Module 3 Lab
CUDA Image Color to Grayscale

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

The purpose of this lab is to convert an RGB image into a gray scale image. The input is an RGB triple of float values and the student will convert that triple to a single float grayscale intensity value. A pseudo-code version of the algorithm is shown bellow:

```
for ii from 0 to height do
    for jj from 0 to width do
        idx = ii * width + jj
        # here channels is 3
        r = input[3*idx]
        g = input[3*idx + 1]
        b = input[3*idx + 2]
        grayImage[idx] = (0.21*r + 0.71*g + 0.07*b)
    end
end
```

PREREQUISITES

Before starting this lab, make sure that:

- You have completed the required module videos

IMAGE FORMAT

For people who are developing on their own system, the input image is stored in PPM `P6` format while the output grayscale image is stored in PPM `P5` format. Students can create their own input images by exporting their image into PPM images. The easiest way to create image is via external tools. On Unix, `bmptoppm` converts BMP images to PPM images.

INSTRUCTIONS

Edit the code in the code tab to perform the following:

- allocate device memory
- copy host memory to device
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
- copy results from device to host
- deallocate device memory

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

LOCAL SETUP INSTRUCTIONS

The most recent version of source code for this lab along with the build-scripts can be found on the Bitbucket repository. A description on how to use the CMake tool in along with how to build the labs for local development found in the README document in the root of the repository.

The executable generated as a result of compiling the lab can be run using the following command:

```
./ImageColorToGrayscale_Template -e <expected.pbm> \
    -i <input.ppm> -o <output.pbm> -t image`.
```

where `<expected.pbm>` is the expected output, `<input.ppm>` is the input dataset, and `<output.pbm>` is an optional path to store the results. The datasets can be generated using the dataset generator built as part of the compilation process.

QUESTIONS

(1) How many floating operations are being performed in your color conversion kernel? EXPLAIN.

ANSWER: **There are 3 floating point multiplications and 2 additions**

(2) Which format would be more efficient for color conversion: a 2D matrix where each entry is an RGB value or a 3D matrix where each slice in the Z axis representes a color. I.e. is it better to have color interleaved in this application? can you name an application where the oposite is true?

ANSWER: **In this case an interleaved representation is better since the data accesses are coallessed.**

(3) How many global memory reads are being performed by your kernel? EXPLAIN.

ANSWER: **There is one global memory read per pixel (i.e. three values are being read).**

(4) How many global memory writes are being performed by your kernel? EXPLAIN.

ANSWER: **The is one global memory write per pixel. We write one floating point value, since the output is a summary of the RGB values.**

(5) Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

ANSWER: **One can operate in texture memory.**

(6) Name three applications for color conversion.

ANSWER: **Color conversion is a requirement to many algorithms. Some examples are image historgram, edge detection, and image effects.**

CODE TEMPLATE

The following code is suggested as a starting point for students. The code handles the import and export as well as the checking of the solution. Students are expected to insert their code is the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
#include <wb.h>

#define wbCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
      wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));      \
      return -1;                                                          \
    }                                                                     \
  } while (0)

//@@ INSERT CODE HERE

int main(int argc, char *argv[]) {
  wbArg_t args;
  int imageChannels;
  int imageWidth;
  int imageHeight;
  char *inputImageFile;
  wbImage_t inputImage;
  wbImage_t outputImage;
  float *hostInputImageData;
  float *hostOutputImageData;
  float *deviceInputImageData;
  float *deviceOutputImageData;

  args = wbArg_read(argc, argv); /* parse the input arguments */

  inputImageFile = wbArg_getInputFile(args, 0);
```

```
    inputImage = wbImport(inputImageFile);

    imageWidth  = wbImage_getWidth(inputImage);
    imageHeight = wbImage_getHeight(inputImage);
    // For this lab the value is always 3
    imageChannels = wbImage_getChannels(inputImage);

    // Since the image is monochromatic, it only contains one channel
    outputImage = wbImage_new(imageWidth, imageHeight, 1);

    hostInputImageData  = wbImage_getData(inputImage);
    hostOutputImageData = wbImage_getData(outputImage);

    wbTime_start(GPU, "Doing GPU Computation (memory + compute)");

    wbTime_start(GPU, "Doing GPU memory allocation");
    cudaMalloc((void **)&deviceInputImageData,
               imageWidth * imageHeight * imageChannels * sizeof(float));
    cudaMalloc((void **)&deviceOutputImageData,
               imageWidth * imageHeight * sizeof(float));
    wbTime_stop(GPU, "Doing GPU memory allocation");

    wbTime_start(Copy, "Copying data to the GPU");
    cudaMemcpy(deviceInputImageData, hostInputImageData,
               imageWidth * imageHeight * imageChannels * sizeof(float),
               cudaMemcpyHostToDevice);
    wbTime_stop(Copy, "Copying data to the GPU");

    ///////////////////////////////////////////////////////
    wbTime_start(Compute, "Doing the computation on the GPU");
    //@@ INSERT CODE HERE

    wbTime_stop(Compute, "Doing the computation on the GPU");

    ///////////////////////////////////////////////////////
    wbTime_start(Copy, "Copying data from the GPU");
    cudaMemcpy(hostOutputImageData, deviceOutputImageData,
               imageWidth * imageHeight * sizeof(float),
               cudaMemcpyDeviceToHost);
    wbTime_stop(Copy, "Copying data from the GPU");

    wbTime_stop(GPU, "Doing GPU Computation (memory + compute)");

    wbSolution(args, outputImage);

    cudaFree(deviceInputImageData);
    cudaFree(deviceOutputImageData);

    wbImage_delete(outputImage);
    wbImage_delete(inputImage);

```

```
  return 0;
}
```

CODE SOLUTION

The following is a possible implementation of the lab. This solution is intended for use only by the teaching staff and should not be distributed to students.

```
#include <wb.h>

#define wbCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                         \
      wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));      \
      return -1;                                                          \
    }                                                                     \
  } while (0)

//@@ INSERT CODE HERE

#define TILE_WIDTH 16
__global__ void rgb2gray(float *grayImage, float *rgbImage, int channels,
                         int width, int height) {
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  if (x < width && y < height) {
    // get 1D coordinate for the grayscale image
    int grayOffset = y * width + x;
    // one can think of the RGB image having
    // CHANNEL times columns than the gray scale image
    int rgbOffset = grayOffset * channels;
    float r       = rgbImage[rgbOffset];     // red value for pixel
    float g       = rgbImage[rgbOffset + 1]; // green value for pixel
    float b       = rgbImage[rgbOffset + 2]; // blue value for pixel
    // perform the rescaling and store it
    // We multiply by floating point constants
    grayImage[grayOffset] = 0.21f * r + 0.71f * g + 0.07f * b;
  }
}

int main(int argc, char *argv[]) {
  wbArg_t args;
  int imageChannels;
  int imageWidth;
  int imageHeight;
  char *inputImageFile;
  wbImage_t inputImage;
  wbImage_t outputImage;
```

```
float *hostInputImageData;
float *hostOutputImageData;
float *deviceInputImageData;
float *deviceOutputImageData;

args = wbArg_read(argc, argv); /* parse the input arguments */

inputImageFile = wbArg_getInputFile(args, 0);

inputImage = wbImport(inputImageFile);

imageWidth  = wbImage_getWidth(inputImage);
imageHeight = wbImage_getHeight(inputImage);
// For this lab the value is always 3
imageChannels = wbImage_getChannels(inputImage);

// Since the image is monochromatic, it only contains one channel
outputImage = wbImage_new(imageWidth, imageHeight, 1);

hostInputImageData  = wbImage_getData(inputImage);
hostOutputImageData = wbImage_getData(outputImage);

wbTime_start(GPU, "Doing GPU Computation (memory + compute)");

wbTime_start(GPU, "Doing GPU memory allocation");
cudaMalloc((void **)&deviceInputImageData,
           imageWidth * imageHeight * imageChannels * sizeof(float));
cudaMalloc((void **)&deviceOutputImageData,
           imageWidth * imageHeight * sizeof(float));
wbTime_stop(GPU, "Doing GPU memory allocation");

wbTime_start(Copy, "Copying data to the GPU");
cudaMemcpy(deviceInputImageData, hostInputImageData,
           imageWidth * imageHeight * imageChannels * sizeof(float),
           cudaMemcpyHostToDevice);
wbTime_stop(Copy, "Copying data to the GPU");

///////////////////////////////////////////////////////
wbTime_start(Compute, "Doing the computation on the GPU");
//@@ INSERT CODE HERE
dim3 dimGrid(ceil((float)imageWidth / TILE_WIDTH),
             ceil((float)imageHeight / TILE_WIDTH));
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
rgb2gray<<<dimGrid, dimBlock>>>(deviceOutputImageData,
                                deviceInputImageData, imageChannels,
                                imageWidth, imageHeight);
wbTime_stop(Compute, "Doing the computation on the GPU");

///////////////////////////////////////////////////////
wbTime_start(Copy, "Copying data from the GPU");
cudaMemcpy(hostOutputImageData, deviceOutputImageData,
           imageWidth * imageHeight * sizeof(float),
           cudaMemcpyDeviceToHost);
```

```
    wbTime_stop(Copy, "Copying data from the GPU");

    wbTime_stop(GPU, "Doing GPU Computation (memory + compute)");

    wbSolution(args, outputImage);

    cudaFree(deviceInputImageData);
    cudaFree(deviceOutputImageData);

    wbImage_delete(outputImage);
    wbImage_delete(inputImage);

    return 0;
}
```