```
/*
    Make magnitudes image
*/

#include "image_utilities.hcu"
#include "int_util.hcu"
#include "uint_util.hcu"

#include <stdio.h>
#include <cublas.h>
#include <cutil.h>


/*
    Image modulus
*/

__global__ void
cuda_image_modulus_kernel( cuComplex *devPtr_in, float *devPtr_out, unsigned int
    number_of_elements )
{
  int idx = blockIdx.x*blockDim.x+threadIdx.x;

  if( idx<number_of_elements ){
      cuComplex val = devPtr_in[idx];
      devPtr_out[idx] = sqrtf(val.x*val.x+val.y*val.y);
  }
}

__host__ void
cuda_image_modulus( cuComplex *devPtr_in, float *devPtr_out, unsigned int number_of_elements
    , bool normalize )
{
    dim3 blockDim(512,1,1);
    dim3 gridDim((unsigned int) ceil((double)(number_of_elements/512)), 1, 1 );

    // Make modulus image
    cuda_image_modulus_kernel<<< gridDim, blockDim >>>( devPtr_in, devPtr_out,
    number_of_elements );

    if( normalize ){

        /*
            Scale using cublas.
            Remember cublasIsamax uses 1-based indexing!!!
        */

        //Find the maximum value in the array
        int max_idx = cublasIsamax (number_of_elements, devPtr_out, 1);

        //Copy that value back to host memory
        float max_val;
        CUDA_SAFE_CALL(cudaMemcpy(&max_val, (devPtr_out+max_idx-1), sizeof(float),
    cudaMemcpyDeviceToHost));

        // printf("\nMax index/val: %d/%f\n", max_idx, max_val );

        //Scale the array
        cublasSscal( number_of_elements, 1.0f/max_val, devPtr_out, 1 );
    }
}


/*
    Clear image
*/

template< class T > __global__ void
cuda_clear_image_kernel( unsigned int num_elements, T val, T *imageDevPtr )
{
    const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if( idx < num_elements ){
```

```
        imageDevPtr[idx] = val;
    }
}

template< class T > __host__ void
cuda_clear_image( unsigned int num_elements, T val, T *imageDevPtr )
{

    // Find dimensions of grid/blocks.

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties( &deviceProp, 0 );

    dim3 dimBlock( deviceProp.maxThreadsPerBlock, 1, 1 );
    dim3 dimGrid( (unsigned int) ceil((double)num_elements/deviceProp.maxThreadsPerBlock), 1
    , 1 );

    // Invoke kernel
    cuda_clear_image_kernel<<< dimGrid, dimBlock >>>( num_elements, val, imageDevPtr );
}


/*
    Border fill image (square)
*/

template < class T, class UINTd> __global__ void
cuda_border_fill_image_kernel( UINTd matrix_size, UINTd matrix_size_os, T value, T *
    imageDevPtr, unsigned int number_of_images )
{
    const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    const unsigned int num_elements = prod(matrix_size_os);

    if( idx < num_elements ){
        const UINTd co = idx_to_co( idx, matrix_size_os );
        const UINTd corner1 = (matrix_size_os-matrix_size)>>1;
        const UINTd corner2 = matrix_size_os-corner1;
        if( weak_less(co,corner1) || weak_greater_equal(co,corner2) )
            for( unsigned int image=0; image<number_of_images; image++ )
                imageDevPtr[image*num_elements+idx] = value;
    }
}

template< class T, class UINTd > __host__ void
cuda_border_fill_image( UINTd matrix_size, UINTd matrix_size_os, T value, T *imageDevPtr,
    unsigned int number_of_images )
{
    // Find dimensions of grid/blocks.

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties( &deviceProp, 0 );

    dim3 dimBlock( deviceProp.maxThreadsPerBlock, 1, 1 );
    dim3 dimGrid( (unsigned int) ceil((double)prod(matrix_size_os)/deviceProp.
    maxThreadsPerBlock), 1, 1 );

    // Invoke kernel
    cudaThreadSynchronize();
    cuda_border_fill_image_kernel<<< dimGrid, dimBlock >>> ( matrix_size, matrix_size_os,
    value, imageDevPtr, number_of_images );
    cudaThreadSynchronize();
}


/*
    Border fill image (circular)
*/

template < class T, class UINTd > __global__ void
cuda_border_fill_image_kernel( UINTd matrix_size_os, unsigned int radius, T value, T *
    imageDevPtr, unsigned int number_of_images )
{
    const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
```

```
    const unsigned int num_elements = prod(matrix_size_os);

    if( idx < num_elements ){
        const UINTd co = idx_to_co( idx, matrix_size_os );
        const unsigned int sq_dist = dot((uint_to_int(co)-uint_to_int((matrix_size_os>>1))),
     (uint_to_int(co)-uint_to_int((matrix_size_os>>1))));
        if( sq_dist>radius*radius )
            for( unsigned int image=0; image<number_of_images; image++ )
                imageDevPtr[image*num_elements+idx] = value;
    }
}

template< class T, class UINTd > __host__ void
cuda_border_fill_image( UINTd matrix_size_os, unsigned int radius, T value, T *imageDevPtr,
    unsigned int number_of_images )
{

    // Find dimensions of grid/blocks.

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties( &deviceProp, 0 );

    dim3 dimBlock( deviceProp.maxThreadsPerBlock, 1, 1 );
    dim3 dimGrid( (unsigned int) ceil((double)prod(matrix_size_os)/deviceProp.
    maxThreadsPerBlock), 1, 1 );

    // Invoke kernel
    cudaThreadSynchronize();
    cuda_border_fill_image_kernel<<< dimGrid, dimBlock >>> ( matrix_size_os, radius, value,
    imageDevPtr, number_of_images );
    cudaThreadSynchronize();
}


/*
    Image crop
*/

template< class T, class UINTd > __global__ void
cuda_crop_image_kernel( UINTd out_matrix_size, UINTd in_matrix_size, T *out_imageDevPtr, T *
    in_imageDevPtr, unsigned int number_of_images )
{
    const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    const unsigned int num_elements = prod(out_matrix_size);

    if( idx < num_elements ){
        const UINTd co = idx_to_co( idx, out_matrix_size );
        const UINTd co_os = co + ((in_matrix_size-out_matrix_size)>>1);
        const unsigned int source_idx = co_to_idx(co_os, in_matrix_size);
        const unsigned int source_elements = prod(in_matrix_size);
        for( unsigned int image=0; image<number_of_images; image++ )
            out_imageDevPtr[image*num_elements+idx] = in_imageDevPtr[image*source_elements+
    source_idx];
    }
}

template< class T, class UINTd >
__host__ void cuda_crop_image( UINTd out_matrix_size, UINTd in_matrix_size, T *
    out_imageDevPtr, T *in_imageDevPtr, unsigned int number_of_images )
{
    // Find dimensions of grid/blocks.

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties( &deviceProp, 0 );

    dim3 dimBlock( deviceProp.maxThreadsPerBlock, 1, 1 );
    dim3 dimGrid( (unsigned int) ceil((double)prod(out_matrix_size)/(double)deviceProp.
    maxThreadsPerBlock), 1, 1 );

    // Invoke kernel
    cudaThreadSynchronize();
    cuda_crop_image_kernel<<< dimGrid, dimBlock >>> ( out_matrix_size, in_matrix_size,
    out_imageDevPtr, in_imageDevPtr, number_of_images );
```

```
    cudaThreadSynchronize();
}


/*
    Add two images
*/

template< class T > __global__ void
cuda_add_images_kernel( unsigned int num_elements, T *target, T *source1, T *source2  )
{
    const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if( idx < num_elements ){
        T out;
        switch(sizeof(T)){
            case sizeof(float):
                *((float*) &out) = ( *((float*) &source1[idx]) + *((float*) &source2[idx]));
                break;
            case sizeof(cuComplex):
                *((cuComplex*) &out) = cuCadd( *((cuComplex*) &source1[idx]), *((cuComplex*)
 &source2[idx]) );
                break;
        }
        target[idx] = out;
    }
}

template< class T > __host__ void
cuda_add_images( unsigned int num_elements, T *targetDevPtr, T *source1DevPtr,  T *
    source2DevPtr  )
{

    // Find dimensions of grid/blocks.

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties( &deviceProp, 0 );

    dim3 dimBlock( deviceProp.maxThreadsPerBlock, 1, 1 );
    dim3 dimGrid( (unsigned int) ceil((double)num_elements/deviceProp.maxThreadsPerBlock), 1
    , 1 );

    // Invoke kernel
    cuda_add_images_kernel<<< dimGrid, dimBlock >>>( num_elements, targetDevPtr,
    source1DevPtr, source2DevPtr );
}


/*
    Square image
*/

template< class T > __global__ void
cuda_square_image_kernel( unsigned int num_elements, T *imageDevPtr )
{
    const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

    if( idx < num_elements ){
        T source = imageDevPtr[idx];
        T out;

        switch(sizeof(T)){
            case sizeof(float):
                *((float*) &out) = ( *((float*) &source) * *((float*) &source));
                break;
            case sizeof(cuComplex):
                *((cuComplex*) &out) = cuCmul( *((cuComplex*) &source), *((cuComplex*) &
    source) );
                break;
        }
        imageDevPtr[idx] = out;
    }
}
```

```
template< class T > __host__ void
cuda_square_image( unsigned int num_elements, T *imageDevPtr )
{
    // Find dimensions of grid/blocks.

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties( &deviceProp, 0 );

    dim3 dimBlock( deviceProp.maxThreadsPerBlock, 1, 1 );
    dim3 dimGrid( (unsigned int) ceil((double)num_elements/(double)deviceProp.
    maxThreadsPerBlock), 1, 1 );

    // Invoke kernel
    cudaThreadSynchronize();
    cuda_square_image_kernel<<< dimGrid, dimBlock >>> ( num_elements, imageDevPtr );
    cudaThreadSynchronize();
}

/*
    Square image
*/

__global__ void
cuda_squareroot_modulus_image_kernel( unsigned int num_elements, float *imageDevPtr )
{
    const unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

    if( idx < num_elements )
        imageDevPtr[idx] = sqrtf(imageDevPtr[idx]);
}

__host__ void
cuda_squareroot_modulus_image( unsigned int num_elements, float *imageDevPtr )
{
    // Find dimensions of grid/blocks.

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties( &deviceProp, 0 );

    dim3 dimBlock( deviceProp.maxThreadsPerBlock, 1, 1 );
    dim3 dimGrid( (unsigned int) ceil((double)num_elements/(double)deviceProp.
    maxThreadsPerBlock), 1, 1 );

    // Invoke kernel
    cudaThreadSynchronize();
    cuda_squareroot_modulus_image_kernel<<< dimGrid, dimBlock >>> ( num_elements,
    imageDevPtr );
    cudaThreadSynchronize();
}


// Instanciation

template void cuda_border_fill_image( ::uint2, ::uint2, cuComplex, cuComplex*, unsigned int
    );
template void cuda_border_fill_image( ::uint3, ::uint3, cuComplex, cuComplex*, unsigned int
    );
template void cuda_border_fill_image( ::uint4, ::uint4, cuComplex, cuComplex*, unsigned int
    );
template void cuda_border_fill_image( ::uint2, ::uint2, float, float*, unsigned int );
template void cuda_border_fill_image( ::uint3, ::uint3, float, float*, unsigned int );
template void cuda_border_fill_image( ::uint4, ::uint4, float, float*, unsigned int );

template void cuda_border_fill_image( ::uint2, unsigned int, cuComplex, cuComplex*, unsigned
    int );
template void cuda_border_fill_image( ::uint3, unsigned int, cuComplex, cuComplex*, unsigned
    int );
template void cuda_border_fill_image( ::uint4, unsigned int, cuComplex, cuComplex*, unsigned
    int );
template void cuda_border_fill_image( ::uint2, unsigned int, float, float*, unsigned int );
template void cuda_border_fill_image( ::uint3, unsigned int, float, float*, unsigned int );
template void cuda_border_fill_image( ::uint4, unsigned int, float, float*, unsigned int );
```

```
template void cuda_crop_image( ::uint2, ::uint2, cuComplex*, cuComplex*, unsigned int );
template void cuda_crop_image( ::uint3, ::uint3, cuComplex*, cuComplex*, unsigned int );
template void cuda_crop_image( ::uint4, ::uint4, cuComplex*, cuComplex*, unsigned int );
template void cuda_crop_image( ::uint2, ::uint2, float*, float*, unsigned int );
template void cuda_crop_image( ::uint3, ::uint3, float*, float*, unsigned int );
template void cuda_crop_image( ::uint4, ::uint4, float*, float*, unsigned int );

template void cuda_clear_image( unsigned int, cuComplex val, cuComplex* );
template void cuda_clear_image( unsigned int, float val, float* );

template void cuda_square_image( unsigned int, cuComplex* );
template void cuda_square_image( unsigned int, float* );

template void cuda_add_images( unsigned int, cuComplex*, cuComplex*, cuComplex* );
template void cuda_add_images( unsigned int, float*, float*, float* );
```