ASHISH TIWARI

002364321

Project4.cpp

Implementing RGB to grayscale image sequentially

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

// Use the library namespace to ease the declarations afterward.
using namespace cimg_library;
using namespace std;

int main() {


	CImg<unsigned char> image("images/flower.jpg"),
		gray(image.width(), image.height(), 1, 1, 0),
		grayWeight(image.width(), image.height(), 1, 1, 0),
		imgR(image.width(), image.height(), 1, 3, 0),
		imgG(image.width(), image.height(), 1, 3, 0),
		imgB(image.width(), image.height(), 1, 3, 0);

	// for all pixels x,y in image
	cimg_forXY(image, x, y) {
		imgR(x, y, 0, 0) = image(x, y, 0, 0),    // Red component of image sent to
imgR
			imgG(x, y, 0, 1) = image(x, y, 0, 1),    // Green component of image
sent to imgG
			imgB(x, y, 0, 2) = image(x, y, 0, 2);    // Blue component of image
sent to imgB


		int R = (int)image(x, y, 0, 0);
		int G = (int)image(x, y, 0, 1);
		int B = (int)image(x, y, 0, 2);
		// Arithmetic addition of channels for gray
		int grayValue = (int)(0.33*R + 0.33*G + 0.33*B);
		// Real weighted addition of channels for gray
```

```
            int grayValueWeight = (int)(0.299*R + 0.587*G + 0.114*B);
            // saving píxel values into image information
            gray(x, y, 0, 0) = grayValue;
            grayWeight(x, y, 0, 0) = grayValueWeight;
        }

        // 4 display windows, one for each image
        CImgDisplay main_disp(image, "Original"),
            draw_dispR(imgR, "Red"),
            draw_dispG(imgG, "Green"),
            draw_dispB(imgB, "Blue"),
            draw_dispGr(gray, "Gray");
        //draw_dispGrWeight(grayPond, "Gray (Weighted)");

        // wait until main window is closed
        while (!main_disp.is_closed()) {
            main_disp.wait();
        }

        return 0;
}
```

Implementing RGB to grayscale image parallelly

```
#include <iostream>
#include "timer.h"
#include "utils.h"
#include <string>
#include <stdio.h>

size_t numRows();  //return # of rows in the image
size_t numCols();  //return # of cols in the image

void preProcess(uchar4 **h_rgbaImage, unsigned char **h_greyImage,
            uchar4 **d_rgbaImage, unsigned char **d_greyImage,
            const std::string& filename);

void postProcess(const std::string& output_file);

void your_rgba_to_greyscale(const uchar4 * const h_rgbaImage, uchar4 * const
d_rgbaImage,
                  unsigned char* const d_greyImage, size_t numRows, size_t
numCols);

//include the definitions of the above functions
```

```
#include "HW.cpp"

int main(int argc, char **argv) {
  uchar4        *h_rgbaImage, *d_rgbaImage;
  unsigned char *h_greyImage, *d_greyImage;

  std::string input_file;
  std::string output_file;
  if (argc == 3) {
    input_file  = std::string(argv[1]);
    output_file = std::string(argv[2]);
  }
  else {
    std::cerr << "Usage: ./hw input_file output_file" << std::endl;
    exit(1);
  }
  //load the image and give us our input and output pointers
  preProcess(&h_rgbaImage, &h_greyImage, &d_rgbaImage, &d_greyImage, input_file);

  GpuTimer timer;
  timer.Start();
  //call the grayscale code
  your_rgba_to_greyscale(h_rgbaImage, d_rgbaImage, d_greyImage, numRows(),
numCols());
  timer.Stop();
  cudaDeviceSynchronize(); checkCudaErrors(cudaGetLastError());
  printf("\n");
  int err = printf("%f msecs.\n", timer.Elapsed());

  if (err < 0) {
    //Couldn't print! Probably the student closed stdout - bad news
    std::cerr << "Couldn't print timing information! STDOUT Closed!" << std::endl;
    exit(1);
  }

  //check results and output the grey image
  postProcess(output_file);

  return 0;
}
```

HW.cpp

```cpp
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/opencv.hpp>
#include "utils.h"
#include <cuda.h>
#include <cuda_runtime.h>
#include <string>

cv::Mat imageRGBA;
cv::Mat imageGrey;

uchar4        *d_rgbaImage__;
unsigned char *d_greyImage__;

size_t numRows() { return imageRGBA.rows; }
size_t numCols() { return imageRGBA.cols; }

//return types are void since any internal error will be handled by quitting
//no point in returning error codes...
//returns a pointer to an RGBA version of the input image
//and a pointer to the single channel grey-scale output
//on both the host and device
void preProcess(uchar4 **inputImage, unsigned char **greyImage,
                uchar4 **d_rgbaImage, unsigned char **d_greyImage,
                const std::string &filename) {
  //make sure the context initializes ok
  checkCudaErrors(cudaFree(0));

  cv::Mat image;
  image = cv::imread(filename.c_str(), CV_LOAD_IMAGE_COLOR);
  if (image.empty()) {
    std::cerr << "Couldn't open file: " << filename << std::endl;
    exit(1);
  }

  cv::cvtColor(image, imageRGBA, CV_BGR2RGBA);

  //allocate memory for the output
  imageGrey.create(image.rows, image.cols, CV_8UC1);

  //This shouldn't ever happen given the way the images are created
  //at least based upon my limited understanding of OpenCV, but better to check
  if (!imageRGBA.isContinuous() || !imageGrey.isContinuous()) {
```

```
    std::cerr << "Images aren't continuous!! Exiting." << std::endl;
    exit(1);
  }

  *inputImage = (uchar4 *)imageRGBA.ptr<unsigned char>(0);
  *greyImage  = imageGrey.ptr<unsigned char>(0);

  const size_t numPixels = numRows() * numCols();
  //allocate memory on the device for both input and output
  checkCudaErrors(cudaMalloc(d_rgbaImage, sizeof(uchar4) * numPixels));
  checkCudaErrors(cudaMalloc(d_greyImage, sizeof(unsigned char) * numPixels));
  checkCudaErrors(cudaMemset(*d_greyImage, 0, numPixels * sizeof(unsigned char)));
//make sure no memory is left laying around

  //copy input array to the GPU
  checkCudaErrors(cudaMemcpy(*d_rgbaImage, *inputImage, sizeof(uchar4) *
numPixels, cudaMemcpyHostToDevice));

  d_rgbaImage__ = *d_rgbaImage;
  d_greyImage__ = *d_greyImage;
}

void postProcess(const std::string& output_file) {
  const int numPixels = numRows() * numCols();
  //copy the output back to the host
  checkCudaErrors(cudaMemcpy(imageGrey.ptr<unsigned char>(0), d_greyImage__,
sizeof(unsigned char) * numPixels, cudaMemcpyDeviceToHost));

  //output the image
  cv::imwrite(output_file.c_str(), imageGrey);

  //cleanup
  cudaFree(d_rgbaImage__);
  cudaFree(d_greyImage__);
}
```

2. **Implement both a parallel version of this algorithm in C/C++ and OpenCL, and also a sequential version in C or C++.**

   a. **Compare the performance of the parallel version and the sequential version. Is the parallel version faster than the sequential version?**

For processing on image, operations must be performed on each pixel. If these operations are performed sequentially it will take too much time. So, to reduce the time, there is need of parallel processing on all the pixels. So that instead of operating on each pixel one by one, operations on all the pixels is done parallelly at a time. By performing Parallel operations speed of processing is increased significantly as compared to sequential one. So, it will also help to perform video processing in faster way. For parallel processing NVIDIA Graphics card is used. Parallel algorithm is performed on CUDA C platform. The parallel implementation of the algorithm Color2Gray is faster than the sequential version.

   b. **Vary the number of work items. Does it make a difference in performance (i.e. time)?**

The parallel implementation of the algorithm Color2Gray greatly improved the time of execution while maintaining the quality of the original implementation, coming to be more than 200 times faster than the traditional implementation in some cases.

   c. **Compare the performance of the parallel version of your algorithm on CPU and GPU. Which one is faster?**

With an increase in the image size, the GPU implementation tends to be faster than the CPU version, while the CPU version rapidly slows down with an increase in size.

d. Include a brief report to describe your experiments and results.

The RGB to Greyscale algorithm has been implemented in C code which is executed sequentially, and in CUDA C code which is executed parallelly. Because of serial code, in C the operation is performed on pixels one by one. There are lacs of pixel in one image. So more time is required for serial code. In contrast, for parallel algorithm, operation on all the pixels is performed simultaneously at a time. So time required for processing reduces enormously.

Algorithm	Time taken for processing
Sequential	0.140msec
Parallel	0.056msec