

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



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

1	PoissonSolver3DCylindricalGPU	1
2	Petunjuk Penggunaan	3
3	Indeks Kelas	9
3.1	Daftar Kelas	9
4	Indeks File	11
4.1	Daftar File	11
5	Dokumentasi Kelas	13
5.1	Referensi Struct PoissonSolver3DCylindricalGPU::MGParameters	13
5.1.1	Keterangan Lengkap	13
5.2	Referensi Kelas PoissonSolver3DCylindricalGPU	14
5.2.1	Keterangan Lengkap	15
5.2.2	Dokumentasi Anggota: Enumerasi	15
5.2.2.1	CycleType	15
5.2.2.2	GridTransferType	16
5.2.2.3	InterpType	16
5.2.2.4	RelaxType	16
5.2.2.5	StrategyType	17
5.2.3	Dokumentasi Konstruktor & Destruktor	17
5.2.3.1	PoissonSolver3DCylindricalGPU()	17
5.2.4	Dokumentasi Anggota: Fungsi	18
5.2.4.1	PoissonSolver3D()	18
5.2.4.2	SetExactSolution()	18
5.2.5	Dokumentasi Anggota: Data	19
5.2.5.1	fgkIFCRadius	19

6 Dokumentasi File	21
6.1 Referensi File <code>/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/example/↔ PoissonSolver3DGPUtest.h</code>	21
6.1.1 Keterangan Lengkap	22
6.1.2 Dokumentasi Fungsi	22
6.1.2.1 <code>DoPoissonSolverExperiment()</code>	22
6.2 Referensi File <code>/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/interface/↔ PoissonSolver3DCylindricalGPU.h</code>	24
6.3 Referensi File <code>/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/kernel/↔ PoissonSolver3DGPU.cu</code>	24
6.3.1 Keterangan Lengkap	25
6.3.2 Dokumentasi Fungsi	26
6.3.2.1 <code>PoissonMultigrid3DSemiCoarseningGPUError()</code>	26
6.3.2.2 <code>PoissonMultigrid3DSemiCoarseningGPUErrorFCycle()</code>	29
6.3.2.3 <code>PoissonMultigrid3DSemiCoarseningGPUErrorWCycle()</code>	34
6.3.2.4 <code>PrintMatrix()</code>	43
6.3.2.5 <code>relaxationGaussSeidelBlack()</code>	44
6.3.2.6 <code>relaxationGaussSeidelRed()</code>	45
6.3.2.7 <code>residueCalculation()</code>	46
6.3.2.8 <code>Restrict_Boundary()</code>	47
6.3.2.9 <code>restriction2DFull()</code>	48
6.4 Referensi File <code>/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/kernel/↔ PoissonSolver3DGPU.h</code>	49
6.4.1 Keterangan Lengkap	50
6.4.2 Dokumentasi Fungsi	50
6.4.2.1 <code>PoissonMultigrid3DSemiCoarseningGPUError()</code>	50
6.4.2.2 <code>PoissonMultigrid3DSemiCoarseningGPUErrorFCycle()</code>	54
6.4.2.3 <code>PoissonMultigrid3DSemiCoarseningGPUErrorWCycle()</code>	59
Indeks	69

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 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 Silindrikan.
- Cara **memakai** Pustaka Pemercepat Komputasi berbasis GPU CUDA untuk Penyelesaian Persamaan *Poisson* dalam Koordinat Silindrikan dalam C++.

Struktur Berkas

Struktur berkas pustaka Pemercepat Komputasi berbasis GPU CUDA untuk Penyelesaian Persamaan *Poisson* dalam Koordinat Silindrikan 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
|   |-- PoissonSolver3DGPUPTest.cpp
|   |-- PoissonSolver3DGPUPTest.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
..
-- Installing 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 `libPoissonSolver3DCylindricalGPU.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 poissonsolvergpu)

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 membangun proyek yang menggunakan pustaka misal **builddexamplepoissonsolver** lalu jalankan **cmake** dengan flag spesial yaitu ****DCMAKE_PREFIX_PATH**** yang ditetapkan dengan absolute path tempat direktori **PoissonSolver3DCylindricalGPU** dibangun.

```
$ mkdir builddexamplepoissonsolver
$ cd builddexamplepoissonsolver
$ 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/builddexamplepoissonsolver
```

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/builddexamplepoissonsolver
[ 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:

PoissonSolver3DCylindricalGPU::MGParameters	13
PoissonSolver3DCylindricalGPU Kelas ini merupakan interface PoissonSolver 3D dalam koordinat silindrikan yang diterapkan pada NVIDIA Cuda	14

Bab 4

Indeks File

4.1 Daftar File

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

/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/example/ PoissonSolver3DGP ↔	
UTest.cpp	??
/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/example/ PoissonSolver3DGPU ↔	
Test.h	
Berkas ini berisi definisi fungsi untuk memakai pustaka libPoissonSolver3DCylindricalGPU.so . . .	21
/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/interface/ PoissonSolver3D ↔	
CylindricalGPU.cxx	??
/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/interface/ PoissonSolver3D ↔	
CylindricalGPU.h	24
/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/kernel/ PoissonSolver3DGPU .cu	
Berkas ini berisi implementasi kernel dalam cuda untuk PoissonSolver Cylindrical berbasis Multigrid	24
/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	49

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/PoissonSolver3DCylindricalGPU.h](#)

5.2 Referensi Kelas PoissonSolver3DCylindricalGPU

Kelas ini merupakan interface PoissonSolver 3D dalam koordinat silindrikan yang diterapkan pada NVIDIA 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 phiSlice, 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 `fgkTPCZO` = 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 silindrikan yang diterapkan pada NVIDIA 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     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 silindrikan 3D

Disediakan algoritma iteratif

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

Parameter

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

Definisi pada baris 85 dalam file PoissonSolver3DCylindricalGPU.cxx.

```
87
88
89     fNRRow = nRRow;
90     fNZColumn = nZColumn;
91     fPhiSlice = phiSlice;
92
93     PoissonMultiGrid3D2D(matricesV, matricesCharge, nRRow, nZColumn, phiSlice, symmetry);
94
95
96 }
```

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

<i>exactSolution</i>	float * array 1D sebesar nRRow * nZColumn * phiSlice
<i>nRRow</i>	int jumlah titik grid pada arah radial
<i>nZColumn</i>	int jumlah titik grid pada arah z
<i>phiSlice</i>	int jumlah titik grid pada arah sudut (phi)

Definisi pada baris 177 dalam file PoissonSolver3DCylindricalGPU.cxx.

```

177
178     {
179         fNRRow = nRRow;
180         fNZColumn = nZColumn;
181         fPhiSlice = phiSlice;
182         fExactSolutionF = new float[fNRRow * fPhiSlice * fNZColumn];
183         fExactPresent = true;
184         fMaxExact = 0.0;
185         for (int i=0; i<nRRow*nZColumn*phiSlice; i++) {
186             fExactSolutionF[i] = exactSolution[i];
187             if (abs(fExactSolutionF[i]) > fMaxExact) fMaxExact = abs(fExactSolutionF[i]);
188         }

```

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/PoissonSolver3DCylindricalGPU.h](#)
- [/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/interface/PoissonSolver3DCylindricalGPU.cxx](#)

Bab 6

Dokumentasi File

6.1 Referensi File /home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3DGPUTest.h

Berkas ini berisi definisi fungsi untuk memakai pustaka libPoissonSolver3DCylindricalGPU.so.

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

Fungsi

- void [DoPoissonSolverExperiment](#) (const int kRows, const int kColumns, const int kPhiSlices, const int kIterations, 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)
Inisiasi 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 [fgkFCRadius](#) = 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 fungsi untuk memakai pustaka `libPoissonSolver3DCylindricalGPU.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 PoissonSolver3DGPUPTest.cpp.

```

15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103

```

```

{
    int kPhiSlicesPerSector = kPhiSlices/18;
    const float gridSizeR = (fgkOFCRadius-fgkIFCRadius) / (kRows-1) ;
    const float gridSizeZ = fgkTPCZ0 / (kColumns-1) ;
    const float gridSizePhi = (M_PI * 2) / ( 18.0 * kPhiSlicesPerSector);
    int size = kRows * kColumns * kPhiSlices;
    float * VPotential = new float[size];
    float * VPotentialExact = new float[size];
    float * RhoCharge = new float[size];
    float * errorConv = new float[200];
    float * errorExact = new float[200];
    InitVoltandCharge3D (VPotentialExact,VPotential,RhoCharge,kRows,kColumns,kPhiSlices,
    gridSizeR,gridSizeZ,gridSizePhi);
    // create poissonSolver
    PoissonSolver3DCylindricalGPU *poissonSolver = new
    PoissonSolver3DCylindricalGPU();
    PoissonSolver3DCylindricalGPU::fgConvergenceError = 1e
    -6;
    // zeroing array of error
    poissonSolver->SetExactSolution(VPotentialExact,kRows,kColumns, kPhiSlices);
    // Case 1. Set the strategy as multigrid, fullmultigrid, and full 3d
    poissonSolver->SetStrategy (PoissonSolver3DCylindricalGPU::kMultiGrid
    );
    poissonSolver->SetCycleType (PoissonSolver3DCylindricalGPU::kFCycle
    );
    // TStopwatch w;
    std::clock_t start,stop;
    double duration;
    start = std::clock();
    //w.Start();
    poissonSolver->PoissonSolver3D (VPotential, RhoCharge, kRows, kColumns, kPhiSlices,
    kIterations, kSymmetry) ;
    //w.Stop();
}

```

```

104     stop = std::clock();
114     duration = ( stop - start ) / (double) CLOCKS_PER_SEC;
115
116     std::cout<<"Poisson Solver 3D Cylindrical GPU test" << '\n';
117     std::cout<<"Ukuran grid (r,phi,z) = (" << kRows << ", " << kColumns << ", " << kPhiSlices <<") \n";
118     std::cout<<"Waktu komputasi: \t\t" << duration << " s \n";
119     std::cout<<"Jumlah iterasi siklus multigrad: \t" << poissonSolver->
fIterations << '\n';
120     std::cout<<"Iterasi\tError Convergen\tError Absolut \n";
121     std::cout<< std::scientific;
122     for (int i=0;i<poissonSolver->fIterations;i++) {
123         std::cout<< "[" << i << "]:\t" << poissonSolver->GetErrorConv(i) << "\t" << poissonSolver->
GetErrorExact(i) << '\n' << std::scientific;
124     }
125     delete poissonSolver;
126     delete VPotential;
127     delete VPotentialExact;
128     delete RhoCharge;
129 }

```

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 silindrikan yang diterapkan pada NVIDIA 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 Multigrad.

```

#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 PhiSlice)
- `__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 RRow, 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 RRow, const int ZColumn, const int PhiSlice, const int Symmetry, float *fparam, int *iparam, bool isExactPresent, 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 isExactPresent, 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 muatan ρ

Parameter

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

Mengembalikan

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

Definisi pada baris 893 dalam file PoissonSolver3DGPU.cu.

```
907 {
908     // variables for CPU memory
909     float *temp_VPotential;
910     float *VPotentialPrev;
911     float *EpsilonError;
912
913     // variables for GPU memory
914     float *d_VPotential;
915     float *d_RhoChargeDensity;
916     float *d_DeltaResidue;
917     float *d_VPotentialPrev;
918     float *d_EpsilonError;
919
920     float *d_coef1;
921     float *d_coef2;
922     float *d_coef3;
923     float *d_coef4;
924     float *d_icoef4;
925
926     // variables for coefficient calculations
927     float *coef1;
928     float *coef2;
929     float *coef3;
930     float *coef4;
```

```

931     float *icoef4;
932     float tempRatioZ;
933     float tempRatioPhi;
934     float radius;
935
936     int gridFrom;
937     int gridTo;
938     int loops;
939
940
941     // variables passed from ALIROOT
942     float gridSizeR    = fparam[0];
943     float gridSizePhi  = fparam[1];
944     float gridSizeZ    = fparam[2];
945     float ratioPhi     = fparam[3];
946     float ratioZ       = fparam[4];
947     float convErr      = fparam[5];
948     float IFCRadius    = fparam[6];
949     int nPre           = iparam[0];
950     int nPost          = iparam[1];
951     int maxLoop        = iparam[2];
952     int nCycle         = iparam[3];
953
954     // variables for calculating GPU memory allocation
955     int grid_RRow;
956     int grid_ZColumn;
957     int grid_PhiSlice = PhiSlice;
958     int grid_Size = 0;
959     int grid_StartPos;
960     int coef_Size = 0;
961     int coef_StartPos;
962     int iOne, jOne;
963     float h, h2, ih2;
964
965     // variables for calculating multigrid maximum depth
966     int depth_RRow = 0;
967     int depth_ZColumn = 0;
968     int temp_RRow = RRow;
969     int temp_ZColumn = ZColumn;
970
971     // calculate depth for multigrid
972     while (temp_RRow >= 1) depth_RRow++;
973     while (temp_ZColumn >= 1) depth_ZColumn++;
974
975     loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
976     loops = (loops > maxLoop) ? maxLoop : loops;
977
978     gridFrom = 1;
979     gridTo = loops;
980
981     // calculate GPU memory allocation for multigrid
982     for (int step = gridFrom; step <= gridTo; step++)
983     {
984         grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
985         grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;
986
987         grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
988         coef_Size += grid_RRow;
989     }
990
991     // allocate memory for temporary output
992     temp_VPotential = (float *) malloc(grid_Size * sizeof(float));
993     VPotentialPrev = (float *) malloc(RRow * ZColumn * PhiSlice * sizeof(float));
994     EpsilonError = (float *) malloc(1 * sizeof(float));
995
996
997     // allocate memory for relaxation coefficient
998     coef1 = (float *) malloc(coef_Size * sizeof(float));
999     coef2 = (float *) malloc(coef_Size * sizeof(float));
1000     coef3 = (float *) malloc(coef_Size * sizeof(float));
1001     coef4 = (float *) malloc(coef_Size * sizeof(float));
1002     icoef4 = (float *) malloc(coef_Size * sizeof(float));
1003
1004     // pre-compute relaxation coefficient
1005     coef_StartPos = 0;
1006     iOne = 1 << (gridFrom - 1);
1007     jOne = 1 << (gridFrom - 1);
1008
1009     for (int step = gridFrom; step <= gridTo; step++)
1010     {
1011         grid_RRow = ((RRow - 1) / iOne) + 1;
1012
1013         h = gridSizeR * iOne;
1014         h2 = h * h;
1015         ih2 = 1.0 / h2;
1016
1017         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);

```

```

1018     tempRatioPhi = ratioPhi * iOne * iOne;
1019
1020     for (int i = 1; i < grid_RRow - 1; i++)
1021     {
1022         radius = IFCRadius + i * h;
1023         coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
1024         coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
1025         coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
1026         coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
1027         icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
1028     }
1029     coef_StartPos += grid_RRow;
1030     iOne = 2 * iOne;
1031     jOne = 2 * jOne;
1032 }
1033
1034 // device memory allocation
1035 cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
1036 cudaMalloc( &d_VPotentialPrev, RRow * ZColumn * PhiSlice * sizeof(float) );
1037 cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );
1038 cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
1039 cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
1040 cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
1041 cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
1042 cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
1043 cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
1044 cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );
1045
1046 // set memory to zero
1047 cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
1048 cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
1049 cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
1050 cudaMemset( d_VPotentialPrev, 0, RRow * ZColumn * PhiSlice * sizeof(float) );
1051 cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
1052
1053
1054 // copy data from host to device
1055 cudaMemcpy( d_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice
); //check
1056 cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyHostToDevice ); //check
1057 cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1058 cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1059 cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1060 cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1061 cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1062 cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyHostToDevice );
1063
1064 // max exact
1065
1066 // float maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
1067 float maxAbsExact = 1.0;
1068
1069 if (isExactPresent == true)
1070     maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
1071 dim3 error_BlockPerGrid((RRow < 16) ? 1 : (RRow / 16), (ZColumn < 16) ? 1 : (ZColumn / 16), PhiSlice);
1072 dim3 error_ThreadPerBlock(16, 16);
1073
1074
1075 for (int cycle = 0; cycle < nCycle; cycle++)
1076 {
1077     cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
1078     if (isExactPresent == true) errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact,
RRow * PhiSlice, ZColumn, maxAbsExact);
1079
1080
1081     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);
1082
1083     errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev, d_VPotential,
d_EpsilonError, RRow, ZColumn, PhiSlice);
1084
1085     cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );
1086
1087     errorConv[cycle] = *EpsilonError / (RRow * ZColumn * PhiSlice);
1088
1089     if (errorConv[cycle] < convErr)
1090     {
1091         //errorConv
1092         nCycle = cycle;
1093         break;
1094     }
1095 }
1096
1097

```



```

1098     cudaMemcpy( d_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1099     cudaMemcpyDeviceToDevice );
1100     cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
1101 }
1102 iparam[3] = nCycle;
1103
1104 // for (int cycle = 0; cycle < nCycle; cycle++)
1105 // {
1106 //     cudaMemcpy( temp_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1107 //     cudaMemcpyDeviceToHost );
1108
1109 //     VCycleSemiCoarseningGPU(d_VPotential, d_RhoChargeDensity, d_DeltaResidue, d_coef1, d_coef2,
1110 //     d_coef3, d_coef4, d_icoef4, gridSizeR, ratioZ, ratioPhi, RRow, ZColumn, PhiSlice, gridFrom, gridTo, nPre, nPost);
1111 //     cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1112 //     cudaMemcpyDeviceToHost );
1113 //     errorConv[cycle] = GetErrorNorm2(temp_VPotential, temp_VPotentialPrev, RRow * PhiSlice, ZColumn,
1114 //     1.0);
1115 //     //errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact, RRow * PhiSlice, ZColumn,
1116 //     1.0);
1117 // }
1118 // copy result from device to host
1119 cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1120 cudaMemcpyDeviceToHost );
1121
1122 memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
1123
1124 // free device memory
1125 cudaFree( d_VPotential );
1126 cudaFree( d_DeltaResidue );
1127 cudaFree( d_RhoChargeDensity );
1128 cudaFree( d_VPotentialPrev );
1129 cudaFree( d_EpsilonError );
1130 cudaFree( d_coef1 );
1131 cudaFree( d_coef2 );
1132 cudaFree( d_coef3 );
1133 cudaFree( d_coef4 );
1134 cudaFree( d_icoef4 );
1135
1136 // free host memory
1137 free( coef1 );
1138 free( coef2 );
1139 free( coef3 );
1140 free( coef4 );
1141 free( icoef4 );
1142 free( temp_VPotential );
1143 free( VPotentialPrev );
1144 }

```

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.

```

1851 {
1852     // variables for CPU memory
1853     float *temp_VPotential;
1854     float *VPotentialPrev;
1855     float *EpsilonError;
1856
1857     // variables for GPU memory
1858     float *d_VPotential;
1859     float *d_RhoChargeDensity;
1860     float *d_DeltaResidue;
1861     float *d_coef1;
1862     float *d_coef2;
1863     float *d_coef3;
1864     float *d_coef4;
1865     float *d_icoef4;
1866     float *d_VPotentialPrev;
1867     float *d_EpsilonError;
1868
1869
1870     // variables for coefficient calculations
1871     float *coef1;
1872     float *coef2;
1873     float *coef3;
1874     float *coef4;
1875     float *icoef4;
1876     float tempRatioZ;
1877     float tempRatioPhi;
1878     float radius;
1879
1880     int gridFrom;
1881     int gridTo;
1882     int loops;
1883
1884     // variables passed from ALIROOT
1885     float gridSizeR = fparam[0];
1886     //float gridSizePhi = fparam[1];
1887     //float gridSizeZ = fparam[2];
1888     float ratioPhi = fparam[3];
1889     float ratioZ = fparam[4];
1890     float convErr = fparam[5];
1891     float IFCRadius = fparam[6];
1892     int nPre = iparam[0];
1893     int nPost = iparam[1];
1894     int maxLoop = iparam[2];
1895     int nCycle = iparam[3];
1896
1897     // variables for calculating GPU memory allocation
1898     int grid_RRow;
1899     int grid_ZColumn;
1900     int grid_PhiSlice = PhiSlice;
1901     int grid_Size = 0;
1902     int grid_StartPos;
1903     int coef_Size = 0;
1904     int coef_StartPos;
1905     int iOne, jOne;
1906     float h, h2, ih2;
1907
1908     // variables for calculating multigrid maximum depth
1909     int depth_RRow = 0;
1910     int depth_ZColumn = 0;
1911     int temp_RRow = RRow;
1912     int temp_ZColumn = ZColumn;
1913
1914     // calculate depth for multigrid
1915     while (temp_RRow >= 1) depth_RRow++;
1916     while (temp_ZColumn >= 1) depth_ZColumn++;
1917
1918     loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
1919     loops = (loops > maxLoop) ? maxLoop : loops;
1920
1921     gridFrom = 1;
1922     gridTo = loops;
1923
1924     // calculate GPU memory allocation for multigrid
1925     for (int step = gridFrom; step <= gridTo; step++)
1926     {
1927         grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
1928         grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;
1929
1930         grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
1931         coef_Size += grid_RRow;
1932     }
1933
1934     // allocate memory for temporary output
1935     temp_VPotential = (float *) malloc(grid_Size * sizeof(float));
1936     VPotentialPrev = (float *) malloc(grid_Size * sizeof(float));
1937     EpsilonError = (float *) malloc(1 * sizeof(float));

```

```

1938
1939
1940
1941     for (int i=0;i<grid_Size;i++) temp_VPotential[i] = 0.0;
1942
1943
1944     // allocate memory for relaxation coefficient
1945     coef1 = (float *) malloc(coef_Size * sizeof(float));
1946     coef2 = (float *) malloc(coef_Size * sizeof(float));
1947     coef3 = (float *) malloc(coef_Size * sizeof(float));
1948     coef4 = (float *) malloc(coef_Size * sizeof(float));
1949     icoeff4 = (float *) malloc(coef_Size * sizeof(float));
1950
1951     // pre-compute relaxation coefficient
1952     // restrict boundary
1953     coef_StartPos = 0;
1954     grid_StartPos = 0;
1955
1956     iOne = 1 << (gridFrom - 1);
1957     jOne = 1 << (gridFrom - 1);
1958
1959     for (int step = gridFrom; step <= gridTo; step++)
1960     {
1961         grid_RRow = ((RRow - 1) / iOne) + 1;
1962         grid_ZColumn = ((ZColumn - 1) / iOne) + 1;
1963
1964         h = gridSizeR * iOne;
1965         h2 = h * h;
1966         ih2 = 1.0 / h2;
1967
1968         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1969         tempRatioPhi = ratioPhi * iOne * iOne;
1970
1971         for (int i = 1; i < grid_RRow - 1; i++)
1972         {
1973             radius = IFCRadius + i * h;
1974             coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
1975             coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
1976             coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
1977             coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
1978             icoeff4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
1979         }
1980
1981         // call restrict boundary
1982         if (step == gridFrom) {
1983             // Copy original VPotential to tempPotential
1984             memcpy(temp_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
1985         }
1986         // else
1987         //{
1988         // Restrict_Boundary(temp_VPotential, grid_RRow, grid_ZColumn, PhiSlice, grid_StartPos);
1989         //}
1990
1991
1992
1993         coef_StartPos += grid_RRow;
1994         grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1995
1996
1997         iOne = 2 * iOne;
1998         jOne = 2 * jOne;
1999     }
2000
2001     // device memory allocation
2002     cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
2003     cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
2004     cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
2005     cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
2006     cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
2007     cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
2008     cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
2009     cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );
2010     cudaMalloc( &d_VPotentialPrev, grid_Size * sizeof(float) );
2011     cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );
2012
2013
2014     // set memory to zero
2015     cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
2016     cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
2017     cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
2018     cudaMemset( d_VPotentialPrev, 0, grid_Size * sizeof(float) );
2019     cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
2020
2021     // set memory to zero
2022     cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
2023     cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
2024     cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );

```

```

2025
2026 // copy data from host to device1
2027 // case of FCycle you need to copy all boundary for all
2028 cudaMemcpy( d_VPotential, temp_VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice ); //check
2029 // cudaMemcpy( d_VPotential, VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice ); //check
2030
2031 cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float),
  cudaMemcpyHostToDevice ); //check
2032 // cudaMemcpy( d_RhoChargeDensity, temp_VPotentialPrev, grid_Size * sizeof(float), cudaMemcpyHostToDevice
  ); //check
2033 cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2034 cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2035 cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2036 cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2037 cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2038 // cudaMemcpy( d_VPotentialPrev, temp_VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice );
2039
2040 // cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
  cudaMemcpyHostToDevice );
2041
2042 // max exact
2043
2044 float maxAbsExact = 1.0;
2045
2046 if (isExactPresent == true)
2047     maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
2048
2049
2050
2051 // init iOne,grid_RRow, grid_ZColumn, grid_StartPos, coef_StartPos
2052 iOne = 1 << (gridFrom - 1);
2053 jOne = 1 << (gridFrom - 1);
2054
2055 grid_RRow      = ((RRow - 1) / iOne) + 1;
2056 grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
2057
2058 grid_StartPos = 0;
2059 coef_StartPos = 0;
2060
2061
2062 for (int step = gridFrom + 1; step <= gridTo; step++)
2063 {
2064
2065     iOne = 1 << (step - 1);
2066     jOne = 1 << (step - 1);
2067
2068     grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
2069     coef_StartPos += grid_RRow;
2070
2071     grid_RRow      = ((RRow - 1) / iOne) + 1;
2072     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
2073
2074     // pre-compute constant memory
2075     h = gridSizeR * iOne;
2076     h2 = h * h;
2077     ih2 = 1.0 / h2;
2078
2079     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
2080     tempRatioPhi = ratioPhi * iOne * iOne;
2081
2082     // copy constant to device memory
2083     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
2084     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
2085     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2086     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2087     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2088     cudaMemcpyToSymbol( d_tempRatioPhi, &tempRatioPhi, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2089
2090     // set kernel grid size and block size
2091     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
  grid_ZColumn / 16), PhiSlice);
2092     dim3 grid_ThreadPerBlock(16, 16);
2093
2094     // restriction
2095     restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
  d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice );
2096
2097     // restrict boundary (already done in cpu)
2099 //     PrintMatrix(temp_VPotential,grid_RRow * PhiSlice,grid_ZColumn);
2100     restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_VPotential,
  grid_RRow, grid_ZColumn, grid_PhiSlice );
2101
2102
2103 }
2104
2105 dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn
  / 16), PhiSlice);
2106 dim3 grid_ThreadPerBlock(16, 16);

```

```

2107
2108
2109 // relax on the coarsest
2110 // red-black gauss seidel relaxation (nPre times)
2111 // printf("rho\n");
2112 // cudaMemcpy( temp_VPotential, d_RhoChargeDensity + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
    sizeof(float), cudaMemcpyDeviceToHost );
2113 // PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);
2114
2115 // printf("v\n");
2116 // cudaMemcpy( temp_VPotential, d_VPotential + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
    sizeof(float), cudaMemcpyDeviceToHost );
2117 // PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);
2118 for (int i = 0; i < nPre; i++)
2119 {
2120     relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
    d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
2121     cudaDeviceSynchronize();
2122     relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
    d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
2123     cudaDeviceSynchronize();
2124 }
2125
2126 // printf("v after relax\n");
2127 // cudaMemcpy( temp_VPotential, d_VPotential + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
    sizeof(float), cudaMemcpyDeviceToHost );
2128 // PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);
2129
2130 // V-Cycle => from coarser to finer grid
2131 for (int step = gridTo - 1 ; step >= gridFrom; step--)
2132 {
2133     iOne = iOne / 2;
2134     jOne = jOne / 2;
2135
2136     grid_RRow = ((RRow - 1) / iOne) + 1;
2137     grid_ZColumn = ((ZColumn - 1) / jOne) + 1;
2138
2139     grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
2140     coef_StartPos -= grid_RRow;
2141
2142     h = gridSizeR * iOne;
2143     h2 = h * h;
2144     ih2 = 1.0 / h2;
2145
2146     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
2147     tempRatioPhi = ratioPhi * iOne * iOne;
2148
2149     // copy constant to device memory
2150     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
2151     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
2152     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2153     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2154     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2155
2156
2157
2158     // set kernel grid size and block size
2159     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
    grid_ZColumn / 16), PhiSlice);
2160     dim3 grid_ThreadPerBlock(16, 16);
2161
2162
2163     prolongation2DHalfNoAdd<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
    grid_ZColumn, grid_PhiSlice );
2164
2165
2166
2167
2168     // just
2169
2170     // max exact
2171     cudaMemcpy( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, grid_RRow *
    grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );
2172
2173     float maxAbsExact = 1.0;
2174
2175     if (isExactPresent == true)
2176         maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
2177     dim3 error_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
    grid_ZColumn / 16), PhiSlice);
2178     dim3 error_ThreadPerBlock(16, 16);
2179
2180
2181
2182     for (int cycle = 0; cycle < nCycle; cycle++)
2183     {
2184

```

```

2185
2186         if (step == gridFrom) {
2187             cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
2188             if (isExactPresent == true )errorExact[cycle] = GetErrorNorm2(temp_VPotential,
VPotentialExact, RRow * PhiSlice,ZColumn, maxAbsExact);
2189         }
2190
2191
2192
2193         //cudaDeviceSynchronize();
2194         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);
2195
2196
2197
2198             //if (step == gridFrom) {
2199                 //cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
2200
2201                 //errorConv[cycle] = GetErrorNorm2(temp_VPotential, VPotentialPrev, RRow *
PhiSlice,ZColumn, 1.0);
2202
2203                 errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev +
grid_StartPos, d_VPotential + grid_StartPos, d_EpsilonError, grid_RRow, grid_ZColumn, PhiSlice);
2204
2205                 cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );
2206
2207                 errorConv[cycle] = *EpsilonError / (grid_RRow * grid_ZColumn * PhiSlice);
2208
2209                 if (errorConv[cycle]< convErr)
2210                 {
2211                     nCycle = cycle;
2212                     break;
2213                 }
2214
2215                 cudaMemcpy( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, grid_RRow *
grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );
2216                 cudaMemcpy( d_EpsilonError, 0, 1 * sizeof(float) );
2217
2218             }
2219
2220
2221         }
2222
2223         iparam[3] = nCycle;
2224
2225         // copy result from device to host
2226         cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
2227
2228         memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
2229
2230         // free device memory
2231         cudaFree( d_VPotential );
2232         cudaFree( d_DeltaResidue );
2233         cudaFree( d_RhoChargeDensity );
2234         cudaFree( d_coef1 );
2235         cudaFree( d_coef2 );
2236         cudaFree( d_coef3 );
2237         cudaFree( d_coef4 );
2238         cudaFree( d_icoef4 );
2239
2240         // free host memory
2241         free( coef1 );
2242         free( coef2 );
2243         free( coef3 );
2244         free( coef4 );
2245         free( icoef4 );
2246         free( temp_VPotential );
2247         free( VPotentialPrev );
2248     }

```

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 )

```

inner 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 coefficient 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

```

```

1217 // variables for calculating multigrid maximum depth
1218 int depth_RRow = 0;
1219 int depth_ZColumn = 0;
1220 int temp_RRow = RRow;
1221 int temp_ZColumn = ZColumn;
1222
1223 // calculate depth for multigrid
1224 while (temp_RRow >>= 1) depth_RRow++;
1225 while (temp_ZColumn >>= 1) depth_ZColumn++;
1226
1227 loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
1228 loops = (loops > maxLoop) ? maxLoop : loops;
1229
1230 gridFrom = 1;
1231 gridTo = loops;
1232
1233 // calculate GPU memory allocation for multigrid
1234 for (int step = gridFrom; step <= gridTo; step++)
1235 {
1236     grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
1237     grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;
1238
1239     grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
1240     coef_Size += grid_RRow;
1241 }
1242
1243 // allocate memory for temporary output
1244 temp_VPotential = (float *) malloc(grid_Size * sizeof(float));
1245 VPotentialPrev = (float *) malloc(RRow * ZColumn * PhiSlice * sizeof(float));
1246 EpsilonError = (float *) malloc(1 * sizeof(float));
1247
1248
1249 // allocate memory for relaxation coefficient
1250 coef1 = (float *) malloc(coef_Size * sizeof(float));
1251 coef2 = (float *) malloc(coef_Size * sizeof(float));
1252 coef3 = (float *) malloc(coef_Size * sizeof(float));
1253 coef4 = (float *) malloc(coef_Size * sizeof(float));
1254 iccoef4 = (float *) malloc(coef_Size * sizeof(float));
1255
1256 // pre-compute relaxation coefficient
1257 coef_StartPos = 0;
1258 iOne = 1 << (gridFrom - 1);
1259 jOne = 1 << (gridFrom - 1);
1260
1261 for (int step = gridFrom; step <= gridTo; step++)
1262 {
1263     grid_RRow = ((RRow - 1) / iOne) + 1;
1264
1265     h = gridSizeR * iOne;
1266     h2 = h * h;
1267     ih2 = 1.0 / h2;
1268
1269     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1270     tempRatioPhi = ratioPhi * iOne * iOne;
1271
1272     for (int i = 1; i < grid_RRow - 1; i++)
1273     {
1274         radius = IFCRadius + i * h;
1275         coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
1276         coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
1277         coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
1278         coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
1279         iccoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
1280     }
1281     coef_StartPos += grid_RRow;
1282     iOne = 2 * iOne;
1283     jOne = 2 * jOne;
1284 }
1285
1286 // device memory allocation
1287 cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
1288 cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
1289 cudaMalloc( &d_VPotentialPrev, RRow * ZColumn * PhiSlice * sizeof(float) );
1290 cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );
1291
1292 cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
1293 cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
1294 cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
1295 cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
1296 cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
1297 cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );
1298
1299 // set memory to zero
1300 cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
1301 cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
1302 cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
1303 cudaMemset( d_VPotentialPrev, 0, RRow * ZColumn * PhiSlice * sizeof(float) );

```



```

1304     cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
1305
1306
1307     // copy data from host to device
1308     cudaMemcpy( d_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice
); //check
1309     cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyHostToDevice ); //check
1310     cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1311     cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1312     cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1313     cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1314     cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1315     cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyHostToDevice );
1316
1317     // max exact float maxAbsExact = GetAbsMax(VPotentialExact,RRow * PhiSlice * ZColumn);
1318     float maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
1319     dim3 error_BlockPerGrid((RRow < 16) ? 1 : (RRow / 16), (ZColumn < 16) ? 1 : (ZColumn / 16), PhiSlice);
1320     dim3 error_ThreadPerBlock(16, 16);
1321
1322
1323     for (int cycle = 0; cycle < nCycle; cycle++)
1324     {
1325         /*V-Cycle starts*/
1326
1327         // error conv
1328         // cudaMemcpy( temp_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
1329
1330         cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
1331         errorExact[cycle] = GetErrorNorm2(temp_VPotential,VPotentialExact,RRow * PhiSlice,ZColumn,
maxAbsExact);
1332
1333
1334         // V-Cycle => Finest Grid
1335         iOne = 1 << (gridFrom - 1);
1336         jOne = 1 << (gridFrom - 1);
1337
1338         grid_RRow      = ((RRow - 1) / iOne) + 1;
1339         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1340
1341         grid_StartPos = 0;
1342         coef_StartPos = 0;
1343
1344         // pre-compute constant memory
1345         h = gridSizeR * iOne;
1346         h2 = h * h;
1347         ih2 = 1.0 / h2;
1348
1349         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1350         tempRatioPhi = ratioPhi * iOne * iOne;
1351
1352         // copy constant to device memory
1353         cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
1354         cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
1355         cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1356         cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1357         cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1358
1359         // set kernel grid size and block size
1360         dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1361         dim3 grid_ThreadPerBlock(16, 16);
1362
1363         // red-black gauss seidel relaxation (nPre times)
1364         for (int i = 0; i < nPre; i++)
1365         {
1366             relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1367             //cudaDeviceSynchronize();
1368             relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1369             //cudaDeviceSynchronize();
1370         }
1371
1372         // residue calculation
1373         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 );
1374         //cudaDeviceSynchronize();
1375
1376         // V-Cycle => from finer to coarsest grid
1377         for (int step = gridFrom + 1; step <= gridTo; step++)
1378         {
1379             iOne = 1 << (step - 1);
1380             jOne = 1 << (step - 1);

```

```

1381
1382     grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1383     coef_StartPos += grid_RRow;
1384
1385     grid_RRow      = ((RRow - 1) / iOne) + 1;
1386     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1387
1388     // pre-compute constant memory
1389     h = gridSizeR * iOne;
1390     h2 = h * h;
1391     ih2 = 1.0 / h2;
1392
1393     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1394     tempRatioPhi = ratioPhi * iOne * iOne;
1395
1396     // copy constant to device memory
1397     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1398     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1399     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1400     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1401     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1402
1403     // set kernel grid size and block size
1404     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1405     dim3 grid_ThreadPerBlock(16, 16);
1406
1407     // restriction
1408     restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
1409     //cudaDeviceSynchronize();
1410
1411     // zeroing V
1412     zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1413     //cudaDeviceSynchronize();
1414
1415     // red-black gauss seidel relaxation (nPre times)
1416     for (int i = 0; i < nPre; i++)
1417     {
1418         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1419         //cudaDeviceSynchronize();
1420         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1421         //cudaDeviceSynchronize();
1422     }
1423
1424     // residue calculation
1425     if (step < gridTo)
1426     {
1427         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 );
1428         //cudaDeviceSynchronize();
1429     }
1430 }
1431 }
1432
1433 // up one
1434
1435 {
1436     int step = (gridTo - 1);
1437     iOne = iOne / 2;
1438     jOne = jOne / 2;
1439
1440     grid_RRow      = ((RRow - 1) / iOne) + 1;
1441     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1442
1443     grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
1444     coef_StartPos -= grid_RRow;
1445
1446     h = gridSizeR * iOne;
1447     h2 = h * h;
1448     ih2 = 1.0 / h2;
1449
1450     tempRatioPhi = ratioPhi * iOne * iOne;
1451
1452     // copy constant to device memory
1453     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1454     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1455     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1456     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );

```

```

1460         cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1461
1462         // set kernel grid size and block size
1463         dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1464         dim3 grid_ThreadPerBlock(16, 16);
1465
1466         // prolongation
1467         prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1468         // cudaDeviceSynchronize();
1469
1470         // red-black gauss seidel relaxation (nPost times)
1471         for (int i = 0; i < nPost; i++)
1472         {
1473             relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1474             // cudaDeviceSynchronize();
1475             relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1476             // cudaDeviceSynchronize();
1477         }
1478     }
1479
1480     // down one
1481     {
1482         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 );
1483
1484         iOne = iOne * 2;
1485         jOne = jOne * 2;
1486
1487         grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1488         coef_StartPos += grid_RRow;
1489
1490         grid_RRow      = ((RRow - 1) / iOne) + 1;
1491         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1492
1493         // pre-compute constant memory
1494         h = gridSizeR * iOne;
1495         h2 = h * h;
1496         ih2 = 1.0 / h2;
1497
1498         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1499         tempRatioPhi = ratioPhi * iOne * iOne;
1500
1501
1502         // copy constant to device memory
1503         cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1504         cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1505
1506         cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1507         cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1508         cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1509
1510         // set kernel grid size and block size
1511         dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1512         dim3 grid_ThreadPerBlock(16, 16);
1513
1514         // restriction
1515         restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
1516         //cudaDeviceSynchronize();
1517
1518         // zeroing V
1519         zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1520         //cudaDeviceSynchronize();
1521
1522         // red-black gauss seidel relaxation (nPre times)
1523         for (int i = 0; i < nPre; i++)
1524         {
1525             relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1526             //cudaDeviceSynchronize();
1527             relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1528             //cudaDeviceSynchronize();
1529         }
1530     }
1531
1532     // up two from gridTo - 1, to gridTo -3
1533     for (int step = (gridTo - 1); step >= gridTo - 3; step--)
1534     {

```

```

1537         iOne = iOne / 2;
1538         jOne = jOne / 2;
1539
1540         grid_RRow      = ((RRow - 1) / iOne) + 1;
1541         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1542
1543         grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
1544         coef_StartPos -= grid_RRow;
1545
1546         h   = gridSizeR * iOne;
1547         h2  = h * h;
1548         ih2 = 1.0 / h2;
1549
1550         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1551         tempRatioPhi = ratioPhi * iOne * iOne;
1552
1553         // copy constant to device memory
1554         cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1555         cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1556         cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1557         cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1558         cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1559
1560         // set kernel grid size and block size
1561         dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1562         dim3 grid_ThreadPerBlock(16, 16);
1563
1564         // prolongation
1565         prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1566 //         cudaDeviceSynchronize();
1567
1568         // red-black gauss seidel relaxation (nPost times)
1569         for (int i = 0; i < nPost; i++)
1570         {
1571             relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1572 //             cudaDeviceSynchronize();
1573             relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1574 //             cudaDeviceSynchronize();
1575         }
1576     }
1577
1578     // down to from gridTo - 1, to gridTo -3
1579     for (int step = gridTo - 3; step <= gridTo - 1; step++)
1580     {
1581         iOne = iOne * 2;
1582         jOne = jOne * 2;
1583
1584         grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1585         coef_StartPos += grid_RRow;
1586
1587         grid_RRow      = ((RRow - 1) / iOne) + 1;
1588         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1589
1590         // pre-compute constant memory
1591         h   = gridSizeR * iOne;
1592         h2  = h * h;
1593         ih2 = 1.0 / h2;
1594
1595         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1596         tempRatioPhi = ratioPhi * iOne * iOne;
1597
1598         // copy constant to device memory
1599         cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1600         cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1601         cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1602         cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1603         cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1604
1605         // set kernel grid size and block size
1606         dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1607         dim3 grid_ThreadPerBlock(16, 16);
1608
1609         // restriction
1610         restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
1611         //cudaDeviceSynchronize();
1612
1613         // zeroing V

```

```

1614     zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1615     //cudaDeviceSynchronize();
1616
1617     // red-black gauss seidel relaxation (nPre times)
1618     for (int i = 0; i < nPre; i++)
1619     {
1620         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1621         //cudaDeviceSynchronize();
1622         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1623         //cudaDeviceSynchronize();
1624     }
1625
1626     // residue calculation
1627     if (step < gridTo)
1628     {
1629         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 );
1630         //cudaDeviceSynchronize();
1631     }
1632 }
1633 }
1634
1635
1636
1638 {
1639     int step = (gridTo - 1);
1640     iOne = iOne / 2;
1641     jOne = jOne / 2;
1642
1643     grid_RRow      = ((RRow - 1) / iOne) + 1;
1644     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1645
1646     grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
1647     coef_StartPos -= grid_RRow;
1648
1649     h = gridSizeR * iOne;
1650     h2 = h * h;
1651     ih2 = 1.0 / h2;
1652
1653     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1654     tempRatioPhi = ratioPhi * iOne * iOne;
1655
1656     // copy constant to device memory
1657     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1658     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1659     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1660     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1661     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1662
1663     // set kernel grid size and block size
1664     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1665     dim3 grid_ThreadPerBlock(16, 16);
1666
1667     // prolongation
1668     prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1669 //     cudaDeviceSynchronize();
1670
1671     // red-black gauss seidel relaxation (nPost times)
1672     for (int i = 0; i < nPost; i++)
1673     {
1674         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1675 //         cudaDeviceSynchronize();
1676         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1677 //         cudaDeviceSynchronize();
1678     }
1679 }
1680
1681     // down one
1682     {
1683         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 );
1684
1685         iOne = iOne * 2;
1686         jOne = jOne * 2;
1687
1688         grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1689         coef_StartPos += grid_RRow;
1690

```

```

1691         grid_RRow      = ((RRow - 1) / iOne) + 1;
1692         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1693
1694         // pre-compute constant memory
1695         h = gridSizeR * iOne;
1696         h2 = h * h;
1697         ih2 = 1.0 / h2;
1698
1699         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1700         tempRatioPhi = ratioPhi * iOne * iOne;
1701
1702
1703         // copy constant to device memory
1704         cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1705         cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1706         cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1707         cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1708         cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1709
1710         // set kernel grid size and block size
1711         dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1712         dim3 grid_ThreadPerBlock(16, 16);
1713
1714         // restriction
1715         restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
1716         //cudaDeviceSynchronize();
1717
1718         // zeroing V
1719         zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1720         //cudaDeviceSynchronize();
1721
1722         // red-black gauss seidel relaxation (nPre times)
1723         for (int i = 0; i < nPre; i++)
1724         {
1725             relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1726             //cudaDeviceSynchronize();
1727             relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1728             //cudaDeviceSynchronize();
1729         }
1730     }
1731 }
1732
1733
1734 // V-Cycle => from coarser to finer grid
1735 for (int step = (gridTo - 1); step >= gridFrom; step--)
1736 {
1737     iOne = iOne / 2;
1738     jOne = jOne / 2;
1739
1740     grid_RRow      = ((RRow - 1) / iOne) + 1;
1741     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1742
1743     grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
1744     coef_StartPos -= grid_RRow;
1745
1746     h = gridSizeR * iOne;
1747     h2 = h * h;
1748     ih2 = 1.0 / h2;
1749
1750     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1751     tempRatioPhi = ratioPhi * iOne * iOne;
1752
1753     // copy constant to device memory
1754     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1755     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1756     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1757     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1758     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1759
1760     // set kernel grid size and block size
1761     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1762     dim3 grid_ThreadPerBlock(16, 16);
1763
1764     // prolongation
1765     prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1766     //cudaDeviceSynchronize();
1767 }

```

```

1769
1770 // red-black gauss seidel relaxation (nPost times)
1771 for (int i = 0; i < nPost; i++)
1772 {
1773     relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
1774     d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1775 //     cudaDeviceSynchronize();
1776     relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
1777     d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1778 //     cudaDeviceSynchronize();
1779 }
1780 }
1781 /*V-Cycle ends*/
1782 errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev, d_VPotential,
1783 d_EpsilonError, RRow, ZColumn, PhiSlice);
1784 cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );
1785
1786 errorConv[cycle] = *EpsilonError / (RRow * ZColumn * PhiSlice);
1787
1788 if (errorConv[cycle] < convErr)
1789 {
1790     //errorConv
1791     nCycle = cycle;
1792     iparam[3] = nCycle;
1793     break;
1794 }
1795
1796 cudaMemcpy( d_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1797 cudaMemcpyDeviceToDevice );
1798 cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
1799
1800
1801 }
1802 }
1803
1804 cudaDeviceSynchronize();
1805 // copy result from device to host
1806 cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1807 cudaMemcpyDeviceToHost );
1808 memcpy( VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
1809
1810 // free device memory
1811 cudaFree( d_VPotential );
1812 cudaFree( d_VPotentialPrev );
1813 cudaFree( d_EpsilonError );
1814
1815
1816 cudaFree( d_DeltaResidue );
1817 cudaFree( d_RhoChargeDensity );
1818 cudaFree( d_coef1 );
1819 cudaFree( d_coef2 );
1820 cudaFree( d_coef3 );
1821 cudaFree( d_coef4 );
1822 cudaFree( d_icoef4 );
1823
1824 // free host memory
1825 free( coef1 );
1826 free( coef2 );
1827 free( coef3 );
1828 free( coef4 );
1829 free( icoef4 );
1830 free( temp_VPotential );
1831 //free( temp_VPotentialPrev );
1832 }

```

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.

```

680 {
681     printf("Matrix (%d,%d)\n",Row,Column);
682     for (int i=0;i<Row;i++)
683     {
684         for (int j=0;j<Column;j++)
685         {
686             printf("%11.4g ",Mat[i*Column + j]);
687         }
688         printf("\n");
689     }
690 }
691 }

```

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

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

Definisi pada baris 104 dalam file PoissonSolver3DGPU.cu.

```

115 {
116     int index_x, index_y, index, index_left, index_right, index_up, index_down, index_front, index_back,
    index_coef;
117
118     index_x = blockIdx.x * blockDim.x + threadIdx.x;
119     index_y = blockIdx.y * blockDim.y + threadIdx.y;
120
121     index      = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + index_x;
122     index_left = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x - 1);
123     index_right = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x + 1);
124     index_up    = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y - 1) * ZColumn + index_x;
125     index_down  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y + 1) * ZColumn + index_x;
126     index_front = d_grid_StartPos + ((blockIdx.z - 1 + PhiSlice) % PhiSlice) * RRow * ZColumn + index_y *
    ZColumn + index_x;
127     index_back  = d_grid_StartPos + ((blockIdx.z + 1) % PhiSlice) * RRow * ZColumn + index_y * ZColumn +
    index_x;
128     index_coef  = d_coef_StartPos + index_y;
129
130     if (index_x != 0 && index_x < (ZColumn - 1) && index_y != 0 && index_y < (RRow - 1))
131     {

```



```

132         //calculate black
133         if ((blockIdx.z % 2 == 0 && (index_x + index_y) % 2 != 0) || (blockIdx.z % 2 != 0 && (index_x +
index_y) % 2 == 0))
134         {
135             VPotential[index] = (coef2[index_coef] * VPotential[index_up] +
136             coef1[index_coef] * VPotential[index_down] +
137             d_tempRatioZ * (VPotential[index_left] + VPotential[index_right]) +
138             coef3[index_coef] * (VPotential[index_front] + VPotential[index_back]) +
139             d_h2 * RhoChargeDensity[index]) * coef4[index_coef];
140         }
141     }
142 }

```

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

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

Definisi pada baris 52 dalam file PoissonSolver3DGPU.cu.

```

63 {
64     int index_x, index_y, index, index_left, index_right, index_up, index_down, index_front, index_back,
index_coef;
65
66     index_x = blockIdx.x * blockDim.x + threadIdx.x;
67     index_y = blockIdx.y * blockDim.y + threadIdx.y;
68
69     index      = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + index_x;
70     index_left = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x - 1);
71     index_right = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x + 1);
72     index_up   = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y - 1) * ZColumn + index_x;
73     index_down = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y + 1) * ZColumn + index_x;
74     index_front = d_grid_StartPos + ((blockIdx.z - 1 + PhiSlice) % PhiSlice) * RRow * ZColumn + index_y *
ZColumn + index_x;
75     index_back  = d_grid_StartPos + ((blockIdx.z + 1) % PhiSlice) * RRow * ZColumn + index_y * ZColumn +
index_x;
76     index_coef = d_coef_StartPos + index_y;
77
78     if (index_x != 0 && index_x < (ZColumn - 1) && index_y != 0 && index_y < (RRow - 1))
79     {

```

```

80     //calculate red
81     if ((blockIdx.z % 2 == 0 && (index_x + index_y) % 2 == 0) || (blockIdx.z % 2 != 0 && (index_x +
index_y) % 2 != 0))
82     {
83         VPotential[index] = (coef2[index_coef] * VPotential[index_up] +
coef1[index_coef] * VPotential[index_down] +
84         d_tempRatioZ * (VPotential[index_left] + VPotential[index_right]) +
85         coef3[index_coef] * (VPotential[index_front] + VPotential[index_back]) +
86         d_h2 * RhoChargeDensity[index]) * coef4[index_coef];
87     }
88 }
89 }
90 }

```

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

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

Definisi pada baris 160 dalam file PoissonSolver3DGPU.cu.

```

172 {
173     int index_x, index_y, index, index_left, index_right, index_up, index_down, index_front, index_back,
index_coef;
174
175     index_x = blockIdx.x * blockDim.x + threadIdx.x;
176     index_y = blockIdx.y * blockDim.y + threadIdx.y;
177
178     index      = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + index_x;
179     index_left  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x - 1);
180     index_right = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + (index_x + 1);
181     index_up    = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y - 1) * ZColumn + index_x;
182     index_down  = d_grid_StartPos + blockIdx.z * RRow * ZColumn + (index_y + 1) * ZColumn + index_x;
183     index_front = d_grid_StartPos + ((blockIdx.z - 1 + PhiSlice) % PhiSlice) * RRow * ZColumn + index_y *
ZColumn + index_x;

```

6.3 Referensi File

/home/nfs/aswardiana/Workspace/mp-headnode/lipi/PoissonSolver3D/kernel/PoissonSolver3DGPU.cu 47

```
184     index_back = d_grid_StartPos + ((blockIdx.z + 1) % PhiSlice) * RRow * ZColumn + index_y * ZColumn +
index_x;
185     index_coef = d_coef_StartPos + index_y;
186
187     if (index_x != 0 && index_x < (ZColumn - 1) && index_y != 0 && index_y < (RRow - 1))
188     {
189         DeltaResidue[index] = d_ih2 * (coef2[index_coef] * VPotential[index_up] +
190             coef1[index_coef] * VPotential[index_down] +
191             d_tempRatioZ * (VPotential[index_left] + VPotential[index_right]) +
192             coef3[index_coef] * (VPotential[index_front] + VPotential[index_back]) -
193             icoef4[index_coef] * VPotential[index]) + RhoChargeDensity[index];
194     }
195 }
```

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) + jj] = 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_ZColumn) + jj + 1];

VPotential[Offset + sliceStart + ((RRow - 1) * ZColumn) + jj] = 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) +
(finier_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_Z
Column) + 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.

```
573 {
574     int i,ii,j,jj;
575     int finer_RRow = 2 * RRow - 1;
576     int finer_ZColumn = 2 * ZColumn - 1;
577
578     int finer_Offset = Offset - (finer_RRow * finer_ZColumn * PhiSlice);
579     int sliceStart;
580     int finer_SliceStart;
581
582     //printf("%d,%d,%d) -> (%d,%d,%d)\n",RRow,ZColumn,Offset,finer_RRow,finer_ZColumn,finer_Offset);
583     // do for each slice
584     for ( int m = 0;m < PhiSlice;m++)
585     {
586         sliceStart = m * (RRow * ZColumn);
587         finer_SliceStart = m * (finer_RRow * finer_ZColumn);
588         // copy boundary
589         for ( j = 0, jj =0; j < ZColumn ; j++,jj+=2)
590         {
591             VPotential[Offset + sliceStart + (0 * ZColumn) + j] =
592                 VPotential[finer_Offset + finer_SliceStart + (0 * finer_ZColumn) + jj];
593
594             VPotential[Offset + sliceStart + ((RRow - 1) * ZColumn) + j] =
595                 VPotential[finer_Offset + finer_SliceStart + ((finer_RRow -1) * finer_ZColumn) + jj];
596
597         }
598         for ( i = 0, ii =0; i < RRow ; i++,ii+=2) {
599             VPotential[Offset + sliceStart + (i * ZColumn)] =
600                 VPotential[finer_Offset + finer_SliceStart + (ii * finer_ZColumn)];
601
602             VPotential[Offset + sliceStart + (i * ZColumn) + (ZColumn - 1)] =
603                 VPotential[finer_Offset + finer_SliceStart + (ii * finer_ZColumn) + (finer_ZColumn - 1)];
604
605         }
606     }
668 }
```

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

010RhoChargeDensityfloat*ArrayrapatarusDeltaResiduefloat*ArrayresiduhasilrelaksasiRRowconstintJumlahbar
r ZColumnconstintJumlahkolomdiarahsumbu z PhiSliceconstintJumlahirisandiarahsumbu_global_voidrestriction2DHalf(float

$I_h^{2h} = \{1\}\{16\} \{bmatrix\}[ccc] 1 \& 2 \& 1 \setminus 2 \& 4 \& 2 \setminus 1 \& 2 \& 1 \{bmatrix\}$

Parameter

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

Definisi pada baris 253 dalam file PoissonSolver3DGPU.cu.

```

260 {
261     int index_x, index_y, index;
262     int finer_RRow, finer_ZColumn, finer_grid_StartPos;
263     int finer_index_x, finer_index_y, finer_index, finer_index_left, finer_index_right, finer_index_up,
    finer_index_down;
264     int finer_index_up_left, finer_index_up_right, finer_index_down_left, finer_index_down_right;
265
266     index_x = blockIdx.x * blockDim.x + threadIdx.x;
267     index_y = blockIdx.y * blockDim.y + threadIdx.y;
268     index = d_grid_StartPos + blockIdx.z * RRow * ZColumn + index_y * ZColumn + index_x;
269
270     finer_RRow = 2 * RRow - 1;
271     finer_ZColumn = 2 * ZColumn - 1;
272
273     finer_grid_StartPos = d_grid_StartPos - (finer_RRow * finer_ZColumn * PhiSlice);
274
275     finer_index_x = index_x * 2;
276     finer_index_y = index_y * 2;
277
278     finer_index = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + finer_index_y *
    finer_ZColumn + finer_index_x;
279     finer_index_left = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + finer_index_y *
    finer_ZColumn + (finer_index_x - 1);
280     finer_index_right = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + finer_index_y *
    finer_ZColumn + (finer_index_x + 1);
281     finer_index_up = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (finer_index_y -
    1) * finer_ZColumn + finer_index_x;
282     finer_index_down = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (finer_index_y +
    1) * finer_ZColumn + finer_index_x;
283     finer_index_up_left = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (
    finer_index_y - 1) * finer_ZColumn + (finer_index_x - 1);
284     finer_index_up_right = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (
    finer_index_y - 1) * finer_ZColumn + (finer_index_x + 1);
285     finer_index_down_left = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (
    finer_index_y + 1) * finer_ZColumn + (finer_index_x - 1);
286     finer_index_down_right = finer_grid_StartPos + blockIdx.z * finer_RRow * finer_ZColumn + (
    finer_index_y + 1) * finer_ZColumn + (finer_index_x + 1);
287
288     if (index_x != 0 && index_x < (ZColumn - 1) && index_y != 0 && index_y < (RRow - 1))
289     {
290         RhoChargeDensity[index] = 0.25 * DeltaResidue[finer_index] +
291             0.125 * (DeltaResidue[finer_index_left] + DeltaResidue[finer_index_right] +
    DeltaResidue[finer_index_up] + DeltaResidue[finer_index_down]) +
292             0.0625 * (DeltaResidue[finer_index_up_left] + DeltaResidue[
    finer_index_up_right] + DeltaResidue[finer_index_down_left] + DeltaResidue[finer_index_down_right]);
293     } else {
294         RhoChargeDensity[index] = DeltaResidue[finer_index];
295     }
296
297
298 }

```

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

Berkas ini berisi definisi fungsi extern yang memiliki 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 RRow, const int ZColumn, const int PhiSlice, const int Symmetry, float *fparam, int *iparam, bool isExactPresent, 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 isExactPresent, float *errorConv, float *errorExact, float *VPotentialExact)

6.4.1 Keterangan Lengkap

Berkas ini berisi definisi fungsi extern yang memiliki 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	<i>VPotential</i>	float[nrows*ncols*nphi] distribusi potensial. Input: hanya nilai tepi. Output: hasil perhitungan penyelesaian persamaan Poisson
in	<i>RhoChangeDensity</i>	float[nrows*ncols*nphi] distribusi muatan listrik.

Mengembalikan

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

Definisi pada baris 893 dalam file PoissonSolver3DGPU.cu.

```

907 {
908     // variables for CPU memory
909     float *temp_VPotential;
910     float *VPotentialPrev;
911     float *EpsilonError;
912
913     // variables for GPU memory
914     float *d_VPotential;
915     float *d_RhoChargeDensity;
916     float *d_DeltaResidue;
917     float *d_VPotentialPrev;
918     float *d_EpsilonError;
919
920     float *d_coef1;
921     float *d_coef2;
922     float *d_coef3;
923     float *d_coef4;
924     float *d_icoef4;
925
926     // variables for coefficient calculations
927     float *coef1;
928     float *coef2;
929     float *coef3;
930     float *coef4;
931     float *icoef4;
932     float tempRatioZ;
933     float tempRatioPhi;
934     float radius;
935
936     int gridFrom;
937     int gridTo;
938     int loops;
939
940
941     // variables passed from ALIROOT
942     float gridSizeR      = fparam[0];
943     float gridSizePhi   = fparam[1];
944     float gridSizeZ     = fparam[2];
945     float ratioPhi      = fparam[3];
946     float ratioZ        = fparam[4];
947     float convErr       = fparam[5];
948     float IFCRadius     = fparam[6];
949     int nPre             = iparam[0];
950     int nPost            = iparam[1];
951     int maxLoop          = iparam[2];
952     int nCycle           = iparam[3];
953
954     // variables for calculating GPU memory allocation
955     int grid_RRow;
956     int grid_ZColumn;
957     int grid_PhiSlice = PhiSlice;
958     int grid_Size = 0;
959     int grid_StartPos;
960     int coef_Size = 0;
961     int coef_StartPos;
962     int iOne, jOne;
963     float h, h2, ih2;
964
965     // variables for calculating multigrid maximum depth
966     int depth_RRow = 0;
967     int depth_ZColumn = 0;
968     int temp_RRow = RRow;
969     int temp_ZColumn = ZColumn;
970
971     // calculate depth for multigrid

```

```

972 while (temp_RRow >>= 1) depth_RRow++;
973 while (temp_ZColumn >>= 1) depth_ZColumn++;
974
975 loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
976 loops = (loops > maxLoop) ? maxLoop : loops;
977
978 gridFrom = 1;
979 gridTo = loops;
980
981 // calculate GPU memory allocation for multigrid
982 for (int step = gridFrom; step <= gridTo; step++)
983 {
984     grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
985     grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;
986
987     grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
988     coef_Size += grid_RRow;
989 }
990
991 // allocate memory for temporary output
992 temp_VPotential = (float *) malloc(grid_Size * sizeof(float));
993 VPotentialPrev = (float *) malloc(RRow * ZColumn * PhiSlice * sizeof(float));
994 EpsilonError = (float *) malloc(1 * sizeof(float));
995
996
997 // allocate memory for relaxation coefficient
998 coef1 = (float *) malloc(coef_Size * sizeof(float));
999 coef2 = (float *) malloc(coef_Size * sizeof(float));
1000 coef3 = (float *) malloc(coef_Size * sizeof(float));
1001 coef4 = (float *) malloc(coef_Size * sizeof(float));
1002 icoef4 = (float *) malloc(coef_Size * sizeof(float));
1003
1004 // pre-compute relaxation coefficient
1005 coef_StartPos = 0;
1006 iOne = 1 << (gridFrom - 1);
1007 jOne = 1 << (gridFrom - 1);
1008
1009 for (int step = gridFrom; step <= gridTo; step++)
1010 {
1011     grid_RRow = ((RRow - 1) / iOne) + 1;
1012
1013     h = gridSizeR * iOne;
1014     h2 = h * h;
1015     ih2 = 1.0 / h2;
1016
1017     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1018     tempRatioPhi = ratioPhi * iOne * iOne;
1019
1020     for (int i = 1; i < grid_RRow - 1; i++)
1021     {
1022         radius = IFCRadius + i * h;
1023         coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
1024         coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
1025         coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
1026         coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
1027         icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
1028     }
1029     coef_StartPos += grid_RRow;
1030     iOne = 2 * iOne;
1031     jOne = 2 * jOne;
1032 }
1033
1034 // device memory allocation
1035 cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
1036 cudaMalloc( &d_VPotentialPrev, RRow * ZColumn * PhiSlice * sizeof(float) );
1037 cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );
1038 cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
1039 cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
1040 cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
1041 cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
1042 cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
1043 cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
1044 cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );
1045
1046 // set memory to zero
1047 cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
1048 cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
1049 cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
1050 cudaMemset( d_VPotentialPrev, 0, RRow * ZColumn * PhiSlice * sizeof(float) );
1051 cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
1052
1053
1054 // copy data from host to device
1055 cudaMemcpy( d_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice ); //check
1056 cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice ); //check

```



```

1057     cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1058     cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1059     cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1060     cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1061     cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1062     cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyHostToDevice );
1063
1064     // max exact
1065
1066     // float maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
1067     float maxAbsExact = 1.0;
1068
1069     if (isExactPresent == true)
1070         maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
1071     dim3 error_BlockPerGrid((RRow < 16) ? 1 : (RRow / 16), (ZColumn < 16) ? 1 : (ZColumn / 16), PhiSlice);
1072     dim3 error_ThreadPerBlock(16, 16);
1073
1074
1075     for (int cycle = 0; cycle < nCycle; cycle++)
1076     {
1077         cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
1078         if (isExactPresent == true) errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact,
RRow * PhiSlice, ZColumn, maxAbsExact);
1079
1080
1081         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);
1082
1083
1084         errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev, d_VPotential,
d_EpsilonError, RRow, ZColumn, PhiSlice);
1085
1086         cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );
1087
1088
1089         errorConv[cycle] = *EpsilonError / (RRow * ZColumn * PhiSlice);
1090
1091         if (errorConv[cycle] < convErr)
1092         {
1093             //errorConv
1094             nCycle = cycle;
1095             break;
1096         }
1097
1098         cudaMemcpy( d_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToDevice );
1099         cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
1100
1101     }
1102     iparam[3] = nCycle;
1103
1104 // for (int cycle = 0; cycle < nCycle; cycle++)
1105 // {
1106 //     cudaMemcpy( temp_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
1107
1108
1109 //     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);
1110
1111 //     cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
1112 //     errorConv[cycle] = GetErrorNorm2(temp_VPotential, temp_VPotentialPrev, RRow * PhiSlice, ZColumn,
1.0);
1113 //     //errorExact[cycle] = GetErrorNorm2(temp_VPotential, VPotentialExact, RRow * PhiSlice, ZColumn,
1.0);
1114 // }
1115
1116
1117 // copy result from device to host
1118     cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
1119
1120     memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
1121
1122     // free device memory
1123     cudaFree( d_VPotential );
1124     cudaFree( d_DeltaResidue );
1125     cudaFree( d_RhoChargeDensity );
1126     cudaFree( d_VPotentialPrev );
1127     cudaFree( d_EpsilonError );
1128     cudaFree( d_coef1 );
1129     cudaFree( d_coef2 );
1130     cudaFree( d_coef3 );
1131     cudaFree( d_coef4 );

```

```

1132     cudaFree( d_icoef4 );
1133
1134     // free host memory
1135     free( coef1 );
1136     free( coef2 );
1137     free( coef3 );
1138     free( coef4 );
1139     free( icoef4 );
1140     free( temp_VPotential );
1141     free( VPotentialPrev );
1142 }

```

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.

```

1851 {
1852     // variables for CPU memory
1853     float *temp_VPotential;
1854     float *VPotentialPrev;
1855     float *EpsilonError;
1856
1857     // variables for GPU memory
1858     float *d_VPotential;
1859     float *d_RhoChargeDensity;
1860     float *d_DeltaResidue;
1861     float *d_coef1;
1862     float *d_coef2;
1863     float *d_coef3;
1864     float *d_coef4;
1865     float *d_icoef4;
1866     float *d_VPotentialPrev;
1867     float *d_EpsilonError;
1868
1869
1870     // variables for coefficient calculations
1871     float *coef1;
1872     float *coef2;
1873     float *coef3;
1874     float *coef4;
1875     float *icoef4;
1876     float tempRatioZ;
1877     float tempRatioPhi;
1878     float radius;
1879
1880     int gridFrom;
1881     int gridTo;
1882     int loops;
1883
1884     // variables passed from ALIROOT
1885     float gridSizeR = fparam[0];
1886     //float gridSizePhi = fparam[1];
1887     //float gridSizeZ = fparam[2];
1888     float ratioPhi = fparam[3];

```

```

1889     float ratioZ      = fparam[4];
1890     float convErr     = fparam[5];
1891     float IFCRadius   = fparam[6];
1892     int nPre          = iparam[0];
1893     int nPost         = iparam[1];
1894     int maxLoop       = iparam[2];
1895     int nCycle        = iparam[3];
1896
1897     // variables for calculating GPU memory allocation
1898     int grid_RRow;
1899     int grid_ZColumn;
1900     int grid_PhiSlice = PhiSlice;
1901     int grid_Size = 0;
1902     int grid_StartPos;
1903     int coef_Size = 0;
1904     int coef_StartPos;
1905     int iOne, jOne;
1906     float h, h2, ih2;
1907
1908     // variables for calculating multigrid maximum depth
1909     int depth_RRow = 0;
1910     int depth_ZColumn = 0;
1911     int temp_RRow = RRow;
1912     int temp_ZColumn = ZColumn;
1913
1914     // calculate depth for multigrid
1915     while (temp_RRow >= 1) depth_RRow++;
1916     while (temp_ZColumn >= 1) depth_ZColumn++;
1917
1918     loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
1919     loops = (loops > maxLoop) ? maxLoop : loops;
1920
1921     gridFrom = 1;
1922     gridTo = loops;
1923
1924     // calculate GPU memory allocation for multigrid
1925     for (int step = gridFrom; step <= gridTo; step++)
1926     {
1927         grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
1928         grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;
1929
1930         grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
1931         coef_Size += grid_RRow;
1932     }
1933
1934     // allocate memory for temporary output
1935     temp_VPotential = (float *) malloc(grid_Size * sizeof(float));
1936     VPotentialPrev = (float *) malloc(grid_Size * sizeof(float));
1937     EpsilonError = (float *) malloc(1 * sizeof(float));
1938
1939
1940
1941     for (int i=0;i<grid_Size;i++) temp_VPotential[i] = 0.0;
1942
1943
1944     // allocate memory for relaxation coefficient
1945     coef1 = (float *) malloc(coef_Size * sizeof(float));
1946     coef2 = (float *) malloc(coef_Size * sizeof(float));
1947     coef3 = (float *) malloc(coef_Size * sizeof(float));
1948     coef4 = (float *) malloc(coef_Size * sizeof(float));
1949     icoef4 = (float *) malloc(coef_Size * sizeof(float));
1950
1951     // pre-compute relaxation coefficient
1952     // restrict boundary
1953     coef_StartPos = 0;
1954     grid_StartPos = 0;
1955
1956     iOne = 1 << (gridFrom - 1);
1957     jOne = 1 << (gridFrom - 1);
1958
1959     for (int step = gridFrom; step <= gridTo; step++)
1960     {
1961         grid_RRow = ((RRow - 1) / iOne) + 1;
1962         grid_ZColumn = ((ZColumn - 1) / iOne) + 1;
1963
1964         h = gridSizeR * iOne;
1965         h2 = h * h;
1966         ih2 = 1.0 / h2;
1967
1968         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1969         tempRatioPhi = ratioPhi * iOne * iOne;
1970
1971         for (int i = 1; i < grid_RRow - 1; i++)
1972         {
1973             radius = IFCRadius + i * h;
1974             coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
1975             coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);

```

```

1976         coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
1977         coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
1978         icoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
1979     }
1980
1981     // call restrict boundary
1982     if (step == gridFrom) {
1983         // Copy original VPotential to tempPotential
1984         memcpy(temp_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
1985     }
1986     // else
1987     //{
1988     // Restrict_Boundary(temp_VPotential, grid_RRow, grid_ZColumn, PhiSlice, grid_StartPos);
1989     //}
1990
1991
1992     coef_StartPos += grid_RRow;
1993     grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1994
1995
1996     iOne = 2 * iOne;
1997     jOne = 2 * jOne;
1998 }
1999
2000 // device memory allocation
2001 cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
2002 cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
2003 cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
2004 cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
2005 cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
2006 cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
2007 cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
2008 cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );
2009 cudaMalloc( &d_VPotentialPrev, grid_Size * sizeof(float) );
2010 cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );
2011
2012 // set memory to zero
2013
2014 cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
2015 cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
2016 cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
2017 cudaMemset( d_VPotentialPrev, 0, grid_Size * sizeof(float) );
2018 cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
2019
2020 // set memory to zero
2021
2022 cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
2023 cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
2024 cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
2025
2026 // copy data from host to devicei
2027 // case of FCycle you need to copy all boundary for all
2028 cudaMemcpy( d_VPotential, temp_VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice ); //check
2029 // cudaMemcpy( d_VPotential, VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice ); //check
2030
2031 cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float),
2032 cudaMemcpyHostToDevice ); //check
2033 // cudaMemcpy( d_RhoChargeDensity, temp_VPotentialPrev, grid_Size * sizeof(float), cudaMemcpyHostToDevice
2034 ); //check
2035 cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2036 cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2037 cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2038 cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2039 // cudaMemcpy( d_icoef4, icoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
2040 // cudaMemcpy( d_VPotentialPrev, temp_VPotential, grid_Size * sizeof(float), cudaMemcpyHostToDevice );
2041
2042 // max exact
2043
2044 float maxAbsExact = 1.0;
2045
2046 if (isExactPresent == true)
2047     maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
2048
2049
2050 // init iOne,grid_RRow, grid_ZColumn, grid_StartPos, coef_StartPos
2051 iOne = 1 << (gridFrom - 1);
2052 jOne = 1 << (gridFrom - 1);
2053
2054 grid_RRow = ((RRow - 1) / iOne) + 1;
2055 grid_ZColumn = ((ZColumn - 1) / jOne) + 1;
2056
2057 grid_StartPos = 0;
2058 coef_StartPos = 0;

```

```

2060
2061
2063     for (int step = gridFrom + 1; step <= gridTo; step++)
2064     {
2065
2066         iOne = 1 << (step - 1);
2067         jOne = 1 << (step - 1);
2068
2069         grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
2070         coef_StartPos += grid_RRow;
2071
2072         grid_RRow      = ((RRow - 1) / iOne) + 1;
2073         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
2074
2075         // pre-compute constant memory
2076         h      = gridSizeR * iOne;
2077         h2     = h * h;
2078         ih2    = 1.0 / h2;
2079
2080         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
2081         tempRatioPhi = ratioPhi * iOne * iOne;
2082
2083         // copy constant to device memory
2084         cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
2085         cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
2086         cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2087         cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2088         cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2089
2090         // set kernel grid size and block size
2091         dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
2092         dim3 grid_ThreadPerBlock(16, 16);
2093
2094         // restriction
2095         restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice );
2096
2097         // restrict boundary (already done in cpu)
2099 //     PrintMatrix(temp_VPotential,grid_RRow * PhiSlice,grid_ZColumn);
2100         restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_VPotential,
grid_RRow, grid_ZColumn, grid_PhiSlice );
2101
2102     }
2103
2104
2105     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (grid_ZColumn
/ 16), PhiSlice);
2106     dim3 grid_ThreadPerBlock(16, 16);
2107
2108
2109     // relax on the coarsest
2110     // red-black gauss seidel relaxation (nPre times)
2111 //     printf("rho\n");
2112 //     cudaMemcpy( temp_VPotential, d_RhoChargeDensity + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
sizeof(float), cudaMemcpyDeviceToHost );
2113 //     PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);
2114
2115 //     printf("v\n");
2116 //     cudaMemcpy( temp_VPotential, d_VPotential + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
sizeof(float), cudaMemcpyDeviceToHost );
2117 //     PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);
2118     for (int i = 0; i < nPre; i++)
2119     {
2120         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
2121         cudaDeviceSynchronize();
2122         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
2123         cudaDeviceSynchronize();
2124     }
2125
2126 //     printf("v after relax\n");
2127 //     cudaMemcpy( temp_VPotential, d_VPotential + grid_StartPos , grid_RRow * grid_ZColumn * PhiSlice *
sizeof(float), cudaMemcpyDeviceToHost );
2128 //     PrintMatrix(temp_VPotential,grid_RRow,grid_ZColumn);
2129
2130     // V-Cycle => from coarser to finer grid
2131     for (int step = gridTo - 1 ; step >= gridFrom; step--)
2132     {
2133         iOne = iOne / 2;
2134         jOne = jOne / 2;
2135
2136         grid_RRow      = ((RRow - 1) / iOne) + 1;
2137         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
2138
2139         grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;

```

```

2140     coef_StartPos -= grid_RRow;
2141
2142     h   = gridSizeR * iOne;
2143     h2  = h * h;
2144     ih2 = 1.0 / h2;
2145
2146     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
2147     tempRatioPhi = ratioPhi * iOne * iOne;
2148
2149     // copy constant to device memory
2150     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
2151     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
2152     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2153     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2154     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
2155
2156
2157
2158     // set kernel grid size and block size
2159     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
2160     dim3 grid_ThreadPerBlock(16, 16);
2161
2162
2163     prolongation2DHalfNoAdd<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
2164
2165
2166
2167
2168     // just
2169
2170     // max exact
2171     cudaMemcpy( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, grid_RRow *
grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );
2172
2173     float maxAbsExact = 1.0;
2174
2175     if (isExactPresent == true)
2176         maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
2177     dim3 error_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
2178     dim3 error_ThreadPerBlock(16, 16);
2179
2180
2181
2182     for (int cycle = 0; cycle < nCycle; cycle++)
2183     {
2184
2185
2186         if (step == gridFrom) {
2187             cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
2188             if (isExactPresent == true) errorExact[cycle] = GetErrorNorm2(temp_VPotential,
VPotentialExact, RRow * PhiSlice, ZColumn, maxAbsExact);
2189         }
2190
2191
2192
2193         //cudaDeviceSynchronize();
2194         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);
2195
2196
2197
2198         //if (step == gridFrom) {
2199             //cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToHost );
2200
2201             //errorConv[cycle] = GetErrorNorm2(temp_VPotential, VPotentialPrev, RRow *
PhiSlice, ZColumn, 1.0);
2202
2203             errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev +
grid_StartPos, d_VPotential + grid_StartPos, d_EpsilonError, grid_RRow, grid_ZColumn, PhiSlice);
2204
2205             cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );
2206
2207             errorConv[cycle] = *EpsilonError / (grid_RRow * grid_ZColumn * PhiSlice);
2208
2209             if (errorConv[cycle] < convErr)
2210             {
2211                 nCycle = cycle;
2212                 break;
2213             }
2214
2215             cudaMemcpy( d_VPotentialPrev + grid_StartPos, d_VPotential + grid_StartPos, grid_RRow *
grid_ZColumn * PhiSlice * sizeof(float), cudaMemcpyDeviceToDevice );

```

```

2216         cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
2217     }
2218 }
2219
2220 }
2221
2222     iparam[3] = nCycle;
2223
2224
2225     // copy result from device to host
2226     cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
                cudaMemcpyDeviceToHost );
2227
2228     memcpy( VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
2229
2230     // free device memory
2231     cudaFree( d_VPotential );
2232     cudaFree( d_DeltaResidue );
2233     cudaFree( d_RhoChargeDensity );
2234     cudaFree( d_coef1 );
2235     cudaFree( d_coef2 );
2236     cudaFree( d_coef3 );
2237     cudaFree( d_coef4 );
2238     cudaFree( d_icoef4 );
2239
2240     // free host memory
2241     free( coef1 );
2242     free( coef2 );
2243     free( coef3 );
2244     free( coef4 );
2245     free( icoef4 );
2246     free( temp_VPotential );
2247     free( VPotentialPrev );
2248 }

```

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 )

```

inner 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 coefficient 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
1217     // variables for calculating multigrid maximum depth
1218     int depth_RRow = 0;
1219     int depth_ZColumn = 0;
1220     int temp_RRow = RRow;
1221     int temp_ZColumn = ZColumn;
1222
1223     // calculate depth for multigrid
1224     while (temp_RRow >>= 1) depth_RRow++;
1225     while (temp_ZColumn >>= 1) depth_ZColumn++;
1226
1227     loops = (depth_RRow > depth_ZColumn) ? depth_ZColumn : depth_RRow;
1228     loops = (loops > maxLoop) ? maxLoop : loops;
1229
1230     gridFrom = 1;
1231     gridTo = loops;
1232
1233     // calculate GPU memory allocation for multigrid
1234     for (int step = gridFrom; step <= gridTo; step++)
1235     {
1236         grid_RRow = ((RRow - 1) / (1 << (step - 1))) + 1;
1237         grid_ZColumn = ((ZColumn - 1) / (1 << (step - 1))) + 1;
1238
1239         grid_Size += grid_RRow * grid_ZColumn * grid_PhiSlice;
1240         coef_Size += grid_RRow;
1241     }
1242
1243     // allocate memory for temporary output
1244     temp_VPotential = (float *) malloc(grid_Size * sizeof(float));
1245     VPotentialPrev = (float *) malloc(RRow * ZColumn * PhiSlice * sizeof(float));
1246     EpsilonError = (float *) malloc(1 * sizeof(float));
1247
1248
1249     // allocate memory for relaxation coefficient
1250     coef1 = (float *) malloc(coef_Size * sizeof(float));
1251     coef2 = (float *) malloc(coef_Size * sizeof(float));
1252     coef3 = (float *) malloc(coef_Size * sizeof(float));
1253     coef4 = (float *) malloc(coef_Size * sizeof(float));

```



```

1254     iccoef4 = (float *) malloc(coef_Size * sizeof(float));
1255
1256     // pre-compute relaxation coefficient
1257     coef_StartPos = 0;
1258     iOne = 1 << (gridFrom - 1);
1259     jOne = 1 << (gridFrom - 1);
1260
1261     for (int step = gridFrom; step <= gridTo; step++)
1262     {
1263         grid_RRow = ((RRow - 1) / iOne) + 1;
1264
1265         h = gridSizeR * iOne;
1266         h2 = h * h;
1267         ih2 = 1.0 / h2;
1268
1269         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1270         tempRatioPhi = ratioPhi * iOne * iOne;
1271
1272         for (int i = 1; i < grid_RRow - 1; i++)
1273         {
1274             radius = IFCRadius + i * h;
1275             coef1[coef_StartPos + i] = 1.0 + h / (2 * radius);
1276             coef2[coef_StartPos + i] = 1.0 - h / (2 * radius);
1277             coef3[coef_StartPos + i] = tempRatioPhi / (radius * radius);
1278             coef4[coef_StartPos + i] = 0.5 / (1.0 + tempRatioZ + coef3[coef_StartPos + i]);
1279             iccoef4[coef_StartPos + i] = 1.0 / coef4[coef_StartPos + i];
1280         }
1281         coef_StartPos += grid_RRow;
1282         iOne = 2 * iOne;
1283         jOne = 2 * jOne;
1284     }
1285
1286     // device memory allocation
1287     cudaMalloc( &d_VPotential, grid_Size * sizeof(float) );
1288     cudaMalloc( &d_DeltaResidue, grid_Size * sizeof(float) );
1289     cudaMalloc( &d_VPotentialPrev, RRow * ZColumn * PhiSlice * sizeof(float) );
1290     cudaMalloc( &d_EpsilonError, 1 * sizeof(float) );
1291
1292     cudaMalloc( &d_RhoChargeDensity, grid_Size * sizeof(float) );
1293     cudaMalloc( &d_coef1, coef_Size * sizeof(float) );
1294     cudaMalloc( &d_coef2, coef_Size * sizeof(float) );
1295     cudaMalloc( &d_coef3, coef_Size * sizeof(float) );
1296     cudaMalloc( &d_coef4, coef_Size * sizeof(float) );
1297     cudaMalloc( &d_icoef4, coef_Size * sizeof(float) );
1298
1299     // set memory to zero
1300     cudaMemset( d_VPotential, 0, grid_Size * sizeof(float) );
1301     cudaMemset( d_DeltaResidue, 0, grid_Size * sizeof(float) );
1302     cudaMemset( d_RhoChargeDensity, 0, grid_Size * sizeof(float) );
1303     cudaMemset( d_VPotentialPrev, 0, RRow * ZColumn * PhiSlice * sizeof(float) );
1304     cudaMemset( d_EpsilonError, 0, 1 * sizeof(float) );
1305
1306
1307     // copy data from host to device
1308     cudaMemcpy( d_VPotential, VPotential, RRow * ZColumn * PhiSlice * sizeof(float), cudaMemcpyHostToDevice
1309 ); //check
1310     cudaMemcpy( d_RhoChargeDensity, RhoChargeDensity, RRow * ZColumn * PhiSlice * sizeof(float),
1311 cudaMemcpyHostToDevice ); //check
1312     cudaMemcpy( d_coef1, coef1, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1313     cudaMemcpy( d_coef2, coef2, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1314     cudaMemcpy( d_coef3, coef3, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1315     cudaMemcpy( d_coef4, coef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1316     cudaMemcpy( d_icoef4, iccoef4, coef_Size * sizeof(float), cudaMemcpyHostToDevice );
1317     cudaMemcpy( d_VPotentialPrev, VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1318 cudaMemcpyHostToDevice );
1319
1320     // max exact float maxAbsExact = GetAbsMax(VPotentialExact,RRow * PhiSlice * ZColumn);
1321     float maxAbsExact = GetAbsMax(VPotentialExact, RRow * PhiSlice * ZColumn);
1322     dim3 error_BlockPerGrid((RRow < 16) ? 1 : (RRow / 16), (ZColumn < 16) ? 1 : (ZColumn / 16), PhiSlice);
1323     dim3 error_ThreadPerBlock(16, 16);
1324
1325     for (int cycle = 0; cycle < nCycle; cycle++)
1326     {
1327         /*V-Cycle starts*/
1328
1329         // error conv
1330         // cudaMemcpy( temp_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1331 cudaMemcpyDeviceToHost );
1332
1333         cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1334 cudaMemcpyDeviceToHost );
1335         errorExact[cycle] = GetErrorNorm2(temp_VPotential,VPotentialExact,RRow * PhiSlice,ZColumn,
1336 maxAbsExact);
1337
1338     }
1339
1340     // V-Cycle => Finest Grid

```

```

1335     iOne = 1 << (gridFrom - 1);
1336     jOne = 1 << (gridFrom - 1);
1337
1338     grid_RRow      = ((RRow - 1) / iOne) + 1;
1339     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1340
1341     grid_StartPos = 0;
1342     coef_StartPos = 0;
1343
1344     // pre-compute constant memory
1345     h = gridSizeR * iOne;
1346     h2 = h * h;
1347     ih2 = 1.0 / h2;
1348
1349     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1350     tempRatioPhi = ratioPhi * iOne * iOne;
1351
1352     // copy constant to device memory
1353     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
1354     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice );
1355     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1356     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1357     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1358
1359     // set kernel grid size and block size
1360     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1361     dim3 grid_ThreadPerBlock(16, 16);
1362
1363     // red-black gauss seidel relaxation (nPre times)
1364     for (int i = 0; i < nPre; i++)
1365     {
1366         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1367         //cudaDeviceSynchronize();
1368         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1369         //cudaDeviceSynchronize();
1370     }
1371
1372     // residue calculation
1373     residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, d_RhoChargeDensity,
d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1374     //cudaDeviceSynchronize();
1375
1376     // V-Cycle => from finer to coarsest grid
1377     for (int step = gridFrom + 1; step <= gridTo; step++)
1378     {
1379         iOne = 1 << (step - 1);
1380         jOne = 1 << (step - 1);
1381
1382         grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1383         coef_StartPos += grid_RRow;
1384
1385         grid_RRow      = ((RRow - 1) / iOne) + 1;
1386         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1387
1388         // pre-compute constant memory
1389         h = gridSizeR * iOne;
1390         h2 = h * h;
1391         ih2 = 1.0 / h2;
1392
1393         tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1394         tempRatioPhi = ratioPhi * iOne * iOne;
1395
1396         // copy constant to device memory
1397         cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1398         cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1399         cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1400         cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1401         cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1402
1403         // set kernel grid size and block size
1404         dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1405         dim3 grid_ThreadPerBlock(16, 16);
1406
1407         // restriction
1408         restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
1409         //cudaDeviceSynchronize();
1410
1411         // zeroing V
1412         zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );

```

```

1413         //cudaDeviceSynchronize();
1414
1415         // red-black gauss seidel relaxation (nPre times)
1416         for (int i = 0; i < nPre; i++)
1417         {
1418             relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1419             //cudaDeviceSynchronize();
1420             relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1421             //cudaDeviceSynchronize();
1422         }
1423
1424         // residue calculation
1425         if (step < gridTo)
1426         {
1427             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 );
1428             //cudaDeviceSynchronize();
1429         }
1430     }
1431 }
1432
1433 // up one
1434
1435 {
1436     int step = (gridTo - 1);
1437     iOne = iOne / 2;
1438     jOne = jOne / 2;
1439
1440     grid_RRow      = ((RRow - 1) / iOne) + 1;
1441     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1442
1443     grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
1444     coef_StartPos -= grid_RRow;
1445
1446     h      = gridSizeR * iOne;
1447     h2     = h * h;
1448     ih2    = 1.0 / h2;
1449
1450     tempRatioPhi = ratioPhi * iOne * iOne;
1451
1452     // copy constant to device memory
1453     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1454     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1455     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1456     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1457     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1458
1459     // set kernel grid size and block size
1460     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1461     dim3 grid_ThreadPerBlock(16, 16);
1462
1463     // prolongation
1464     prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1465     // cudaDeviceSynchronize();
1466
1467     // red-black gauss seidel relaxation (nPost times)
1468     for (int i = 0; i < nPost; i++)
1469     {
1470         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1471         // cudaDeviceSynchronize();
1472         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1473         // cudaDeviceSynchronize();
1474     }
1475
1476     // down one
1477     {
1478         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 );
1479
1480         iOne = iOne * 2;
1481         jOne = jOne * 2;
1482
1483         grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1484         coef_StartPos += grid_RRow;
1485
1486         grid_RRow      = ((RRow - 1) / iOne) + 1;
1487         grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;

```

```

1492
1493 // pre-compute constant memory
1494 h = gridSizeR * iOne;
1495 h2 = h * h;
1496 ih2 = 1.0 / h2;
1497
1498 tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1499 tempRatioPhi = ratioPhi * iOne * iOne;
1500
1501
1502 // copy constant to device memory
1503 cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1504 cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1505 cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1506 cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1507 cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1508
1509 // set kernel grid size and block size
1510 dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1511 dim3 grid_ThreadPerBlock(16, 16);
1512
1513 // restriction
1514 restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
1515 //cudaDeviceSynchronize();
1516
1517 // zeroing V
1518 zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1519 //cudaDeviceSynchronize();
1520
1521 // red-black gauss seidel relaxation (nPre times)
1522 for (int i = 0; i < nPre; i++)
1523 {
1524 relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1525 //cudaDeviceSynchronize();
1526 relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1527 //cudaDeviceSynchronize();
1528 }
1529
1530 }
1531
1532 // up two from gridTo - 1, to gridTo -3
1533 for (int step = (gridTo - 1); step >= gridTo - 3; step--)
1534 {
1535 iOne = iOne / 2;
1536 jOne = jOne / 2;
1537
1538 grid_RRow = ((RRow - 1) / iOne) + 1;
1539 grid_ZColumn = ((ZColumn - 1) / jOne) + 1;
1540
1541 grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
1542 coef_StartPos -= grid_RRow;
1543
1544 h = gridSizeR * iOne;
1545 h2 = h * h;
1546 ih2 = 1.0 / h2;
1547
1548 tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1549 tempRatioPhi = ratioPhi * iOne * iOne;
1550
1551 // copy constant to device memory
1552 cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1553 cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1554 cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1555 cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1556 cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1557
1558 // set kernel grid size and block size
1559 dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1560 dim3 grid_ThreadPerBlock(16, 16);
1561
1562 // prolongation
1563 prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1564 // cudaDeviceSynchronize();
1565
1566 // red-black gauss seidel relaxation (nPost times)
1567 for (int i = 0; i < nPost; i++)

```

```

1570     {
1571         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1572 //         cudaDeviceSynchronize();
1573         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1574 //         cudaDeviceSynchronize();
1575     }
1576 }
1577
1578 // down to from gridTo - 1, to gridTo -3
1579 for (int step = gridTo - 3; step <= gridTo - 1; step++)
1580 {
1581     iOne = iOne * 2;
1582     jOne = jOne * 2;
1583
1584     grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1585     coef_StartPos += grid_RRow;
1586
1587     grid_RRow      = ((RRow - 1) / iOne) + 1;
1588     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1589
1590     // pre-compute constant memory
1591     h      = gridSizeR * iOne;
1592     h2     = h * h;
1593     ih2    = 1.0 / h2;
1594
1595     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1596     tempRatioPhi = ratioPhi * iOne * iOne;
1597
1598     // copy constant to device memory
1599     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1600     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1601     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1602     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1603     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1604
1605     // set kernel grid size and block size
1606     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1607     dim3 grid_ThreadPerBlock(16, 16);
1608
1609     // restriction
1610     restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
1611     //cudaDeviceSynchronize();
1612
1613     // zeroing V
1614     zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1615     //cudaDeviceSynchronize();
1616
1617     // red-black gauss seidel relaxation (nPre times)
1618     for (int i = 0; i < nPre; i++)
1619     {
1620         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1621         //cudaDeviceSynchronize();
1622         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1623         //cudaDeviceSynchronize();
1624     }
1625
1626     // residue calculation
1627     if (step < gridTo)
1628     {
1629         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 );
1630         //cudaDeviceSynchronize();
1631     }
1632 }
1633
1634
1635
1636
1637
1638 {
1639     int step = (gridTo - 1);
1640     iOne = iOne / 2;
1641     jOne = jOne / 2;
1642
1643     grid_RRow      = ((RRow - 1) / iOne) + 1;
1644     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1645
1646     grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
1647     coef_StartPos -= grid_RRow;

```

```

1648
1649     h = gridSizeR * iOne;
1650     h2 = h * h;
1651     ih2 = 1.0 / h2;
1652
1653     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1654     tempRatioPhi = ratioPhi * iOne * iOne;
1655
1656     // copy constant to device memory
1657     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
1658 );
1659     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
1660 );
1661     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1662     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1663     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1664
1665     // set kernel grid size and block size
1666     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
1667 grid_ZColumn / 16), PhiSlice);
1668     dim3 grid_ThreadPerBlock(16, 16);
1669
1670     // prolongation
1671     prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
1672 grid_ZColumn, grid_PhiSlice );
1673 // cudaDeviceSynchronize();
1674
1675     // red-black gauss seidel relaxation (nPost times)
1676     for (int i = 0; i < nPost; i++)
1677     {
1678         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
1679 d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1680 // cudaDeviceSynchronize();
1681         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
1682 d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1683 // cudaDeviceSynchronize();
1684     }
1685 }
1686
1687 // down one
1688 {
1689     residueCalculation<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
1690 d_RhoChargeDensity, d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_icoef4 );
1691
1692     iOne = iOne * 2;
1693     jOne = jOne * 2;
1694
1695     grid_StartPos += grid_RRow * grid_ZColumn * PhiSlice;
1696     coef_StartPos += grid_RRow;
1697
1698     grid_RRow = ((RRow - 1) / iOne) + 1;
1699     grid_ZColumn = ((ZColumn - 1) / jOne) + 1;
1700
1701     // pre-compute constant memory
1702     h = gridSizeR * iOne;
1703     h2 = h * h;
1704     ih2 = 1.0 / h2;
1705
1706     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1707     tempRatioPhi = ratioPhi * iOne * iOne;
1708
1709     // copy constant to device memory
1710     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
1711 );
1712     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
1713 );
1714     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1715     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1716     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1717
1718     // set kernel grid size and block size
1719     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
1720 grid_ZColumn / 16), PhiSlice);
1721     dim3 grid_ThreadPerBlock(16, 16);
1722
1723     // restriction
1724     restriction2DFull<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_RhoChargeDensity,
1725 d_DeltaResidue, grid_RRow, grid_ZColumn, grid_PhiSlice );
1726 //cudaDeviceSynchronize();
1727
1728     // zeroing V
1729     zeroingVPotential<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
1730 grid_ZColumn, grid_PhiSlice );
1731 //cudaDeviceSynchronize();
1732
1733     // red-black gauss seidel relaxation (nPre times)

```

```

1723         for (int i = 0; i < nPre; i++)
1724         {
1725             relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1726             //cudaDeviceSynchronize();
1727             relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1728             //cudaDeviceSynchronize();
1729         }
1730     }
1731 }
1732
1733 // V-Cycle => from coarser to finer grid
1734 for (int step = (gridTo - 1); step >= gridFrom; step--)
1735 {
1736     iOne = iOne / 2;
1737     jOne = jOne / 2;
1738
1739     grid_RRow      = ((RRow - 1) / iOne) + 1;
1740     grid_ZColumn   = ((ZColumn - 1) / jOne) + 1;
1741
1742     grid_StartPos -= grid_RRow * grid_ZColumn * PhiSlice;
1743     coef_StartPos -= grid_RRow;
1744
1745     h      = gridSizeR * iOne;
1746     h2     = h * h;
1747     ih2    = 1.0 / h2;
1748
1749     tempRatioZ = ratioZ * iOne * iOne / (jOne * jOne);
1750     tempRatioPhi = ratioPhi * iOne * iOne;
1751
1752     // copy constant to device memory
1753     cudaMemcpyToSymbol( d_grid_StartPos, &grid_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1754     cudaMemcpyToSymbol( d_coef_StartPos, &coef_StartPos, 1 * sizeof(int), 0, cudaMemcpyHostToDevice
);
1755     cudaMemcpyToSymbol( d_h2, &h2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1756     cudaMemcpyToSymbol( d_ih2, &ih2, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1757     cudaMemcpyToSymbol( d_tempRatioZ, &tempRatioZ, 1 * sizeof(float), 0, cudaMemcpyHostToDevice );
1758
1759     // set kernel grid size and block size
1760     dim3 grid_BlockPerGrid((grid_RRow < 16) ? 1 : (grid_RRow / 16), (grid_ZColumn < 16) ? 1 : (
grid_ZColumn / 16), PhiSlice);
1761     dim3 grid_ThreadPerBlock(16, 16);
1762
1763     // prolongation
1764     prolongation2DHalf<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential, grid_RRow,
grid_ZColumn, grid_PhiSlice );
1765     //
1766     cudaDeviceSynchronize();
1767
1768     // red-black gauss seidel relaxation (nPost times)
1769     for (int i = 0; i < nPost; i++)
1770     {
1771         relaxationGaussSeidelRed<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1772         //
1773         cudaDeviceSynchronize();
1774         relaxationGaussSeidelBlack<<< grid_BlockPerGrid, grid_ThreadPerBlock >>>( d_VPotential,
d_RhoChargeDensity, grid_RRow, grid_ZColumn, grid_PhiSlice, d_coef1, d_coef2, d_coef3, d_coef4 );
1775         //
1776         cudaDeviceSynchronize();
1777     }
1778 }
1779
1780 /*V-Cycle ends*/
1781
1782 errorCalculation<<< error_BlockPerGrid, error_ThreadPerBlock >>> ( d_VPotentialPrev, d_VPotential,
d_EpsilonError, RRow, ZColumn, PhiSlice);
1783
1784 cudaMemcpy( EpsilonError, d_EpsilonError, 1 * sizeof(float), cudaMemcpyDeviceToHost );
1785
1786 errorConv[cycle] = *EpsilonError / (RRow * ZColumn * PhiSlice);
1787
1788 if (errorConv[cycle] < convErr)
1789 {
1790     //errorConv
1791     nCycle = cycle;
1792     iparam[3] = nCycle;
1793     break;
1794 }
1795
1796 cudaMemcpy( d_VPotentialPrev, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
cudaMemcpyDeviceToDevice );
1797 cudaMemcpy( d_EpsilonError, 0, 1 * sizeof(float) );
1798
1799
1800
1801

```

```
1802     }
1803
1804     cudaDeviceSynchronize();
1805     // copy result from device to host
1806     cudaMemcpy( temp_VPotential, d_VPotential, RRow * ZColumn * PhiSlice * sizeof(float),
1807               cudaMemcpyDeviceToHost );
1808
1809     memcpy(VPotential, temp_VPotential, RRow * ZColumn * PhiSlice * sizeof(float));
1810
1811     // free device memory
1812     cudaFree( d_VPotential );
1813     cudaFree( d_VPotentialPrev );
1814     cudaFree( d_EpsilonError );
1815
1816     cudaFree( d_DeltaResidue );
1817     cudaFree( d_RhoChargeDensity );
1818     cudaFree( d_coef1 );
1819     cudaFree( d_coef2 );
1820     cudaFree( d_coef3 );
1821     cudaFree( d_coef4 );
1822     cudaFree( d_icoef4 );
1823
1824     // free host memory
1825     free( coef1 );
1826     free( coef2 );
1827     free( coef3 );
1828     free( coef4 );
1829     free( icoef4 );
1830     free( temp_VPotential );
1831     //free( temp_VPotentialPrev );
1832 }
```


Indeks

- [/home/nfs/aswardiana/Workspace/mp-headnode/lipi/↔
PoissonSolver3D/example/PoissonSolver3↔
DGPUtest.h, 21](#)
- [/home/nfs/aswardiana/Workspace/mp-headnode/lipi/↔
PoissonSolver3D/interface/PoissonSolver3↔
DCylindricalGPU.h, 24](#)
- [/home/nfs/aswardiana/Workspace/mp-headnode/lipi/↔
PoissonSolver3D/kernel/PoissonSolver3DG↔
PU.cu, 24](#)
- [/home/nfs/aswardiana/Workspace/mp-headnode/lipi/↔
PoissonSolver3D/kernel/PoissonSolver3DG↔
PU.h, 49](#)
- [CycleType
PoissonSolver3DCylindricalGPU, 15](#)
- [DoPoissonSolverExperiment
PoissonSolver3DGPUtest.h, 22](#)
- [fgkIFCRadius
PoissonSolver3DCylindricalGPU, 19](#)
- [GridTransferType
PoissonSolver3DCylindricalGPU, 16](#)
- [InterpType
PoissonSolver3DCylindricalGPU, 16](#)
- [PoissonMultigrid3DSemiCoarseningGPUError
PoissonSolver3DGPU.cu, 26
PoissonSolver3DGPU.h, 50](#)
- [PoissonMultigrid3DSemiCoarseningGPUErrorFCycle
PoissonSolver3DGPU.cu, 29
PoissonSolver3DGPU.h, 54](#)
- [PoissonMultigrid3DSemiCoarseningGPUErrorWCycle
PoissonSolver3DGPU.cu, 34
PoissonSolver3DGPU.h, 59](#)
- [PoissonSolver3DCylindricalGPU::MGParameters, 13](#)
- [PoissonSolver3DCylindricalGPU, 14
 - \[CycleType, 15\]\(#\)
 - \[fgkIFCRadius, 19\]\(#\)
 - \[GridTransferType, 16\]\(#\)
 - \[InterpType, 16\]\(#\)
 - \[PoissonSolver3DCylindricalGPU, 17\]\(#\)
 - \[PoissonSolver3D, 18\]\(#\)
 - \[RelaxType, 16\]\(#\)
 - \[SetExactSolution, 18\]\(#\)
 - \[StrategyType, 17\]\(#\)](#)
- [PoissonSolver3DGPU.cu
 - \[PoissonMultigrid3DSemiCoarseningGPUError, 26\]\(#\)
 - \[PoissonMultigrid3DSemiCoarseningGPUErrorF↔
Cycle, 29\]\(#\)
 - \[PoissonMultigrid3DSemiCoarseningGPUErrorW↔
Cycle, 34\]\(#\)
 - \[PrintMatrix, 43\]\(#\)
 - \[relaxationGaussSeidelBlack, 44\]\(#\)
 - \[relaxationGaussSeidelRed, 45\]\(#\)
 - \[residueCalculation, 46\]\(#\)
 - \[Restrict_Boundary, 47\]\(#\)
 - \[restriction2DFull, 48\]\(#\)](#)
- [PoissonSolver3DGPU.h
 - \[PoissonMultigrid3DSemiCoarseningGPUError, 50\]\(#\)
 - \[PoissonMultigrid3DSemiCoarseningGPUErrorF↔
Cycle, 54\]\(#\)
 - \[PoissonMultigrid3DSemiCoarseningGPUErrorW↔
Cycle, 59\]\(#\)](#)
- [PoissonSolver3DGPUtest.h
 - \[DoPoissonSolverExperiment, 22\]\(#\)](#)
- [PoissonSolver3D
 - \[PoissonSolver3DCylindricalGPU, 18\]\(#\)](#)
- [PrintMatrix
 - \[PoissonSolver3DGPU.cu, 43\]\(#\)](#)
- [RelaxType
 - \[PoissonSolver3DCylindricalGPU, 16\]\(#\)](#)
- [relaxationGaussSeidelBlack
 - \[PoissonSolver3DGPU.cu, 44\]\(#\)](#)
- [relaxationGaussSeidelRed
 - \[PoissonSolver3DGPU.cu, 45\]\(#\)](#)
- [residueCalculation
 - \[PoissonSolver3DGPU.cu, 46\]\(#\)](#)
- [Restrict_Boundary
 - \[PoissonSolver3DGPU.cu, 47\]\(#\)](#)
- [restriction2DFull
 - \[PoissonSolver3DGPU.cu, 48\]\(#\)](#)
- [SetExactSolution
 - \[PoissonSolver3DCylindricalGPU, 18\]\(#\)](#)
- [StrategyType
 - \[PoissonSolver3DCylindricalGPU, 17\]\(#\)](#)