartPos
    iOne = 1 << (gridFrom - 1);
    jOne = 1 << (gridFrom - 1);

    grid_RRow       = ((RRow - 1) / iOne) + 1;
    grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

    grid_StartPos = 0;
    coef_StartPos = 0;


    for (int step = gridFrom + 1; step <= gridTo; step++)
    {

        iOne = 1 << (step - 1);
        jOne = 1 << (step - 1);

        grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
        coef_StartPos += grid_RRow;

        grid_RRow       = ((RRow - 1) / iOne) + 1;
        grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

        // pre-compute constant memory
        h   = gridSizeR * iOne;
        h2  = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        // copy constant to device memory
        cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

        // set kernel grid size and block size
        dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
        dim3 grid_ThreadPerBlock(16, 16);

        // restriction
        restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice );

        // restrict boundary (already done in cpu)
//      PrintMatrix(temp_VPotential,grid_RRow * PhiSlice,grid_ZColumn);
        restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );


    }

    dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
    dim3 grid_ThreadPerBlock(16, 16);
```

```


    // relax on the coarsest
    // red-black gauss seidel relaxation (nPre times)
//  printf("rho\n");
//  cudaMemcpy( temp_VPotential, d_RhoChargeDensity + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
//  PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);

//  printf("v\n");
//  cudaMemcpy( temp_VPotential, d_VPotential + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
//  PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);
    for (int i = 0; i < nPre; i++)
    {
        relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
        cudaDeviceSynchronize();
        relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
        cudaDeviceSynchronize();
    }

//  printf("v after relax\n");
//  cudaMemcpy( temp_VPotential, d_VPotential + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
//  PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);

    // V-Cycle => from coarser to finer grid
    for (int step = gridTo -1 ; step >= gridFrom; step--)
    {
        iOne = iOne / 2;
        jOne = jOne / 2;

        grid_RRow       = ((RRow - 1) / iOne) + 1;
        grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

        grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
        coef_StartPos -= grid_RRow;

        h   = gridSizeR * iOne;
        h2  = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        // copy constant to device memory
        cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );



        // set kernel grid size and block size
        dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : ( grid_ZColumn / 16), PhiSlice);
        dim3 grid_ThreadPerBlock(16, 16);


        prolongation2DHalfNoAdd<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );




        // just

        // max exact
        cudaMemcpy( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, grid_RRow * grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );

        float maxAbsExact = 1.0;

        if (isExactPresent == true)
            maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
        dim3 error_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : ( grid_ZColumn / 16), PhiSlice);
        dim3 error_ThreadPerBlock(16, 16);



        for (int cycle = 0; cycle < nCycle; cycle++)
        {

```

```
            if (step == gridFrom) {
                cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
                if (isExactPresent == true )errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact, RRow * PhiSlice,ZColumn, maxAbsExact);
            }
            //cudaDeviceSynchronize();
            VCycleSemiCoarseningGPU(d_VPotential, d_RhoChargeDensity, d_DeltaResidue, d_coef1, d_coef2, d_coef3, d_coef4, d_icoef4, gridSizeR, ratioZ, ratioPhi, RRow, ZColumn, PhiSlice, step, gridTo, nPre, nPost);
                //if (step == gridFrom) {
                //cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
                //errorConv[cycle] = GetErrorNorm2(temp_VPotential, VPotentialPrev, RRow * PhiSlice,ZColumn, 1.0);
                errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, d_EpsilonError, grid_RRow, grid_ZColumn, PhiSlice);
                cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );
                errorConv[cycle] = *EpsilonError  / (grid_RRow * grid_ZColumn * PhiSlice);
                if (errorConv[cycle]< convErr)
                {
                    nCycle = cycle;
                    break;
                }
                cudaMemcpy( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, grid_RRow * grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );
                cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
        }
    }
    iparam[3] = nCycle;
    // copy result from device to host
    cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
    memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
    // free device memory
    cudaFree( d_VPotential );
    cudaFree( d_DeltaResidue );
    cudaFree( d_RhoChargeDensity );
    cudaFree( d_coef1 );
    cudaFree( d_coef2 );
    cudaFree( d_coef3 );
    cudaFree( d_coef4 );
    cudaFree( d_icoef4 );
    // free host memory
    free( coef1 );
    free( coef2 );
    free( coef3 );
    free( coef4 );
    free( icoef4 );
    free( temp_VPotential );
    free( VPotentialPrev );
}
```

6.3.2.3 PoissonMultigrid3DSemiCoarseningGPUErrorWCycle()

```
void PoissonMultigrid3DSemiCoarseningGPUErrorWCycle (
        float * VPotential,
        float * RhoChargeDensity,
        const int RRow,
        const int ZColumn,
```

```
const int PhiSlice,
const int Symmetry,
float * fparam,
int * iparam,
float * errorConv,
float * errorExact,
float * VPotentialExact )
```

innner w cycle up one down one

end up one down on

up two down two

up one down one

end up one down one

Definisi pada baris 1147 dalam file PoissonSolver3DGPU.cu.

```
1160 {
1161     // variables for CPU memory
1162     float *temp_VPotential;
1163     float *VPotentialPrev;
1164     float *EpsilonError;
1165
1166     // variables for GPU memory
1167     float *d_VPotential;
1168     float *d_RhoChargeDensity;
1169     float *d_DeltaResidue;
1170     float *d_coef1;
1171     float *d_coef2;
1172     float *d_coef3;
1173     float *d_coef4;
1174     float *d_icoef4;
1175     float *d_VPotentialPrev;
1176     float *d_EpsilonError;
1177
1178
1179     // variables for coefficent calculations
1180     float *coef1;
1181     float *coef2;
1182     float *coef3;
1183     float *coef4;
1184     float *icoef4;
1185     float tempRatioZ;
1186     float tempRatioPhi;
1187     float radius;
1188
1189     int gridFrom;
1190     int gridTo;
1191     int loops;
1192
1193     // variables passed from ALIROOT
1194     float gridSizeR     = fparam[0];
1195     //float gridSizePhi = fparam[1];
1196     //float gridSizeZ       = fparam[2];
1197     float ratioPhi      = fparam[3];
1198     float ratioZ        = fparam[4];
1199     float convErr       = fparam[5];
1200     float IFCRadius     = fparam[6];
1201     int nPre    = iparam[0];
1202     int nPost   = iparam[1];
1203     int maxLoop = iparam[2];
1204     int nCycle  = iparam[3];
1205
1206     // variables for calculating GPU memory allocation
1207     int grid_RRow;
1208     int grid_ZColumn;
1209     int grid_PhiSlice = PhiSlice;
1210     int grid_Size = 0;
1211     int grid_StartPos;
1212     int coef_Size = 0;
1213     int coef_StartPos;
1214     int iOne, jOne;
1215     float h, h2, ih2;
1216
```

```
    // variables for calculating multigrid maximum depth
    int depth_RRow = 0;
    int depth_ZColumn = 0;
    int temp_RRow = RRow;
    int temp_ZColumn = ZColumn;

    // calculate depth for multigrid
    while (temp_RRow >>= 1) depth_RRow++;
    while (temp_ZColumn >>= 1) depth_ZColumn++;

    loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
    loops = (loops > maxLoop) ? maxLoop : loops;

    gridFrom = 1;
    gridTo = loops;

    // calculate GPU memory allocation for multigrid
    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
        grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;

        grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
        coef_Size += grid_RRow;
    }

    // allocate memory for temporary output
    temp_VPotential         = (float *) malloc(grid_Size * sizeof(float));
    VPotentialPrev = (float *) malloc(RRow * ZColumn * PhiSlice * sizeof(float));
    EpsilonError = (float *) malloc(1 * sizeof(float));


    // allocate memory for relaxation coefficient
    coef1 = (float *) malloc(coef_Size * sizeof(float));
    coef2 = (float *) malloc(coef_Size * sizeof(float));
    coef3 = (float *) malloc(coef_Size * sizeof(float));
    coef4 = (float *) malloc(coef_Size * sizeof(float));
    icoef4 = (float *) malloc(coef_Size * sizeof(float));

    // pre-compute relaxation coefficient
    coef_StartPos = 0;
    iOne = 1 << (gridFrom - 1);
    jOne = 1 << (gridFrom - 1);

    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / iOne) + 1;

        h = gridSizeR * iOne;
        h2 = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        for (int i = 1; i < grid_RRow - 1; i++)
        {
            radius = IFCRadius + i * h;
            coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
            coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
            coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
            coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
            icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
        }
        coef_StartPos += grid_RRow;
        iOne = 2 * iOne;
        jOne = 2 * jOne;
    }

    // device memory allocation
    cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
    cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
    cudaMalloc( &d_VPotentialPrev, RRow * ZColumn * PhiSlice * sizeof(float) );
    cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );

    cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
    cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
    cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );

    // set memory to zero
    cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
    cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
    cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
    cudaMemset( d_VPotentialPrev, 0, RRow * ZColumn * PhiSlice * sizeof(float) );
```

```
    cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );


    // copy data from host to device
    cudaMemcpy( d_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice ); //check
    cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice ); //check
    cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice );

    // max exact    float maxAbsExact = GetAbsMax(VPotentialExact,RRow * PhiSlice * ZColumn);
    float maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
    dim3 error_BlockPerGrid((RRow < 16) ? 1 : (RRow / 16), (ZColumn < 16) ? 1 : (ZColumn / 16), PhiSlice);
    dim3 error_ThreadPerBlock(16, 16);


    for (int cycle = 0; cycle < nCycle; cycle++)
    {
    /*V-Cycle starts*/

        // error conv
        //  cudaMemcpy( temp_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );

        cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
        errorExact[cycle] = GetErrorNorm2(temp_VPotential,VPotentialExact,RRow * PhiSlice,ZColumn, maxAbsExact);


        // V-Cycle => Finest Grid
        iOne = 1 << (gridFrom - 1);
        jOne = 1 << (gridFrom - 1);

        grid_RRow       = ((RRow - 1) / iOne) + 1;
        grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

        grid_StartPos = 0;
        coef_StartPos = 0;

        // pre-compute constant memory
        h   = gridSizeR * iOne;
        h2  = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        // copy constant to device memory
        cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

        // set kernel grid size and block size
        dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : ( grid_ZColumn / 16), PhiSlice);
        dim3 grid_ThreadPerBlock(16, 16);

        // red-black gauss seidel relaxation (nPre times)
        for (int i = 0; i < nPre; i++)
        {
            relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
            //cudaDeviceSynchronize();
            relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
            //cudaDeviceSynchronize();
        }

        // residue calculation
        residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );
        //cudaDeviceSynchronize();

        // V-Cycle => from finer to coarsest grid
        for (int step = gridFrom + 1; step <= gridTo; step++)
        {
            iOne = 1 << (step - 1);
            jOne = 1 << (step - 1);
```

```

            grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos += grid_RRow;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            // pre-compute constant memory
            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // restriction
            restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // zeroing V
            zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPre times)
            for (int i = 0; i < nPre; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
            }

            // residue calculation
            if (step < gridTo)
            {
                residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );
                //cudaDeviceSynchronize();

            }
        }

        // up one


        {
            int step = (gridTo - 1);
            iOne = iOne / 2;
            jOne = jOne / 2;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos -= grid_RRow;

            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
```

```
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

        // prolongation
            prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
//          cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPost times)
            for (int i = 0; i < nPost; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
            }
        }

        // down one
        {
            residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );

            iOne = iOne * 2;
            jOne = jOne * 2;

            grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos += grid_RRow;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            // pre-compute constant memory
            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;


            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // restriction
            restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // zeroing V
            zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPre times)
            for (int i = 0; i < nPre; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
            }

        }

        // up two from gridTo - 1, to gridTo -3
        for (int step = (gridTo - 1); step >= gridTo - 3; step--)
        {
```

```
            iOne = iOne / 2;
            jOne = jOne / 2;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos -= grid_RRow;

            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // prolongation
            prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
//          cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPost times)
            for (int i = 0; i < nPost; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
            }
        }

        // down to from gridTo - 1, to gridTo -3
        for (int step = gridTo - 3; step <= gridTo - 1; step++)
        {
            iOne = iOne * 2;
            jOne = jOne * 2;

            grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos += grid_RRow;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            // pre-compute constant memory
            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // restriction
            restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // zeroing V
```

```
            zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPre times)
            for (int i = 0; i < nPre; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
            }

            // residue calculation
            if (step < gridTo)
            {
                residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );
                //cudaDeviceSynchronize();

            }
        }



        {
            int step = (gridTo - 1);
            iOne = iOne / 2;
            jOne = jOne / 2;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos -= grid_RRow;

            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : ( grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

        // prolongation
            prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
//          cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPost times)
            for (int i = 0; i < nPost; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
            }
        }

        // down one
        {
            residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );

            iOne = iOne * 2;
            jOne = jOne * 2;

            grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos += grid_RRow;

```

```
            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            // pre-compute constant memory
            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;


            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // restriction
            restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // zeroing V
            zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPre times)
            for (int i = 0; i < nPre; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
            }

        }


        // V-Cycle => from coarser to finer grid
        for (int step = (gridTo - 1); step >= gridFrom; step--)
        {
            iOne = iOne / 2;
            jOne = jOne / 2;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos -= grid_RRow;

            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // prolongation
            prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
//          cudaDeviceSynchronize();
```

```

            // red-black gauss seidel relaxation (nPost times)
            for (int i = 0; i < nPost; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
            }
        }

    /*V-Cycle ends*/

        errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev, d_VPotential, d_EpsilonError, RRow, ZColumn, PhiSlice);

        cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );


        errorConv[cycle] = *EpsilonError  / (RRow * ZColumn * PhiSlice);

        if (errorConv[cycle] < convErr)
        {
            //errorConv
            nCycle = cycle;
            iparam[3] = nCycle;
            break;
        }

        cudaMemcpy( d_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );
        cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );



    }

    cudaDeviceSynchronize();
    // copy result from device to host
    cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );

    memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));

    // free device memory
    cudaFree( d_VPotential );
    cudaFree( d_VPotentialPrev );
    cudaFree( d_EpsilonError );


    cudaFree( d_DeltaResidue );
    cudaFree( d_RhoChargeDensity );
    cudaFree( d_coef1 );
    cudaFree( d_coef2 );
    cudaFree( d_coef3 );
    cudaFree( d_coef4 );
    cudaFree( d_icoef4 );

    // free host memory
    free( coef1 );
    free( coef2 );
    free( coef3 );
    free( coef4 );
    free( icoef4 );
    free( temp_VPotential );
    //free( temp_VPotentialPrev );
}
```

6.3.2.4 PrintMatrix()

```
void PrintMatrix (
            float * Mat,
            const int Row,
            const int Column )
```

Print matrix

Definisi pada baris 675 dalam file PoissonSolver3DGPU.cu.

```
{
    printf("Matrix (%d,%d)\n",Row,Column);
    for (int i=0;i<Row;i++)
    {
        for (int j=0;j<Column;j++)
        {
            printf("%11.4g ",Mat[i*Column + j]);
        }
        printf("\n");
    }

}
```

6.3.2.5 relaxationGaussSeidelBlack()

```
__global__ void relaxationGaussSeidelBlack (
            float * VPotential,
            float * RhoChargeDensity,
            const int RRow,
            const int ZColumn,
            const int PhiSlice,
            float * coef1,
            float * coef2,
            float * coef3,
            float * coef4 )
```

Relaksasi menggunakan penyelesaian iteratif Red-Black Gauss-Seidel (bagian Black)

Parameter

VPotential	float* Array potensial
RhoChargeDensity	float* Array rapat arus
RRow	int Jumlah baris di arah sumbu r
ZColumn	int Jumlah kolom di arah sumbu z
PhiSlice	int Jumlah irisan di arah sumbu ϕ
coef1	float* Array untuk koefisien $V_{x+1,y,z}$
coef2	float* Array untuk koefisien $V_{x-1,y,z}$
coef3	float* Array untuk koefisien z
coef4	float* Array untuk koefisien $f(r, \phi, z)$

Definisi pada baris 104 dalam file PoissonSolver3DGPU.cu.

```
{
    int index_x, index_y, index, index_left, index_right, index_up, index_down, index_front, index_back, index_coef;

    index_x = blockIdx.x * blockDim.x + threadIdx.x;
    index_y = blockIdx.y * blockDim.y + threadIdx.y;

    index       = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + index_x;
    index_left  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x - 1);
    index_right = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x + 1);
    index_up    = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y - 1) * ZColumn + index_x;
    index_down  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y + 1) * ZColumn + index_x;
    index_front = d_grid_StartPos + ((blockIdx.z - 1 + PhiSlice) % PhiSlice) * RRow * ZColumn + index_y * ZColumn + index_x;
    index_back  = d_grid_StartPos + ((blockIdx.z + 1) % PhiSlice) * RRow * ZColumn + index_y * ZColumn + index_x;
    index_coef  = d_coef_StartPos + index_y;

    if (index_x != 0 && index_x < (ZColumn - 1) && index_y != 0 && index_y < (RRow - 1))
    {
```

```
        //calculate black
        if ((blockIdx.z % 2 == 0 && (index_x + index_y) % 2 != 0) || (blockIdx.z % 2 != 0 && (index_x + index_y) % 2 == 0))
        {
            VPotential[index] = (coef2[index_coef] * VPotential[index_up] +
                                coef1[index_coef] * VPotential[index_down] +
                                d_tempRatioZ * (VPotential[index_left] + VPotential[index_right]) +
                                coef3[index_coef] * (VPotential[index_front] + VPotential[index_back]) +
                                d_h2 * RhoChargeDensity[index]) * coef4[index_coef];
        }
    }
}
```

6.3.2.6 relaxationGaussSeidelRed()

```
__global__ void relaxationGaussSeidelRed (
            float * VPotential,
            float * RhoChargeDensity,
            const int RRow,
            const int ZColumn,
            const int PhiSlice,
            float * coef1,
            float * coef2,
            float * coef3,
            float * coef4 )
```

Relaksasi menggunakan penyelesaian iteratif Red-Black Gauss-Seidel (bagian Red)

Parameter

VPotential	float∗ Array potensial
RhoChargeDensity	float∗ Array rapat arus
RRow	int Jumlah baris di arah sumbu r
ZColumn	int Jumlah kolom di arah sumbu z
PhiSlice	int Jumlah irisan di arah sumbu ϕ
coef1	float∗ Array untuk koefisien $V_{x+1,y,z}$
coef2	float∗ Array untuk koefisien $V_{x-1,y,z}$
coef3	float∗ Array untuk koefisien z
coef4	float∗ Array untuk koefisien $f(r, \phi, z)$

Definisi pada baris 52 dalam file PoissonSolver3DGPU.cu.

```
{
    int index_x, index_y, index, index_left, index_right, index_up, index_down, index_front, index_back, index_coef;

    index_x = blockIdx.x * blockDim.x + threadIdx.x;
    index_y = blockIdx.y * blockDim.y + threadIdx.y;

    index       = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + index_x;
    index_left  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x - 1);
    index_right = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x + 1);
    index_up    = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y - 1) * ZColumn + index_x;
    index_down  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y + 1) * ZColumn + index_x;
    index_front = d_grid_StartPos + ((blockIdx.z - 1 + PhiSlice) % PhiSlice) * RRow * ZColumn + index_y * ZColumn + index_x;
    index_back  = d_grid_StartPos + ((blockIdx.z + 1) % PhiSlice) * RRow * ZColumn + index_y * ZColumn + index_x;
    index_coef  = d_coef_StartPos + index_y;

    if (index_x != 0 && index_x < (ZColumn - 1) && index_y != 0 && index_y < (RRow - 1))
    {
```

```
        //calculate red
        if ((blockIdx.z % 2 == 0 && (index_x + index_y) % 2 == 0) || (blockIdx.z % 2 != 0 && (index_x + index_y) % 2 != 0))
        {
            VPotential[index] = (coef2[index_coef] * VPotential[index_up] +
                                 coef1[index_coef] * VPotential[index_down] +
                                 d_tempRatioZ * (VPotential[index_left] + VPotential[index_right]) +
                                 coef3[index_coef] * (VPotential[index_front] + VPotential[index_back]) +
                                 d_h2 * RhoChargeDensity[index]) * coef4[index_coef];
        }
    }
}
```

6.3.2.7 residueCalculation()

```
__global__ void residueCalculation (
        float * VPotential,
        float * RhoChargeDensity,
        float * DeltaResidue,
        const int RRow,
        const int ZColumn,
        const int PhiSlice,
        float * coef1,
        float * coef2,
        float * coef3,
        float * icoef4 )
```

Menghitung residu dari hasil proses relaksasi

Rumus:

Parameter

VPotential	float∗ Array potensial
RhoChargeDensity	float∗ Array rapat arus
DeltaResidue	float∗ Array residu
RRow	int Jumlah baris di arah sumbu r
ZColumn	int Jumlah kolom di arah sumbu z
PhiSlice	int Jumlah irisan di arah sumbu ϕ
coef1	float∗ Array untuk koefisien $V_{x+1,y,z}$
coef2	float∗ Array untuk koefisien $V_{x-1,y,z}$
coef3	float∗ Array untuk koefisien z
icoef4	float∗ Array untuk koefisien invers dari $f(r,\phi,z)$

Definisi pada baris 160 dalam file PoissonSolver3DGPU.cu.

```
{
    int index_x, index_y, index, index_left, index_right, index_up, index_down, index_front, index_back, index_coef;

    index_x = blockIdx.x * blockDim.x + threadIdx.x;
    index_y = blockIdx.y * blockDim.y + threadIdx.y;

    index       = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + index_x;
    index_left  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x - 1);
    index_right = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x + 1);
    index_up    = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y - 1) * ZColumn + index_x;
    index_down  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y + 1) * ZColumn + index_x;
    index_front = d_grid_StartPos + ((blockIdx.z - 1 + PhiSlice) % PhiSlice)  * RRow * ZColumn + index_y * ZColumn + index_x;
```

```
    index_back  = d_grid_StartPos + ((blockIdx.z + 1) % PhiSlice)  * RRow * ZColumn + index_y * ZColumn +
  index_x;
    index_coef  = d_coef_StartPos + index_y;

    if (index_x != 0 && index_x < (ZColumn - 1) && index_y != 0 && index_y < (RRow - 1))
    {
        DeltaResidue[index] = d_ih2 * (coef2[index_coef] * VPotential[index_up] +
                        coef1[index_coef] * VPotential[index_down] +
                        d_tempRatioZ * (VPotential[index_left] + VPotential[index_right]) +
                        coef3[index_coef] * (VPotential[index_front] + VPotential[index_back]) -
                        icoef4[index_coef] * VPotential[index]) + RhoChargeDensity[index];
    }
}
```

6.3.2.8 Restrict_Boundary()

```
void Restrict_Boundary (
            float * VPotential,
            const int RRow,
            const int ZColumn,
            const int PhiSlice,
            const int Offset )
```

top left (0,0)

boundary in top and down for (j = 1, jj =2; j < ZColumn-1 ; j++,jj+=2) { VPotential[Offset + sliceStart + (0 ∗ ZColumn) + j] = 0.5 ∗ VPotential[finer_Offset + finer_SliceStart + (0 ∗ finer_ZColumn) + jj] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + (0 ∗ finer_ZColumn) + jj - 1] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + (0 ∗ finer_Z←Column) + jj + 1];

VPotential[Offset + sliceStart + ((RRow - 1) ∗ ZColumn) + j] = 0.5 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow -1) ∗ finer_ZColumn) + jj] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow -1) ∗ finer_ZColumn) + jj - 1] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow -1) ∗ finer_ZColumn) + jj + 1];

```
    }
```

boundary in left and right for (i = 1, ii =2; i < RRow - 1 ; i++,ii+=2) { VPotential[Offset + sliceStart + (i ∗ ZColumn)] = 0.5 ∗ VPotential[finer_Offset + finer_SliceStart + (ii ∗ finer_ZColumn)] + 0.25 ∗ VPotential[finer_Offset + finer_←SliceStart + ((ii-1) ∗ finer_ZColumn)] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((ii + 1) ∗ finer_ZColumn)];

VPotential[Offset + sliceStart + (i ∗ ZColumn) + (ZColumn - 1)] = 0.5 ∗ VPotential[finer_Offset + finer_SliceStart + (ii ∗ finer_ZColumn) + jj + (finer_ZColumn - 1)] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((ii -1) ∗ finer←_ZColumn) + (finer_ZColumn - 1)] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((ii +1) ∗ finer_ZColumn) + (finer_ZColumn - 1)];

}

top left (0,0)

```
    VPotential[Offset + sliceStart + (0 * ZColumn) + 0] =
        0.5 * VPotential[finer_Offset  + finer_SliceStart] +
        0.25 * VPotential[finer_Offset + finer_SliceStart + (0 * finer_ZColumn) + 1] +
        0.25 * VPotential[finer_Offset + finer_SliceStart + (1 * finer_ZColumn)];
```

top right VPotential[Offset + sliceStart + (0 ∗ ZColumn) + (ZColumn - 1)] = 0.5 ∗ VPotential[finer_Offset + finer_SliceStart + (0 ∗ finer_ZColumn) + (finer_ZColumn -1)] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + (0 ∗ finer_ZColumn) + (finer_ZColumn - 2)] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + (1 ∗ finer_ZColumn) + (finer_ZColumn - 1)];

bottom left VPotential[Offset + sliceStart + ((RRow - 1) ∗ ZColumn) + 0] = 0.5 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow - 1) ∗ finer_ZColumn) + 0] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow - 1) ∗ finer_ZColumn) + 1] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow - 2) ∗ finer_ZColumn) + 0];

bottom right VPotential[Offset + sliceStart + ((RRow - 1) ∗ ZColumn) + (ZColumn - 1)] = 0.5 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow - 1) ∗ finer_ZColumn) + (finer_ZColumn - 1)] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow - 1) ∗ finer_ZColumn) + (finer_ZColumn - 2)] + 0.25 ∗ VPotential[finer_Offset + finer_SliceStart + ((finer_RRow - 2) ∗ finer_ZColumn) + (finer_ZColumn - 1)];

}

Definisi pada baris 566 dalam file PoissonSolver3DGPU.cu.

```
{
    int i,ii,j,jj;
    int finer_RRow = 2 * RRow - 1;
    int finer_ZColumn = 2 * ZColumn - 1;

    int finer_Offset = Offset - (finer_RRow * finer_ZColumn * PhiSlice);
    int sliceStart;
    int finer_SliceStart;

    //printf("(%d,%d,%d) -> (%d,%d,%d)\n",RRow,ZColumn,Offset,finer_RRow,finer_ZColumn,finer_Offset);
    // do for each slice
    for ( int m = 0;m < PhiSlice;m++)
    {
        sliceStart = m * (RRow * ZColumn);
        finer_SliceStart = m * (finer_RRow * finer_ZColumn);
        // copy boundary
        for ( j = 0, jj =0; j < ZColumn ; j++,jj+=2)
        {
            VPotential[Offset + sliceStart + (0 * ZColumn) + j] =
                VPotential[finer_Offset + finer_SliceStart + (0 * finer_ZColumn) + jj];

            VPotential[Offset + sliceStart + ((RRow - 1) * ZColumn) + j] =
                VPotential[finer_Offset + finer_SliceStart + ((finer_RRow -1) * finer_ZColumn) + jj];

        }
        for ( i = 0, ii =0; i < RRow  ; i++,ii+=2) {
            VPotential[Offset + sliceStart + (i * ZColumn)] =
                VPotential[finer_Offset + finer_SliceStart + (ii * finer_ZColumn)];

            VPotential[Offset + sliceStart + (i * ZColumn) + (ZColumn - 1)] =
                VPotential[finer_Offset + finer_SliceStart + (ii * finer_ZColumn) + (finer_ZColumn - 1)];

        }
    }
}
```

6.3.2.9 restriction2DFull()

```
__global__ void restriction2DFull (
            float * RhoChargeDensity,
            float * DeltaResidue,
            const int RRow,
            const int ZColumn,
            const int PhiSlice )
```

Restriksi dari finer grid ke coarser grid dengan operator Half Weighting

$I_h^{2h} = \frac{1}{8}[ccc]010$
141
$010 RhoChargeDensity float* Arrayrapatarus DeltaResidue float* Arrayresiduhasilrelaksasi RRow constint Jumlahbar$
r $ZColumn constint Jumlahkolomdiarahsumbu$ z $PhiSlice constint Jumlahirisandiarahsumbu$ $_{global_{voidrestriction2DHalf(floa}}$
I_h^{2h} = {1}{16} {bmatrix}[ccc] 1 & 2 & 1 \ 2 & 4 & 2\ 1 & 2 & 1 {bmatrix}

Parameter

RhoChargeDensity	float* Array rapat arus
DeltaResidue	float* Array residu hasil relaksasi
RRow	const int Jumlah baris di arah sumbu r
ZColumn	const int Jumlah kolom di arah sumbu z
PhiSlice	const int Jumlah irisan di arah sumbu ϕ

Definisi pada baris 253 dalam file PoissonSolver3DGPU.cu.

```
{
    int index_x, index_y, index;
    int finer_RRow, finer_ZColumn, finer_grid_StartPos;
    int finer_index_x, finer_index_y, finer_index, finer_index_left, finer_index_right, finer_index_up, finer_index_down;
    int finer_index_up_left, finer_index_up_right, finer_index_down_left, finer_index_down_right;

    index_x = blockIdx.x * blockDim.x + threadIdx.x;
    index_y = blockIdx.y * blockDim.y + threadIdx.y;
    index   = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + index_x;

    finer_RRow = 2 * RRow - 1;
    finer_ZColumn = 2 * ZColumn - 1;

    finer_grid_StartPos = d_grid_StartPos - (finer_RRow * finer_ZColumn * PhiSlice);

    finer_index_x = index_x * 2;
    finer_index_y = index_y * 2;

    finer_index          = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + finer_index_y * finer_ZColumn + finer_index_x;
    finer_index_left     = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + finer_index_y * finer_ZColumn + (finer_index_x - 1);
    finer_index_right    = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + finer_index_y * finer_ZColumn + (finer_index_x + 1);
    finer_index_up       = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (finer_index_y - 1) * finer_ZColumn + finer_index_x;
    finer_index_down     = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (finer_index_y + 1) * finer_ZColumn + finer_index_x;
    finer_index_up_left      = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (finer_index_y - 1) * finer_ZColumn + (finer_index_x - 1);
    finer_index_up_right     = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (finer_index_y - 1) * finer_ZColumn + (finer_index_x + 1);
    finer_index_down_left    = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (finer_index_y + 1) * finer_ZColumn + (finer_index_x - 1);
    finer_index_down_right   = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (finer_index_y + 1) * finer_ZColumn + (finer_index_x + 1);

    if (index_x != 0 && index_x < (ZColumn - 1) && index_y != 0 && index_y < (RRow - 1))
    {
        RhoChargeDensity[index] = 0.25 * DeltaResidue[finer_index] +
                                  0.125 * (DeltaResidue[finer_index_left] + DeltaResidue[finer_index_right] + DeltaResidue[finer_index_up] + DeltaResidue[finer_index_down]) +
                                  0.0625 * (DeltaResidue[finer_index_up_left] + DeltaResidue[finer_index_up_right] + DeltaResidue[finer_index_down_left] + DeltaResidue[finer_index_down_right]);
    } else {
        RhoChargeDensity[index] =  DeltaResidue[finer_index];
    }


}
```

6.4 Referensi File /home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/kernel/PoissonSolver3DGPU.h

Berkas ini berisi definisi fungsi extern yang dimiliki implementasi CUDA yang didapat dipanggil CPU.

```
#include <ctime>
#include <iomanip>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
```

Fungsi

- void PoissonMultigrid3DSemiCoarseningGPUError (float ∗VPotential, float ∗RhoChargeDensity, const int R←↩ Row, const int ZColumn, const int PhiSlice, const int Symmetry, float ∗fparam, int ∗iparam, bool isExact←↩ Present, float ∗errorConv, float ∗errorExact, float ∗VPotentialExact)
- void PoissonMultigrid3DSemiCoarseningGPUErrorWCycle (float ∗VPotential, float ∗RhoChargeDensity, const int RRow, const int ZColumn, const int PhiSlice, const int Symmetry, float ∗fparam, int ∗iparam, float ∗errorConv, float ∗errorExact, float ∗VPotentialExact)
- void PoissonMultigrid3DSemiCoarseningGPUErrorFCycle (float ∗VPotential, float ∗RhoChargeDensity, const int RRow, const int ZColumn, const int PhiSlice, const int Symmetry, float ∗fparam, int ∗iparam, bool is←↩ ExactPresent, float ∗errorConv, float ∗errorExact, float ∗VPotentialExact)

6.4.1 Keterangan Lengkap

Berkas ini berisi definisi fungsi extern yang dimiliki implementasi CUDA yang didapat dipanggil CPU.

Penulis

Rifki Sadikin rifki.sadikin@lipi.go.id, Pusat Penelitian Informatika, Lembaga Ilmu Pengetahuan Indonesia
I Wayan Aditya Swardiana i.wayan.aditya.swardiana@lipi.go.id, Pusat Penelitian Informatika, Lembaga Ilmu Pengetahuan Indonesia

Tanggal

November 8, 2018

6.4.2 Dokumentasi Fungsi

6.4.2.1 PoissonMultigrid3DSemiCoarseningGPUError()

```
void PoissonMultigrid3DSemiCoarseningGPUError (
            float * VPotential,
            float * RhoChargeDensity,
            const int RRow,
            const int ZColumn,
            const int PhiSlice,
            const int Symmetry,
            float * fparam,
            int * iparam,
            bool isExactPresent,
            float * errorConv,
            float * errorExact,
            float * VPotentialExact )
```

Fungsi ini menghitung solusi terhadap Persamaan Poisson

$$\nabla^2(r,\phi,z) = \rho(r,\phi,z)$$

dengan diketahui nilai tepi (Boundary Value) pada V (potensial) dan distribusi buatan ρ

Parameter

in,out	*VPotential*	float[nrows∗ncols∗nphi] distribusi potensial. Input: hanya nilai tepi. Output: hasil perhitungan penyelesaian persamaan Poisson
in	*RhoChangeDensity*	float[nrows∗ncols∗nphi] distributsi muatan listrik.

Mengembalikan

A fixed number that has nothing to do with what the function does

Definisi pada baris 893 dalam file PoissonSolver3DGPU.cu.

```
{
    // variables for CPU memory
    float *temp_VPotential;
    float *VPotentialPrev;
    float *EpsilonError;

    // variables for GPU memory
    float *d_VPotential;
    float *d_RhoChargeDensity;
    float *d_DeltaResidue;
    float *d_VPotentialPrev;
    float *d_EpsilonError;

    float *d_coef1;
    float *d_coef2;
    float *d_coef3;
    float *d_coef4;
    float *d_icoef4;

    // variables for coefficent calculations
    float *coef1;
    float *coef2;
    float *coef3;
    float *coef4;
    float *icoef4;
    float tempRatioZ;
    float tempRatioPhi;
    float radius;

    int gridFrom;
    int gridTo;
    int loops;


    // variables passed from ALIROOT
    float gridSizeR     = fparam[0];
    float gridSizePhi   = fparam[1];
    float gridSizeZ     = fparam[2];
    float ratioPhi      = fparam[3];
    float ratioZ        = fparam[4];
    float convErr       = fparam[5];
    float IFCRadius     = fparam[6];
    int nPre        = iparam[0];
    int nPost       = iparam[1];
    int maxLoop     = iparam[2];
    int nCycle      = iparam[3];

    // variables for calculating GPU memory allocation
    int grid_RRow;
    int grid_ZColumn;
    int grid_PhiSlice = PhiSlice;
    int grid_Size = 0;
    int grid_StartPos;
    int coef_Size = 0;
    int coef_StartPos;
    int iOne, jOne;
    float h, h2, ih2;

    // variables for calculating multigrid maximum depth
    int depth_RRow = 0;
    int depth_ZColumn = 0;
    int temp_RRow = RRow;
    int temp_ZColumn = ZColumn;

    // calculate depth for multigrid
```

```
    while (temp_RRow >>= 1) depth_RRow++;
    while (temp_ZColumn >>= 1) depth_ZColumn++;

    loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
    loops = (loops > maxLoop) ? maxLoop : loops;

    gridFrom = 1;
    gridTo = loops;

    // calculate GPU memory allocation for multigrid
    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
        grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;

        grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
        coef_Size += grid_RRow;
    }

    // allocate memory for temporary output
    temp_VPotential         = (float *) malloc(grid_Size * sizeof(float));
    VPotentialPrev = (float *) malloc(RRow * ZColumn * PhiSlice * sizeof(float));
    EpsilonError = (float *) malloc(1 * sizeof(float));


    // allocate memory for relaxation coefficient
    coef1 = (float *) malloc(coef_Size * sizeof(float));
    coef2 = (float *) malloc(coef_Size * sizeof(float));
    coef3 = (float *) malloc(coef_Size * sizeof(float));
    coef4 = (float *) malloc(coef_Size * sizeof(float));
    icoef4 = (float *) malloc(coef_Size * sizeof(float));

    // pre-compute relaxation coefficient
    coef_StartPos = 0;
    iOne = 1 << (gridFrom - 1);
    jOne = 1 << (gridFrom - 1);

    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / iOne) + 1;

        h = gridSizeR * iOne;
        h2 = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        for (int i = 1; i < grid_RRow - 1; i++)
        {
            radius = IFCRadius + i * h;
            coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
            coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
            coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
            coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
            icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
        }
        coef_StartPos += grid_RRow;
        iOne = 2 * iOne;
        jOne = 2 * jOne;
    }

    // device memory allocation
    cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
    cudaMalloc( &d_VPotentialPrev, RRow * ZColumn * PhiSlice * sizeof(float) );
    cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );
    cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
    cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
    cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
    cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );

    // set memory to zero
    cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
    cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
    cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
    cudaMemset( d_VPotentialPrev, 0, RRow * ZColumn * PhiSlice * sizeof(float) );
    cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );


    // copy data from host to device
    cudaMemcpy( d_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice
 ); //check
    cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyHostToDevice ); //check
```

```
    cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
 cudaMemcpyHostToDevice );

    // max exact

    // float maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
    float maxAbsExact = 1.0;

    if (isExactPresent == true)
        maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
    dim3 error_BlockPerGrid((RRow < 16) ? 1 : (RRow / 16), (ZColumn < 16) ? 1 : (ZColumn / 16), PhiSlice);
    dim3 error_ThreadPerBlock(16, 16);


    for (int cycle = 0; cycle < nCycle; cycle++)
    {
        cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
 cudaMemcpyDeviceToHost );
        if (isExactPresent == true) errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact,
 RRow * PhiSlice,ZColumn, maxAbsExact);


        VCycleSemiCoarseningGPU(d_VPotential, d_RhoChargeDensity, d_DeltaResidue, d_coef1, d_coef2, d_coef3
 , d_coef4, d_icoef4, gridSizeR, ratioZ, ratioPhi, RRow, ZColumn, PhiSlice, gridFrom, gridTo, nPre, nPost);


        errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev, d_VPotential,
 d_EpsilonError, RRow, ZColumn, PhiSlice);

        cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );


        errorConv[cycle] = *EpsilonError  / (RRow * ZColumn * PhiSlice);

        if (errorConv[cycle] < convErr)
        {
            //errorConv
            nCycle = cycle;
            break;
        }

        cudaMemcpy( d_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
 cudaMemcpyDeviceToDevice );
        cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );

    }
    iparam[3] = nCycle;

//  for (int cycle = 0; cycle < nCycle; cycle++)
//  {
//      cudaMemcpy( temp_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
  cudaMemcpyDeviceToHost );


//      VCycleSemiCoarseningGPU(d_VPotential, d_RhoChargeDensity, d_DeltaResidue, d_coef1, d_coef2,
  d_coef3, d_coef4, d_icoef4, gridSizeR, ratioZ, ratioPhi, RRow, ZColumn, PhiSlice, gridFrom, gridTo, nPre, nPost);

//      cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
  cudaMemcpyDeviceToHost );
//      errorConv[cycle] = GetErrorNorm2(temp_VPotential, temp_VPotentialPrev, RRow * PhiSlice, ZColumn,
  1.0);
//      //errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact, RRow * PhiSlice,ZColumn,
  1.0);
//  }


    // copy result from device to host
    cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
 cudaMemcpyDeviceToHost );

    memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));

    // free device memory
    cudaFree( d_VPotential );
    cudaFree( d_DeltaResidue );
    cudaFree( d_RhoChargeDensity );
    cudaFree( d_VPotentialPrev );
    cudaFree( d_EpsilonError );
    cudaFree( d_coef1 );
    cudaFree( d_coef2 );
    cudaFree( d_coef3 );
    cudaFree( d_coef4 );
```

```
    cudaFree( d_icoef4 );

    // free host memory
    free( coef1 );
    free( coef2 );
    free( coef3 );
    free( coef4 );
    free( icoef4 );
    free( temp_VPotential );
    free( VPotentialPrev );
}
```

6.4.2.2 PoissonMultigrid3DSemiCoarseningGPUErrorFCycle()

```
void PoissonMultigrid3DSemiCoarseningGPUErrorFCycle (
            float * VPotential,
            float * RhoChargeDensity,
            const int RRow,
            const int ZColumn,
            const int PhiSlice,
            const int Symmetry,
            float * fparam,
            int * iparam,
            bool isExactPresent,
            float * errorConv,
            float * errorExact,
            float * VPotentialExact )
```

cudaMemcpy(temp_VPotential, d_RhoChargeDensity + grid_StartPos , grid_RRow ∗ grid_ZColumn ∗ PhiSlice ∗ sizeof(float), cudaMemcpyDeviceToHost);

Definisi pada baris 1837 dalam file PoissonSolver3DGPU.cu.

```
{
    // variables for CPU memory
    float *temp_VPotential;
    float *VPotentialPrev;
    float *EpsilonError;

    // variables for GPU memory
    float *d_VPotential;
    float *d_RhoChargeDensity;
    float *d_DeltaResidue;
    float *d_coef1;
    float *d_coef2;
    float *d_coef3;
    float *d_coef4;
    float *d_icoef4;
    float *d_VPotentialPrev;
    float *d_EpsilonError;


    // variables for coefficent calculations
    float *coef1;
    float *coef2;
    float *coef3;
    float *coef4;
    float *icoef4;
    float tempRatioZ;
    float tempRatioPhi;
    float radius;

    int gridFrom;
    int gridTo;
    int loops;

    // variables passed from ALIROOT
    float gridSizeR     = fparam[0];
    //float gridSizePhi = fparam[1];
    //float gridSizeZ       = fparam[2];
    float ratioPhi      = fparam[3];
```

```
    float ratioZ        = fparam[4];
    float convErr       = fparam[5];
    float IFCRadius     = fparam[6];
    int nPre    = iparam[0];
    int nPost   = iparam[1];
    int maxLoop = iparam[2];
    int nCycle  = iparam[3];

    // variables for calculating GPU memory allocation
    int grid_RRow;
    int grid_ZColumn;
    int grid_PhiSlice = PhiSlice;
    int grid_Size = 0;
    int grid_StartPos;
    int coef_Size = 0;
    int coef_StartPos;
    int iOne, jOne;
    float h, h2, ih2;

    // variables for calculating multigrid maximum depth
    int depth_RRow = 0;
    int depth_ZColumn = 0;
    int temp_RRow = RRow;
    int temp_ZColumn = ZColumn;

    // calculate depth for multigrid
    while (temp_RRow >>= 1) depth_RRow++;
    while (temp_ZColumn >>= 1) depth_ZColumn++;

    loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
    loops = (loops > maxLoop) ? maxLoop : loops;

    gridFrom = 1;
    gridTo = loops;

    // calculate GPU memory allocation for multigrid
    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
        grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;

        grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
        coef_Size += grid_RRow;
    }

    // allocate memory for temporary output
    temp_VPotential         = (float *) malloc(grid_Size * sizeof(float));
    VPotentialPrev = (float *) malloc(grid_Size * sizeof(float));
    EpsilonError = (float *) malloc(1 * sizeof(float));



    for (int i=0;i<grid_Size;i++) temp_VPotential[i] = 0.0;


    // allocate memory for relaxation coefficient
    coef1 = (float *) malloc(coef_Size * sizeof(float));
    coef2 = (float *) malloc(coef_Size * sizeof(float));
    coef3 = (float *) malloc(coef_Size * sizeof(float));
    coef4 = (float *) malloc(coef_Size * sizeof(float));
    icoef4 = (float *) malloc(coef_Size * sizeof(float));

    // pre-compute relaxation coefficient
    // restrict boundary
    coef_StartPos = 0;
    grid_StartPos = 0;

    iOne = 1 << (gridFrom - 1);
    jOne = 1 << (gridFrom - 1);

    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / iOne) + 1;
        grid_ZColumn = ((ZColumn - 1) / iOne) + 1;

        h = gridSizeR * iOne;
        h2 = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        for (int i = 1; i < grid_RRow - 1; i++)
        {
            radius = IFCRadius + i * h;
            coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
            coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
```

```
            coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
            coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
            icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
        }

        // call restrict boundary
        if (step == gridFrom) {
            // Copy original VPotential to tempPotential
            memcpy(temp_VPotential,     VPotential, RRow * ZColumn * PhiSlice * sizeof(float));

        }
        // else
        //{
        //  Restrict_Boundary(temp_VPotential, grid_RRow, grid_ZColumn, PhiSlice, grid_StartPos);
        //}


        coef_StartPos += grid_RRow;
        grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;


        iOne = 2 * iOne;
        jOne = 2 * jOne;
    }

    // device memory allocation
    cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
    cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
    cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
    cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
    cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );
    cudaMalloc( &d_VPotentialPrev, grid_Size * sizeof(float) );
    cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );


    // set memory to zero
    cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
    cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
    cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
    cudaMemset( d_VPotentialPrev, 0, grid_Size * sizeof(float) );
    cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );

    // set memory to zero
    cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
    cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
    cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );

    // copy data from host to devicei
    // case of FCycle you need to copy all boundary for all
    cudaMemcpy( d_VPotential, temp_VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice ); //check
//  cudaMemcpy( d_VPotential, VPotential, grid_Size * isizeof(float), cudaMemcpyHostToDevice ); //check

    cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float),
  cudaMemcpyHostToDevice ); //check
//  cudaMemcpy( d_RhoChargeDensity, temp_VPotentialPrev, grid_Size * sizeof(float), cudaMemcpyHostToDevice
   ); //check
    cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
//  cudaMemcpy( d_VPotentialPrev, temp_VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice );

//  cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
   cudaMemcpyHostToDevice );

    // max exact

    float maxAbsExact = 1.0;

    if (isExactPresent == true)
        maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);



    // init iOne,grid_RRow, grid_ZColumn, grid_StartPos, coef_StartPos
    iOne = 1 << (gridFrom - 1);
    jOne = 1 << (gridFrom - 1);

    grid_RRow       = ((RRow - 1) / iOne) + 1;
    grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

    grid_StartPos = 0;
    coef_StartPos = 0;
```

```


    for (int step = gridFrom + 1; step <= gridTo; step++)
    {

        iOne = 1 << (step - 1);
        jOne = 1 << (step - 1);

        grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
        coef_StartPos += grid_RRow;

        grid_RRow       = ((RRow - 1) / iOne) + 1;
        grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

        // pre-compute constant memory
        h   = gridSizeR * iOne;
        h2  = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        // copy constant to device memory
        cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

        // set kernel grid size and block size
        dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
    grid_ZColumn / 16), PhiSlice);
        dim3 grid_ThreadPerBlock(16, 16);

        // restriction
        restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
    d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice );

        // restrict boundary (already done in cpu)
//      PrintMatrix(temp_VPotential,grid_RRow * PhiSlice,grid_ZColumn);
        restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_VPotential,
    grid_RRow, grid_ZColumn, grid_PhiSlice );


    }

    dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn
    / 16), PhiSlice);
    dim3 grid_ThreadPerBlock(16, 16);


    // relax on the coarsest
    // red-black gauss seidel relaxation (nPre times)
//  printf("rho\n");
//  cudaMemcpy( temp_VPotential, d_RhoChargeDensity + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
    sizeof(float), cudaMemcpyDeviceToHost );
//  PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);

//  printf("v\n");
//  cudaMemcpy( temp_VPotential, d_VPotential + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
    sizeof(float), cudaMemcpyDeviceToHost );
//  PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);
    for (int i = 0; i < nPre; i++)
    {
        relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
    d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
        cudaDeviceSynchronize();
        relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
    d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
        cudaDeviceSynchronize();
    }

//  printf("v after relax\n");
//  cudaMemcpy( temp_VPotential, d_VPotential + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
    sizeof(float), cudaMemcpyDeviceToHost );
//  PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);

    // V-Cycle => from coarser to finer grid
    for (int step = gridTo -1 ; step >= gridFrom; step--)
    {
        iOne = iOne / 2;
        jOne = jOne / 2;

        grid_RRow       = ((RRow - 1) / iOne) + 1;
        grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

        grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
```

```
        coef_StartPos -= grid_RRow;

        h   = gridSizeR * iOne;
        h2  = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        // copy constant to device memory
        cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );



        // set kernel grid size and block size
        dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
        dim3 grid_ThreadPerBlock(16, 16);


        prolongation2DHalfNoAdd<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );




        // just

        // max exact
        cudaMemcpy( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, grid_RRow * grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );

        float maxAbsExact = 1.0;

        if (isExactPresent == true)
            maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
        dim3 error_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
        dim3 error_ThreadPerBlock(16, 16);



        for (int cycle = 0; cycle < nCycle; cycle++)
        {


            if (step == gridFrom) {
                cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
                if (isExactPresent == true )errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact, RRow * PhiSlice,ZColumn, maxAbsExact);
            }



            //cudaDeviceSynchronize();
            VCycleSemiCoarseningGPU(d_VPotential, d_RhoChargeDensity, d_DeltaResidue, d_coef1, d_coef2, d_coef3, d_coef4, d_icoef4, gridSizeR, ratioZ, ratioPhi, RRow, ZColumn, PhiSlice, step, gridTo, nPre, nPost);



                //if (step == gridFrom) {
                //cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );

                //errorConv[cycle] = GetErrorNorm2(temp_VPotential, VPotentialPrev, RRow * PhiSlice,ZColumn, 1.0);

                errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, d_EpsilonError, grid_RRow, grid_ZColumn, PhiSlice);

                cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );

                errorConv[cycle] = *EpsilonError  / (grid_RRow * grid_ZColumn * PhiSlice);

                if (errorConv[cycle]< convErr)
                {
                    nCycle = cycle;
                    break;
                }

                cudaMemcpy( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, grid_RRow * grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );
```

```
                cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );

        }


    }

    iparam[3] = nCycle;

    // copy result from device to host
    cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
      cudaMemcpyDeviceToHost );

    memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));

    // free device memory
    cudaFree( d_VPotential );
    cudaFree( d_DeltaResidue );
    cudaFree( d_RhoChargeDensity );
    cudaFree( d_coef1 );
    cudaFree( d_coef2 );
    cudaFree( d_coef3 );
    cudaFree( d_coef4 );
    cudaFree( d_icoef4 );

    // free host memory
    free( coef1 );
    free( coef2 );
    free( coef3 );
    free( coef4 );
    free( icoef4 );
    free( temp_VPotential );
    free( VPotentialPrev );
}
```

6.4.2.3 PoissonMultigrid3DSemiCoarseningGPUErrorWCycle()

```
void PoissonMultigrid3DSemiCoarseningGPUErrorWCycle (
            float * VPotential,
            float * RhoChargeDensity,
            const int RRow,
            const int ZColumn,
            const int PhiSlice,
            const int Symmetry,
            float * fparam,
            int * iparam,
            float * errorConv,
            float * errorExact,
            float * VPotentialExact )
```

innner w cycle up one down one

end up one down on

up two down two

up one down one

end up one down one

Definisi pada baris 1147 dalam file PoissonSolver3DGPU.cu.

```
{
    // variables for CPU memory
    float *temp_VPotential;
    float *VPotentialPrev;
    float *EpsilonError;

    // variables for GPU memory
```

```
    float *d_VPotential;
    float *d_RhoChargeDensity;
    float *d_DeltaResidue;
    float *d_coef1;
    float *d_coef2;
    float *d_coef3;
    float *d_coef4;
    float *d_icoef4;
    float *d_VPotentialPrev;
    float *d_EpsilonError;


    // variables for coefficent calculations
    float *coef1;
    float *coef2;
    float *coef3;
    float *coef4;
    float *icoef4;
    float tempRatioZ;
    float tempRatioPhi;
    float radius;

    int gridFrom;
    int gridTo;
    int loops;

    // variables passed from ALIROOT
    float gridSizeR     = fparam[0];
    //float gridSizePhi = fparam[1];
    //float gridSizeZ       = fparam[2];
    float ratioPhi      = fparam[3];
    float ratioZ        = fparam[4];
    float convErr       = fparam[5];
    float IFCRadius     = fparam[6];
    int nPre    = iparam[0];
    int nPost   = iparam[1];
    int maxLoop = iparam[2];
    int nCycle  = iparam[3];

    // variables for calculating GPU memory allocation
    int grid_RRow;
    int grid_ZColumn;
    int grid_PhiSlice = PhiSlice;
    int grid_Size = 0;
    int grid_StartPos;
    int coef_Size = 0;
    int coef_StartPos;
    int iOne, jOne;
    float h, h2, ih2;

    // variables for calculating multigrid maximum depth
    int depth_RRow = 0;
    int depth_ZColumn = 0;
    int temp_RRow = RRow;
    int temp_ZColumn = ZColumn;

    // calculate depth for multigrid
    while (temp_RRow >>= 1) depth_RRow++;
    while (temp_ZColumn >>= 1) depth_ZColumn++;

    loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
    loops = (loops > maxLoop) ? maxLoop : loops;

    gridFrom = 1;
    gridTo = loops;

    // calculate GPU memory allocation for multigrid
    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
        grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;

        grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
        coef_Size += grid_RRow;
    }

    // allocate memory for temporary output
    temp_VPotential         = (float *) malloc(grid_Size * sizeof(float));
    VPotentialPrev = (float *) malloc(RRow * ZColumn * PhiSlice * sizeof(float));
    EpsilonError = (float *) malloc(1 * sizeof(float));


    // allocate memory for relaxation coefficient
    coef1 = (float *) malloc(coef_Size * sizeof(float));
    coef2 = (float *) malloc(coef_Size * sizeof(float));
    coef3 = (float *) malloc(coef_Size * sizeof(float));
    coef4 = (float *) malloc(coef_Size * sizeof(float));
```

```
    icoef4 = (float *) malloc(coef_Size * sizeof(float));

    // pre-compute relaxation coefficient
    coef_StartPos = 0;
    iOne = 1 << (gridFrom - 1);
    jOne = 1 << (gridFrom - 1);

    for (int step = gridFrom; step <= gridTo; step++)
    {
        grid_RRow = ((RRow - 1) / iOne) + 1;

        h = gridSizeR * iOne;
        h2 = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        for (int i = 1; i < grid_RRow - 1; i++)
        {
            radius = IFCRadius + i * h;
            coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
            coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
            coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
            coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
            icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
        }
        coef_StartPos += grid_RRow;
        iOne = 2 * iOne;
        jOne = 2 * jOne;
    }

    // device memory allocation
    cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
    cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
    cudaMalloc( &d_VPotentialPrev, RRow * ZColumn * PhiSlice * sizeof(float) );
    cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );

    cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
    cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
    cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
    cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );

    // set memory to zero
    cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
    cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
    cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
    cudaMemset( d_VPotentialPrev, 0, RRow * ZColumn * PhiSlice * sizeof(float) );
    cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );


    // copy data from host to device
    cudaMemcpy( d_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice ); //check
    cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice ); //check
    cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice );

    // max exact    float maxAbsExact = GetAbsMax(VPotentialExact,RRow * PhiSlice * ZColumn);
    float maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
    dim3 error_BlockPerGrid((RRow < 16) ? 1 : (RRow / 16), (ZColumn < 16) ? 1 : (ZColumn / 16), PhiSlice);
    dim3 error_ThreadPerBlock(16, 16);


    for (int cycle = 0; cycle < nCycle; cycle++)
    {
    /*V-Cycle starts*/

        // error conv
        //  cudaMemcpy( temp_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );

        cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToHost );
        errorExact[cycle] = GetErrorNorm2(temp_VPotential,VPotentialExact,RRow * PhiSlice,ZColumn, maxAbsExact);


        // V-Cycle => Finest Grid
```

```
        iOne = 1 << (gridFrom - 1);
        jOne = 1 << (gridFrom - 1);

        grid_RRow       = ((RRow - 1) / iOne) + 1;
        grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

        grid_StartPos = 0;
        coef_StartPos = 0;

        // pre-compute constant memory
        h   = gridSizeR * iOne;
        h2  = h * h;
        ih2 = 1.0 / h2;

        tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
        tempRatioPhi = ratioPhi * iOne * iOne;

        // copy constant to device memory
        cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
        cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

        // set kernel grid size and block size
        dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
    grid_ZColumn / 16), PhiSlice);
        dim3 grid_ThreadPerBlock(16, 16);

        // red-black gauss seidel relaxation (nPre times)
        for (int i = 0; i < nPre; i++)
        {
            relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
    d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
            //cudaDeviceSynchronize();
            relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
    d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
            //cudaDeviceSynchronize();
        }

        // residue calculation
        residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity,
     d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );
        //cudaDeviceSynchronize();

        // V-Cycle => from finer to coarsest grid
        for (int step = gridFrom + 1; step <= gridTo; step++)
        {
            iOne = 1 << (step - 1);
            jOne = 1 << (step - 1);

            grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos += grid_RRow;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            // pre-compute constant memory
            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
     );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
     );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
    grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // restriction
            restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
    d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // zeroing V
            zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
    grid_ZColumn, grid_PhiSlice );
```

```
            //cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPre times)
            for (int i = 0; i < nPre; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
            }

            // residue calculation
            if (step < gridTo)
            {
                residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );
                //cudaDeviceSynchronize();

            }
        }

        // up one


        {
            int step = (gridTo - 1);
            iOne = iOne / 2;
            jOne = jOne / 2;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos -= grid_RRow;

            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : ( grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

        // prolongation
            prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
//          cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPost times)
            for (int i = 0; i < nPost; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
            }
        }

        // down one
        {
            residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );

            iOne = iOne * 2;
            jOne = jOne * 2;

            grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos += grid_RRow;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;
```

```

            // pre-compute constant memory
            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;


            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // restriction
            restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // zeroing V
            zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPre times)
            for (int i = 0; i < nPre; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
            }

        }

        // up two from gridTo - 1, to gridTo -3
        for (int step = (gridTo - 1); step >= gridTo - 3; step--)
        {
            iOne = iOne / 2;
            jOne = jOne / 2;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos -= grid_RRow;

            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // prolongation
            prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
//          cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPost times)
            for (int i = 0; i < nPost; i++)
```

```
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
            }
        }

        // down to from gridTo - 1, to gridTo -3
        for (int step = gridTo - 3; step <= gridTo - 1; step++)
        {
            iOne = iOne * 2;
            jOne = jOne * 2;

            grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos += grid_RRow;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            // pre-compute constant memory
            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // restriction
            restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // zeroing V
            zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPre times)
            for (int i = 0; i < nPre; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
            }

            // residue calculation
            if (step < gridTo)
            {
                residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );
                //cudaDeviceSynchronize();

            }
        }



        {
            int step = (gridTo - 1);
            iOne = iOne / 2;
            jOne = jOne / 2;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos -= grid_RRow;
```

```

            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

        // prolongation
            prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
//          cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPost times)
            for (int i = 0; i < nPost; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
            }
        }

        // down one
        {
            residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );

            iOne = iOne * 2;
            jOne = jOne * 2;

            grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos += grid_RRow;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            // pre-compute constant memory
            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;


            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // restriction
            restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // zeroing V
            zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
            //cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPre times)
```

```
            for (int i = 0; i < nPre; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
                //cudaDeviceSynchronize();
            }

        }


        // V-Cycle => from coarser to finer grid
        for (int step = (gridTo - 1); step >= gridFrom; step--)
        {
            iOne = iOne / 2;
            jOne = jOne / 2;

            grid_RRow       = ((RRow - 1) / iOne) + 1;
            grid_ZColumn    = ((ZColumn - 1) / jOne) + 1;

            grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
            coef_StartPos -= grid_RRow;

            h   = gridSizeR * iOne;
            h2  = h * h;
            ih2 = 1.0 / h2;

            tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
            tempRatioPhi = ratioPhi * iOne * iOne;

            // copy constant to device memory
            cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
            cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

            // set kernel grid size and block size
            dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : ( grid_ZColumn / 16), PhiSlice);
            dim3 grid_ThreadPerBlock(16, 16);

            // prolongation
            prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow, grid_ZColumn, grid_PhiSlice );
//          cudaDeviceSynchronize();

            // red-black gauss seidel relaxation (nPost times)
            for (int i = 0; i < nPost; i++)
            {
                relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
                relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
//              cudaDeviceSynchronize();
            }
        }

    /*V-Cycle ends*/

        errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev, d_VPotential, d_EpsilonError, RRow, ZColumn, PhiSlice);

        cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );


        errorConv[cycle] = *EpsilonError  / (RRow * ZColumn * PhiSlice);

        if (errorConv[cycle] < convErr)
        {
            //errorConv
            nCycle = cycle;
            iparam[3] = nCycle;
            break;
        }

        cudaMemcpy( d_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );
        cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
```

```
    }

    cudaDeviceSynchronize();
    // copy result from device to host
    cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
  cudaMemcpyDeviceToHost );

    memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));

    // free device memory
    cudaFree( d_VPotential );
    cudaFree( d_VPotentialPrev );
    cudaFree( d_EpsilonError );


    cudaFree( d_DeltaResidue );
    cudaFree( d_RhoChargeDensity );
    cudaFree( d_coef1 );
    cudaFree( d_coef2 );
    cudaFree( d_coef3 );
    cudaFree( d_coef4 );
    cudaFree( d_icoef4 );

    // free host memory
    free( coef1 );
    free( coef2 );
    free( coef3 );
    free( coef4 );
    free( icoef4 );
    free( temp_VPotential );
    //free( temp_VPotentialPrev );
}
```

Indeks