Lab 4: Introduction to GPU

Converting RGB to Grayscale image

CPU

First, we'll implement this conversion on the CPU as a baseline. This is also a good method to make sure your algorithm is producing the output you want.

1. In `src/img_proc.cu`, complete the function `img_rgb2gray_cpu()` to convert an N-channel (3 in this case) image to a 1 channel image by averaging the N channels
2. In `src/main.cpp` call the `img_rgb2gray_cpu()` function.
    2. Add `img_rgb2gray_cpu(gray.ptr<uchar>(), rgb.ptr<uchar>(), WIDTH, HEIGHT, CHANNELS);` under the CPU case of the `while()` loop.
3. Compile and run using `./lab4 1`. Compare with OpenCV (using `./lab4 0`).

GPU using separate memory

Now, let's move to the GPU.

Changes to `img_proc.cu`

The changes/additions here are the actual algorithm implementation on the GPU and a wrapper function to be called on the host.

1. Add a new kernel function under `// =================== GPU Kernel Functions ===================`. This function needs to run on the device and be called from the host. The GPU function will have the same parameters as the CPU function.
    1. Your CPU function should have two nested `for()` loops to loop over the entire image. In the GPU kernel function, these loops should be replaced with the block and thread indexes.
    2. For the most part, the rest of the function will remain untouched.
2. Add a wrapper function to call your kernel function under `// =================== GPU Host Functions ===================`. This function will be called from `main()` and needs to define the `grid` and `block` and launch the kernel function.

Changes to `main.cpp`

The changes here will manage the GPU device memory and calling the wrapper function we wrote above. **We're NOT using unified memory**

1. Make sure `#define UNIFIED_MEMORY` is commented out at the top of `main.cpp`
2. Allocate memory on the device.
    1. Declare `unsigned char* gray_device`.
    2. Allocate the `gray_device` memory for the GPU device `cudaMalloc((void **)&gray_device, <SIZE_TO_ALLOCATE>)`
    3. Declare `unsigned char* rgb_device`.
    4. Allocate the `rgb_device` memory for the GPU device `cudaMalloc((void **)&rgb_device, <SIZE_TO_ALLOCATE>)`
3. Under the GPU case, copy the data from host -> device, call the GPU wrapper function, copy data from device -> host.
    1. Copy from host -> device `cudaMemcpy(<PTR_TO_DEVICE_MEM>, <PTR_TO_HOST_MEM>, <SIZE_TO_COPY>, cudaMemcpyHostToDevice);` This is copying the input data to our function. This will be the 3 channel RGB array. (rgb -> rgb_device)
    2. Call the host wrapper we wrote previously `img_rgb2gray_gpu()`. The parameters require us to pass a pointer to the ouput and a pointer to the input, these are pointers to the device memory locations.
    3. Copy the result from device -> host `cudaMemcpy(<PTR_TO_HOST_MEM>, <PTR_TO_DEVICE_MEM>, <SIZE_TO_COPY>, cudaMemcpyDeviceToHost);` This is copying the output from the device memory to the host (gray_device -> gray)
4. Compile and run the GPU code using `./lab4 2`.

GPU using unified memory

There is also memory that can be shared by host and device. The benefit of this is less code, but it is often less efficient that allocated device memory.

1. Change the device allocation to `cudaMallocManaged(&gray_device, <SIZE_TO_AOLLOCATE>)`
2. Change the matrix definition to use the `gray_device` memory. `Mat gray = Mag(HEIGHT, WIDTH, CV_8U, gray_device);`
3. Do the same changes for `rgb_device`
4. Call the wrapper function as `img_rgb2gray_gpu(gray.ptr<uchar>(), rgb.ptr<uchar>, ...);`

Notice the benefit of not having to write code to copy the data everytime. What are the computation benefits/downfalls to each method?

INVERT

Repeat the above steps, but the kernel function will invert the image (pixval = 255 - pixval).

BLUR

Repeat the above steps, but the kernel function will average a BLUR_SIZE square of pixels.