```
/*
Start: 02/03/23
End  :

Sources:
- https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/
- https://github.com/marcinsokolowski/msfitslib/blob/master/apps/avg_images.cpp
- https://github.com/marcinsokolowski/msfitslib/blob/master/src/bg_fits.h

Links for cufftPlanMany()
- https://docs.nvidia.com/cuda/cufft/
- https://docs.nvidia.com/cuda/cufft/index.html#function-cufftplanmany
- https://stackoverflow.com/questions/26918101/1d-ffts-of-columns-and-rows-of-a-3d-matrix-in-cuda

Links for high_resolution_clock:
- https://cplusplus.com/reference/chrono/high_resolution_clock/now/

Steps for writing a cuda code
Step 1: Declare CPU and GPU variables
Step 2: Allocate memory for CPU and GPU variables
Step 3: Copy contents from CPU to GPU variables
Step 4: Call to GPU kernel
Step 5: Copy contents from GPU to CPU
Step 6: Free CPU and GPU memory

Compile on Topaz:
salloc --partition gpuq-dev --time 1:00:00 --nodes=1 --mem=10gb --gres=gpu:1
nvidia-smi
./rm!
./build.sh

Modules to be loaded:
module purge
module load cuda
module load cascadelake slurm/20.02.3 gcc/8.3.0 cmake/3.18.0
module use /group/director2183/software/centos7.6/modulefiles
module load ds9
module load msfitslib/devel
module load msfitslib/devel libnova
module load pal/0.9.8
module load libnova/0.15.0
module load cascadelake
module load gcc/8.3.0
module load cfitsio/3.48
module load cmake/3.18.0
// Additional modules to be loaded from ./build.sh: (24/02/23)
// In order to use my GPU Imager
module use /group/director2183/msok/software/centos7.6/modulefiles/
module use /group/courses0100/software/nvhpc/modulefiles
module load nvhpc/21.9
module load cuda/11.4.2
// module load cuda/11.1

Run on Topaz:
/group/director2183/data/test/ganiruddha/NEW_TEST/cuFFT_GITLAB_140323/imager_devel/build/cufft_blocks -u u.fi

Download .fits files:
scp ganiruddha@topaz.pawsey.org.au:/group/director2183/data/test/ganiruddha/NEW_TEST/cuFFT_GITLAB_140323/imag

Upload .fits files from CIRA Desktop:
scp *.fits ganiruddha@topaz.pawsey.org.au:/group/director2183/data/test/ganiruddha/NEW_TEST/cuFFT_GITLAB_NCHA

Location of test data: (BLINK)
- GitLab: http://146.118.67.64/blink/test-data/-/tree/main/eda2/20200209/images

Images for comparison:
- dirty_image_20221018T094254446_real.fits
- dirty_image_20221018T094254447_imag.fits

Inputs from user:
1) Correlation matrix:
- real visibilities: chan_204_20200209T034646_vis_real.fits
- imag visibilities: chan_204_20200209T034646_vis_imag.fits
2) u, v and w coordinates from antenna positions: u.fits, v.fits and w.fits
Operation:
- cuFFT + gridding on one BIG BLOCK for multiple frequency channels.
- Here gridding visibilities in every block is done SEQUENTIALLY.
Output: .fits file (N images generated from N blocks)
*/
```

```
// #include"filename": programmer defined
// #include<filename>: compiler defined
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

#include <math.h>
#include <string.h>
#include <time.h>
#include <vector>

// For cuda functions
#include <cuda_runtime.h>
#include <cuda.h>
#include <cufft.h>
#include <cufftw.h>

// So that it recognises: blockIdx
#include <device_launch_parameters.h>

// For handling .fits files
#include "bg_fits.h"
#include <bg_globals.h>

// In order to concatenate two strings
#include <bits/stdc++.h>
// #include <cstring>

// In order to use high resolution clock
#include <ctime>
#include <ratio>
#include <chrono>

// In order to use the gridding kernel
#include "../src/hip/gridding_imaging_cuda.h"
#include "../src/hip/pacer_imager_hip_defines.h"

// In order to use #pragma acc parallel loop directive
#include <openacc.h>

bool gWriteFits=false;
string u = "u.fits";
string v = "v.fits";
string w = "w.fits";
string vis_real = "vis_real.fits";
string vis_imag = "vis_imag.fits";
int N = 1; // default number of blocks =  1
int n_pixels = 180; // default image pixels = 180
int n_channels = 1; // default number of channels of the code = 1
int nStreams = 15; // number of CUDA streams (like queues for kernel executions)
bool gUseBlocks=false;
bool gDebuggerCalculateControlSum=false;

// constant UVW (like for EDA2) -> does not require recaculation of UVW grid for every single timestamp:
bool gConstantUVW=true; // for EDA2

// Observing parameters :
double frequency_MHz = 159.375; // default value in MHz

#define VEL_LIGHT  299792458.0
// #define NTHREADS 1024 // in pacer_imager_hip_defines.h

  // fft_shift(): Taken from Marcin's code
  void fft_shift( CBgFits& dirty_image, CBgFits& out_image )
  {
    int xSize = dirty_image.GetXSize();
    int ySize = dirty_image.GetYSize();

    CBgFits tmp_image( xSize, ySize );

    int center_freq_x = int( xSize/2 );
    int center_freq_y = int( ySize/2 );

    int is_odd = 0;
    if ( (xSize%2) == 1 && (ySize%2) == 1 )
    {
        is_odd = 1;
    }
```

```
    for(int y=0;y<ySize;y++)
    {
        float* tmp_data = tmp_image.get_line(y);
        float* image_data = dirty_image.get_line(y);

        for(int x=0;x<=center_freq_x;x++)
        {
          tmp_data[center_freq_x+x] = image_data[x];
        }
        for(int x=(center_freq_x+is_odd);x<xSize;x++)
        {
          tmp_data[x-(center_freq_x+is_odd)] = image_data[x];
        }
    }

    for(int x=0;x<xSize;x++)
    {
        for(int y=0;y<=center_freq_y;y++)
        {
          out_image.setXY(x,center_freq_y+y,tmp_image.getXY(x,y));
        }
        for(int y=(center_freq_y+is_odd);y<ySize;y++)
        {
          out_image.setXY( x , y-(center_freq_y+is_odd),tmp_image.getXY(x,y));
        }
    }
}

void usage()
{
    printf("cufft_blocks.cu OPTIONS\n");
    printf("-u u.fits : input FITS file with U values [default %s]\n",u.c_str());
    printf("-v v.fits : input FITS file with V values [default %s]\n",v.c_str());
    printf("-w w.fits : input FITS file with W values [default %s]\n",w.c_str());
    printf("-r VIS_REAL : inputs FITS file with REAL part of visibilities [default %s]\n",vis_real.c_str());
    printf("-i VIS_IMAG : inputs FITS file with IMAG part of visibilities [default %s]\n",vis_imag.c_str());
    printf("-F WRITE_OUTPUT_FITS_FILES : write output FITS files [default %d]\n",gWriteFits);
    printf("-n N_BLOCKS : number of blocks to test [default %d]\n",N);
    printf("-f N_CHANNELS : number of frequency channels (or iterations) [default %d]\n",n_channels);
    printf("-p SIZE : size of image (on side) -> full number of pixels is SIZE x SIZE [default SIZE = %d]\n",
    printf("-c : enable calculation of control sum on gridded visibilities to check if they are not zero [def
    printf("-s NUMBER_OF_STREAMS : number of CUDA streams used in gridding [default %d]\n",nStreams);
    printf("-M : changing UVW like for the MWA [default %d , i.e. constant like for all-sky images like for E
    printf("-B : use gridding kernels with blocks [default %d]. Maybe worth start testing with -s 1 (number c
    printf("-m : frequency in MHz [default %.6f MHz]\n",frequency_MHz);

    exit(0);
}



// Something to do with command-line arguments
void parse_cmdline(int argc, char * argv[])
{
  char optstring[] = "cn:p:f:r:i:u:v:w:F:s:MBm:";
  int opt;

  while ((opt = getopt(argc, argv, optstring)) != -1)
  {
     switch (opt)
     {
        case 'B':
           gUseBlocks = true;
           break;

        case 'M':
           gConstantUVW = false;
           break;

        case 'c':
           gDebuggerCalculateControlSum = true;
           break;

        case 'm':
           if( optarg )
           {
              frequency_MHz = atof( optarg );
           }
           break;
```

```
           break;

        case 'u':
           if( optarg )
           {
              u = optarg;
           }
           break;

        case 'v':
           if( optarg )
           {
              v = optarg;
           }
           break;

        case 'w':
           if( optarg )
           {
              w = optarg;
           }
           break;

        case 'n':
           if( optarg )
           {
              N = atoi(optarg);
           }
           break;

        case 'f':
           if( optarg )
           {
              n_channels = atoi(optarg);
           }
           break;

        case 'p':
           if( optarg )
           {
              n_pixels = atoi(optarg);
           }
           break;

        case 'r':
           if( optarg )
           {
              vis_real = optarg;
           }
           break;

        case 'i':
           if( optarg )
           {
              vis_imag = optarg;
           }
           break;

        case 'F':
           if( optarg )
           {
              gWriteFits = (atol(optarg)>0);
           }
           break;

        case 's':
           if( optarg )
           {
              nStreams = atol(optarg);
           }
           break;

        default:
           fprintf(stderr,"Unknown option %c\n",opt);
           exit(0);
     }
  }

/*  if( (n_pixels%2) != 0 ){
     printf("ERROR : only even image sizes are allowed in this version (to optimise kernel), change value of
```

```
     exit(-1);
  }*/
}

/*
argc: Number of command line arguments
argv: List of command line arguments
*/
int main(int argc, char* argv[])
{
    using namespace std::chrono;
    if( argc>=2 && strncmp(argv[1],"-h",2)==0 ){
       usage();
    }

    // CODE START
    printf("\n START CODE");
    clock_t start_time = clock();

    // Printing number of command line arguments
    cout << "\n You have entered " << argc << " arguments:" << "\n";

    // Printing the list of command line arguments
    for (int i = 0; i < argc; ++i)
        cout << i+1 << ":" << argv[i] << "\n";

    parse_cmdline(argc,argv);

    // Values specific to this program
    double frequency_Hz = frequency_MHz*1e6;
    double wavelength = VEL_LIGHT/frequency_Hz;
    printf("\n OK frequency_Hz: %.4f, wavelength: %.4f",frequency_Hz, wavelength);

    // Reading .fits files:
    printf("\n Reading in .fits files..");
    // Input .fits
    CBgFits u_fits;
    CBgFits v_fits;
    CBgFits w_fits;
    CBgFits vis_real_fits;
    CBgFits vis_imag_fits;

    u_fits.ReadFits( u.c_str() , 0, 1, 1 );
    v_fits.ReadFits( v.c_str() , 0, 1, 1 );
    w_fits.ReadFits( w.c_str() , 0, 1, 1 );
    vis_real_fits.ReadFits( vis_real.c_str() , 0, 1, 1 );
    vis_imag_fits.ReadFits( vis_imag.c_str() , 0, 1, 1 );

    // Input size: u, v and w
    int u_xSize = u_fits.GetXSize();
    int u_ySize = u_fits.GetYSize();
    int xySize = (u_xSize*u_ySize); // 256x256 for EDA2
    printf("\n OK xySize (u,v,w size) = %d", xySize);

    // Input size: vis_real
    int vis_real_xSize =  vis_real_fits.GetXSize();
    int vis_real_ySize =  vis_real_fits.GetYSize();
    int vis_real_size = (vis_real_xSize*vis_real_ySize); // 256x256 for EDA2
    printf("\n OK vis_real_size = %d", vis_real_size);

    // Input size: vis_imag
    int vis_imag_xSize =  vis_imag_fits.GetXSize();
    int vis_imag_ySize =  vis_imag_fits.GetYSize();
    int vis_imag_size = (vis_real_xSize*vis_real_ySize); // 256X156 for EDA2
    printf("\n OK vis_imag_size = %d", vis_imag_size);

    // Image dimensions
    int width = n_pixels;
    int height = n_pixels;
    int image_size = (width*height);
    int block_size =  (N*image_size);
    printf("\n OK PARAMETERS : ");
    printf("\n OK Observing frequency = %.6f [MHz]\n",frequency_MHz);
    printf("\n OK Number of blocks: %d", N);
    printf("\n OK Number of channels: %d", n_channels);
    printf("\n OK n_pixels: %d", n_pixels);
    printf("\n OK Image Width: %d", width);
    printf("\n OK Image Height: %d", height);
    printf("\n OK Overall Image Size: %d", image_size);
    printf("\n OK Overall Block size: %d", block_size);
```

```
    printf("\n OK Overall Block size: %d", block_size);
    printf("\n OK Number of CUDA streams: %d", nStreams);
    printf("\n OK Constant UVW      : %d", gConstantUVW);
    printf("\n OK Use CUDA BLOCKS   : %d", gUseBlocks);

    // Step 1: Declare CPU/GPU Input/Output Variables

    // GPU input/output variables
    cufftComplex *m_in_buffer_gpu=NULL;
    cufftComplex *m_out_buffer_gpu=NULL;

    float *u_gpu=NULL;
    float *v_gpu=NULL;
    float *vis_real_gpu=NULL;
    float *vis_imag_gpu=NULL;

    float *uv_grid_real_gpu=NULL;
    float *uv_grid_imag_gpu=NULL;
    float *uv_grid_counter_gpu=NULL;
    float *uv_grid_counter_single_gpu=NULL; // for a single (constant UVW)

    // CPU input/output variables
    cufftComplex *m_in_buffer_cpu=NULL;
    cufftComplex *m_out_buffer_cpu=NULL;

    float *u_cpu = u_fits.get_data();
    float *v_cpu = v_fits.get_data();
    float *vis_real_cpu = vis_real_fits.get_data();
    float *vis_imag_cpu = vis_imag_fits.get_data();

    CBgFits uv_grid_real_fits(width, height);
    CBgFits uv_grid_imag_fits(width, height);
    CBgFits uv_grid_counter_fits(width, height);
    uv_grid_real_fits.SetValue( 0.00 );
    uv_grid_imag_fits.SetValue( 0.00 );
    uv_grid_counter_fits.SetValue( 0.00 );

    float *uv_grid_real_cpu = uv_grid_real_fits.get_data();
    float *uv_grid_imag_cpu = uv_grid_imag_fits.get_data();
    float *uv_grid_counter_cpu = uv_grid_counter_fits.get_data();
    printf("\n OK: Input/Output buffers declared");

    // Step 2: Allocate memory for Input/Output GPU variables
    CUDA_CHECK_ERROR(cudaMalloc((void**) &m_in_buffer_gpu, sizeof(cufftComplex)*image_size*N));
    CUDA_CHECK_ERROR(cudaMalloc((void**) &m_out_buffer_gpu, sizeof(cufftComplex)*image_size*N));

    CUDA_CHECK_ERROR(cudaMalloc((float**)&vis_real_gpu, xySize*sizeof(float)));
    CUDA_CHECK_ERROR(cudaMalloc((float**)&vis_imag_gpu, xySize*sizeof(float)));
    CUDA_CHECK_ERROR(cudaMalloc((float**)&u_gpu, xySize*sizeof(float)));
    CUDA_CHECK_ERROR(cudaMalloc((float**)&v_gpu, xySize*sizeof(float)));

    CUDA_CHECK_ERROR(cudaMalloc((float**)&uv_grid_real_gpu, image_size*sizeof(float)));
    CUDA_CHECK_ERROR(cudaMalloc((float**)&uv_grid_imag_gpu, image_size*sizeof(float)));

    if( gConstantUVW ){
       // for constant UVW - counter can be calculated once and for all !
       CUDA_CHECK_ERROR(cudaMalloc((float**)&uv_grid_counter_single_gpu, image_size*sizeof(float)));
       CUDA_CHECK_ERROR(cudaMemset((float*)uv_grid_counter_single_gpu, 0, sizeof(float)*image_size) );

       // WARNING : temporarily until the new kernel is created which will not be calcuating uv_counter this
       // CUDA_CHECK_ERROR(cudaMalloc((float**)&uv_grid_counter_gpu, image_size*sizeof(float)*N));
    }else{
       // for non-constant UVW - counter should be calculated for every timestep
       CUDA_CHECK_ERROR(cudaMalloc((float**)&uv_grid_counter_gpu, image_size*sizeof(float)*N));
       CUDA_CHECK_ERROR(cudaMemset((float*)uv_grid_counter_gpu, 0, sizeof(float)*image_size*N) );
    }

    if( gDebuggerCalculateControlSum ){
       m_in_buffer_cpu = (cufftComplex*)malloc(sizeof(cufftComplex)*image_size*N);
       if( !m_in_buffer_cpu )
       {
          printf("ERROR : while allocating Host (Input) memory size  ...\n");
          exit(-1);
       }
    }

    m_out_buffer_cpu = (cufftComplex*)malloc(sizeof(cufftComplex)*image_size*N);
    if( !m_out_buffer_cpu )
    {
       printf("ERROR : while allocating Host (Output) memory size  ...\n");
```

```
      exit(-1);
    }
    printf("\n OK: Input/Output buffers memory allocated");

    // Step 3: Copy contents from CPU to GPU variables
    CUDA_CHECK_ERROR(cudaMemcpy((float*)u_gpu, (float*)u_cpu, sizeof(float)*xySize, cudaMemcpyHostToDevice));
    CUDA_CHECK_ERROR(cudaMemcpy((float*)v_gpu, (float*)v_cpu, sizeof(float)*xySize, cudaMemcpyHostToDevice));

    // CONSTANTS once :
    // delta_u, delta_v calculations
    double FOV_degrees = 180.00;
    double FoV_radians = FOV_degrees*M_PI/180;
    double delta_u = 1.00/(FoV_radians);
    double delta_v = 1.00/(FoV_radians);
    printf("\n OK M_PI: %f",M_PI);
    printf("\n OK FOV_degrees: %f", FOV_degrees);
    printf("\n OK FOV_radians: %f", FoV_radians);
    printf("\n OK delta_u, delta_v (C++) : %f %f", delta_u, delta_v);
    int center_x = int(n_pixels/2);
    int center_y = int(n_pixels/2);
    double min_uv = -1000;

    // Setting the initial values of is_odd_x, is_odd_y = 0
    int is_odd_x = 0;
    int is_odd_y = 0;
    // Calculating new values of is_odd_x, is_odd_y, depending on image dimensions
    if( (n_pixels % 2) == 1 )
    {
      is_odd_x = 1;
      is_odd_y = 1;
    }

    if( gConstantUVW ){
       // calculate counter once :
       int nBlocks = (xySize + NTHREADS -1)/NTHREADS;
       calculate_counter<<<nBlocks,NTHREADS>>>(xySize, u_gpu, v_gpu, wavelength, image_size, delta_u, delta_v

       CUDA_CHECK_ERROR(cudaMemcpy((float*)uv_grid_counter_cpu, (float*)uv_grid_counter_single_gpu, sizeof(fl
       // Need to reset CUDA memory after this gridding to only calculate counter :
       // ptr_b_gpu, but uv_grid_real_gpu and uv_grid_imag_gpu are not used so they can be ignored and in the
//        CUDA_CHECK_ERROR( cudaMemset(ptr_b_gpu, 0, sizeof(cufftComplex)*image_size) );
    }

    // this emulates copying data from somewhere else -> so can stay inside this loop
    CUDA_CHECK_ERROR(cudaMemcpy((float*)vis_real_gpu, (float*)vis_real_cpu, sizeof(float)*xySize, cudaMemcpyH
    CUDA_CHECK_ERROR(cudaMemcpy((float*)vis_imag_gpu, (float*)vis_imag_cpu, sizeof(float)*xySize, cudaMemcpyH

    int n[2];
    n[0] = width;
    n[1] = height;

    // START: cufftPLanMany()
    high_resolution_clock::time_point t1 = high_resolution_clock::now();

    cufftHandle plan;
    cufftPlanMany(&plan, 2, n, NULL, 1, image_size, NULL, 1, image_size, CUFFT_C2C, N);
    high_resolution_clock::time_point t1a = high_resolution_clock::now();
    duration<double> time_span1 = duration_cast<duration<double>>(t1a - t1);
    printf("\n CLOCK cufftPlanMany() took: %.6f seconds. PARAMETERS ( N_PIXELS , N_BLOCKS , N_STREAMS , N_CHA
    // END: cufftPlanMany()

    // Iterating over number of frequency channels
    for(int c=0; c< n_channels; c++)
    {
      // re-initialise memory in UV grid before every iteration (other wise the previous gridding is there !)
      CUDA_CHECK_ERROR( cudaMemset(m_in_buffer_gpu, 0, sizeof(cufftComplex)*image_size*N) );

      int nBlocks = (xySize + NTHREADS -1)/NTHREADS;
      printf("\n CHECK: NTHREADS = %d", NTHREADS);
      printf("\n CHECK: nBlocks = %d", nBlocks);

      // START: visibilities into N blocks
      high_resolution_clock::time_point t1A = high_resolution_clock::now();

      // Iterating through every block
      // For future optimisations:
      cudaStream_t *streams = new cudaStream_t[nStreams];
      for(int i = 0; i < nStreams; i++)
         cudaStreamCreate(&streams[i]);
```

```
    high_resolution_clock::time_point t1B = high_resolution_clock::now();
    duration<double> time_span_stream_create = duration_cast<duration<double>>(t1B - t1A);
    printf("\n CLOCK cudaStreamCreate() took: %.6f seconds. PARAMETERS ( N_PIXELS , N_BLOCKS , N_STREAMS ,


    if( gUseBlocks ){
       int nBlocksPerThread = (xySize + NTHREADS -1)/NTHREADS;
       dim3 blocks( nBlocksPerThread, N, 1 );

       printf("DEBUG : using blocks (gridsize and no loop)\n");
       if( gConstantUVW ){
           printf("DEBUG : blocks (no loop) and constant UVW\n");
           printf("DEBUG : using gridding kernel with gridsize %d x %d x 1 and %d streams (constant UVW)\n"

           gridding_imaging_cuda_blocks_optimised_nocounter<<<blocks,NTHREADS>>>(xySize, u_gpu, v_gpu, wave
       }else{
           printf("DEBUG : blocks (no loop) and non-constant UVW\n");
           printf("DEBUG : using gridding kernel with gridsize %d x %d x 1 and %d streams (constant UVW)\n"

           gridding_imaging_cuda_blocks_optimised<<<blocks,NTHREADS>>>(xySize, u_gpu, v_gpu, wavelength, im
       }
    }else{
       printf("DEBUG : not using blocks (loop)\n");
       for(int b=0; b<N; b++)
       {
          printf("\n OK PRAGMA CHECK ORDER: BLOCK: %d",b);
          // gpu_subarray =  gpu_data + start_index;
          cufftComplex* ptr_b_gpu = m_in_buffer_gpu + (b*image_size);
          float* ptr_counter_gpu = uv_grid_counter_gpu + (b*image_size);

          if( gConstantUVW ){
             printf("DEBUG :  executing kernel which does not require counter re-calculation for every BLOC
             printf("DEBUG : executing kernel without blocks (%d x %d) and %d streams (constant UVW)\n",nBl
             // Optimised and not calculating counter (done earlier using call to calculate_counter)
             gridding_imaging_cuda_optimised_nocounter<<<nBlocks,NTHREADS, 0, streams[b % nStreams]>>>(xySi
          }else{
             printf("DEBUG : executing kernel which requires recalculation of counter for every BLOCK (time
             printf("DEBUG : executing kernel without blocks (%d x %d) and %d streams (constant UVW)\n",nBl
             gridding_imaging_cuda_optimised<<<nBlocks,NTHREADS, 0, streams[b % nStreams]>>>(xySize, u_gpu,
          }
          CUDA_CHECK_ERROR(cudaGetLastError());
       }
    }
    CUDA_CHECK_ERROR(cudaDeviceSynchronize());

    // Destroy and FREE
    for(int i = 0; i < nStreams; i++)
      cudaStreamDestroy( streams[i] );

    delete [] streams;

    // END: gridding visibilities into N blocks
    high_resolution_clock::time_point t2B = high_resolution_clock::now();
    duration<double> time_span_gridding = duration_cast<duration<double>>(t2B - t1B);
    printf("\n CLOCK gridding() cudaStream took: %.6f seconds. PARAMETERS ( N_PIXELS , N_BLOCKS , N_STREAMS
    printf("\n OK: Visibilities gridded into m_in_buffer_gpu");

    if( m_in_buffer_cpu )
    {
       // only if checking debug-gridded visibilities (calculation of control sum is required)
       CUDA_CHECK_ERROR(cudaMemcpy((cufftComplex*)m_in_buffer_cpu, (cufftComplex*)m_in_buffer_gpu, sizeof(c
       printf("\n OK:(m_in_buffer_gpu to m_in_buffer_cpu) Data copied from GPU to CPU for CHECKS");

       // Checking values of m_in_buffer_cpu
       float sum_real_input[N] = {0};
       float sum_imag_input[N] = {0};

       cufftComplex* ptr_b_temp_input;
       for(int b=0; b<N; b++)
       {
         ptr_b_temp_input = m_in_buffer_cpu + (b*image_size);
         for(int i=0;i<image_size;i++)
         {
           sum_real_input[b] += ptr_b_temp_input[i].x;
           sum_imag_input[b] += ptr_b_temp_input[i].y;
         }
       }
       // Sum of all real, imag values of every block
       for(int b=0; b<N; b++)
```

```
    {
      printf("\n sum_real_input[%d] = %f,  sum_imag_input[%d] = %f ", b, sum_real_input[b],b, sum_imag_i
    }
  }
  else
  {
     printf("\nDEBUG : calculation of control sum on gridded visibilities is not required -> no need to c
  }

  // measure separately :
  t1 = high_resolution_clock::now();
  cufftExecC2C(plan, m_in_buffer_gpu, m_out_buffer_gpu, CUFFT_FORWARD);
  cudaDeviceSynchronize();
  printf("\n OK cufftPlanMany() executed!");

  high_resolution_clock::time_point t2 = high_resolution_clock::now();
  duration<double> time_span = duration_cast<duration<double>>(t2 - t1);
  // std::cout << "\n CLOCK cufftExecC2C() took: " << time_span.count() << " seconds. \n";
  printf("\n CLOCK cufftExecC2C() took: %.6f seconds. PARAMETERS ( N_PIXELS , N_BLOCKS , N_STREAMS , N_CH

  // Step 5: Copy contents from GPU to CPU
  CUDA_CHECK_ERROR(cudaMemcpy((cufftComplex*)m_out_buffer_cpu, (cufftComplex*)m_out_buffer_gpu, sizeof(cu
  printf("\n OK: Data copied from GPU to CPU");

  // Checking values of m_out_buffer_cpu
  float sum_real_output[N] = {0};
  float sum_imag_output[N] = {0};

  cufftComplex* ptr_b_temp_output;
  for(int b=0; b<N; b++)
  {
    ptr_b_temp_output = m_out_buffer_cpu + (b*image_size);
    for(int i=0;i<image_size;i++)
    {
      sum_real_output[b] += ptr_b_temp_output[i].x;
      sum_imag_output[b] += ptr_b_temp_output[i].y;
    }
  }
  // Sum of all real/imag values of every block
  for(int b=0; b<N; b++)
  {
    printf("\n sum_real_output[%d] = %f, sum_imag_output[%d] = %f", b, sum_real_output[b], b, sum_imag_ou
  }

  // CUDA_CHECK_ERROR(cudaMemcpy((float*)uv_grid_counter_cpu, (float*)uv_grid_counter_gpu, sizeof(float)*
  // double fnorm = 1.00/uv_grid_counter_fits.Sum();
  double fnorm = 1.00/uv_grid_counter_fits.Sum();
  double fnorm_hardcoded = 0.000015;
  printf("\n fnorm = %f", fnorm);
  printf("\n fnorm_hardcoded = %f", fnorm_hardcoded);

  // For storing the final outputs
  CBgFits out_image_real(width, height);
  CBgFits out_image_imag(width, height);
  CBgFits out_image_real_shifted(width, height);
  CBgFits out_image_imag_shifted(width, height);

  char filename_real[1024];
  char filename_imag[1024];

  cufftComplex* ptr_output;

  for(int b=0;b<N;b++)
  {
    float* out_data_real = out_image_real.get_data();
    float* out_data_imag = out_image_imag.get_data();

    ptr_output = m_out_buffer_cpu + (b*image_size);

    for(int i=0;i<image_size;i++)
    {
      out_data_real[i] =  ptr_output[i].x*fnorm;
      out_data_imag[i] =  ptr_output[i].y*fnorm;
    }

    fft_shift(out_image_real,out_image_real_shifted);
    fft_shift(out_image_imag,out_image_imag_shifted);

    if( gWriteFits ){
      sprintf(filename_real,"re_%d%d.fits",c,b);
```

```
            printf("\n filename_real saved: %s", filename_real);
            out_image_real_shifted.WriteFits(filename_real);

            sprintf(filename_imag,"im_%d%d.fits",c,b);
            printf("\n filename_imag saved: %s", filename_imag);
            out_image_imag_shifted.WriteFits(filename_imag);
         }
       }

    }

    // Step 6: Free GPU memory
    cudaFree(m_in_buffer_gpu);
    cudaFree(m_out_buffer_gpu);
    cudaFree(u_gpu);
    cudaFree(v_gpu);
    cudaFree(vis_real_gpu);
    cudaFree(vis_imag_gpu);
    cudaFree(uv_grid_real_gpu);
    cudaFree(uv_grid_imag_gpu);
    if( uv_grid_counter_gpu ){
       cudaFree(uv_grid_counter_gpu);
    }
    if( uv_grid_counter_single_gpu ){
       cudaFree(uv_grid_counter_single_gpu);
    }

    if( m_in_buffer_cpu ){
       free(m_in_buffer_cpu);
    }
    free(m_out_buffer_cpu);

    printf("\n OK: CPU/GPU variables free");

    printf("\n END CODE");
    printf("\n");
    // End of main() function
    return 0;
}
```