GPGPU Assignment #3

Author: Yu-Sheng Lin Instructor: Wei-Chao Chen

July 2, 2017

1 Goals

Implement Poisson Editing for image cloning on the GPUs. Go fancy and implement faster algorithms to gain the bonus points.

2 Description

2.1 Image Cloning

Figure 1 shows input examples for image cloning algorithms, where (a) is the background, (b) is the target image, and (c) is a boolean mask. You may perform naive image cloning such as Figure 2 by copying and pasting the image from the target image to the background image according to the binary mask. This algorithm is provided in the sample codes.

Obviously there are rooms for improvement from this naive algorithm. In this assignment, we ask you to implement Poisson Editing, a computationally intensive but effective algorithm that is designed to seamlessly blend the background and target images by means of differential domain operations. The algorithm is described in more details later in this document, and we encourage you to read the original paper *Poisson Image Editing* from SIGGRAPH 2003 by P. Perez et al.

2.2 Function Signature

The function signature for this assignment is Listing 1. `background` and `output` are interlaced RGB images of size $W_b \times H_b$ and range from 0 to 255. `target` is the same except that its size is $W_t \times H_t$. `mask` is the mask that contains only one color channel and we use `0.0f/255.0f` to represent false/true.

```
void PoissonImageCloning(
    const float *background,
    const float *target,
    const float *mask,
    float *output,
    const int wb, const int hb, const int wt, const int ht,
    const int oy, const int ox
);
```

Listing 1: Function signature

(a) The background image $W_b \times H_b$.

(b) The target image which will be pasted to the background, $W_t \times H_t$.

(c) The mask $W_t \times H_t$.

Figure 1: The input images of this assignment.

We will also assign the offset O_y, O_x, which means the offset of the target image in the background image (from the top-left corner), and we will test your program using these two commands.

```
./a.out img_background.ppm img_target.ppm img_mask.pgm 130 600 output.ppm
./a.out img_background.ppm img_target.ppm img_mask.pgm 130 900 output.ppm
```

Listing 2: Execute your code

We use the very popular PGM/PPM image format, which can be edited by many image processing softwares. You can generate new test cases if you wish.

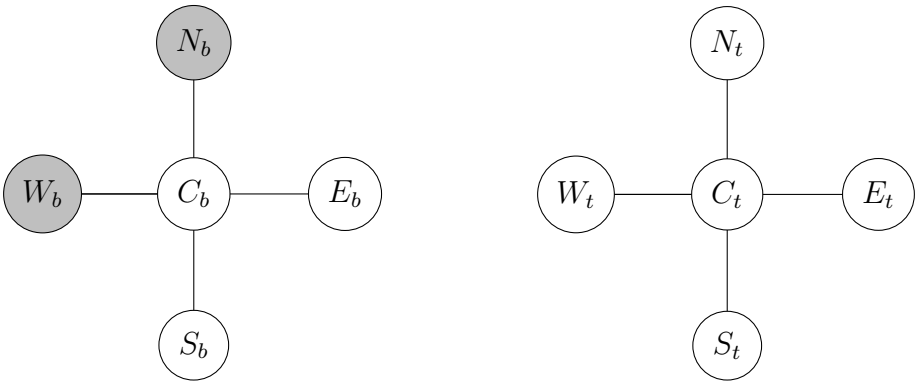
2.3 Poisson Editing

If you wish to bypass the mathematical details in the original paper, you may proceed by implementing this 4-neighbor linear system in Equation 1 (also refer to Figure 3).

$$4C_b - (S_b + E_b) = 4C_t - (N_t + W_t + S_t + E_t) + (N_b + W_b). \tag{1}$$

With the **Jacobi Iteration**, the iteration step is in Equation 2, where C_b' is the value of the next step and S_b, E_b is the value of current step. You may also notice that C_b' is independent

Figure 2: A very naive and ineffective image cloning output.



(a) The values to be solved.

(b) The corresponding target image.

Figure 3: The gray nodes are the boundary (the black pixels in the mask).

of C_b but only depends on it's neighbors.

$$C'_b = \frac{1}{4}\left[\underbrace{4C_t - (N_t + W_t + S_t + E_t) + (N_b + W_b)}_{\text{Fixed during iterations}} + \underbrace{(S_b + E_b)}_{\text{Current value}}\right] \tag{2}$$

It is your job to generalize and figure out the equation for (1) when the locations of gray points change, and (2) when the point has less than four neighbors.

2.4 Acceleration

This part is counted as bonus. To qualify for the bonus, you would also need to write a short report about your speed up and implementation. You may, for example, compare convergence against the number of iterations or execution time. We describe a few possible speed-up mechanisms, and our conjectures about how much you may gain from implementing these suggestions.

First, you may observe that the time for a value to propagate from the left to the right of the image is proportional to the width of the image. Therefore it may require the square of image sizes to actually converge to a proper solution. A naive implementation would require thousands of iterations to converge, which is very impractical. Figure 4 shows results with TA's baseline implementation. As you can see, it takes 20000 iterations to converge.

A cheaper solution is to use a hierarchical method. Start by solving the problem at a lower resolution, upsample, and then solve it at a higher resolution. You could do this at 1/8x, 1/4x, 1/2x and 1x scales with the nearest-neighbor upsampling algorithm, for example. Note that the number of iterations after each scale promotion would be less than solving the complete problem, because your lower-resolution solutions would look reasonably similar to the higher resolution ones already.

You may also try the **successive over-relaxation method (SOR)**, which changes the iteration steps to Equation 3,

$$C'_{b,SOR} = \omega C_b + (1 - \omega) C_{b,SOR}. \tag{3}$$

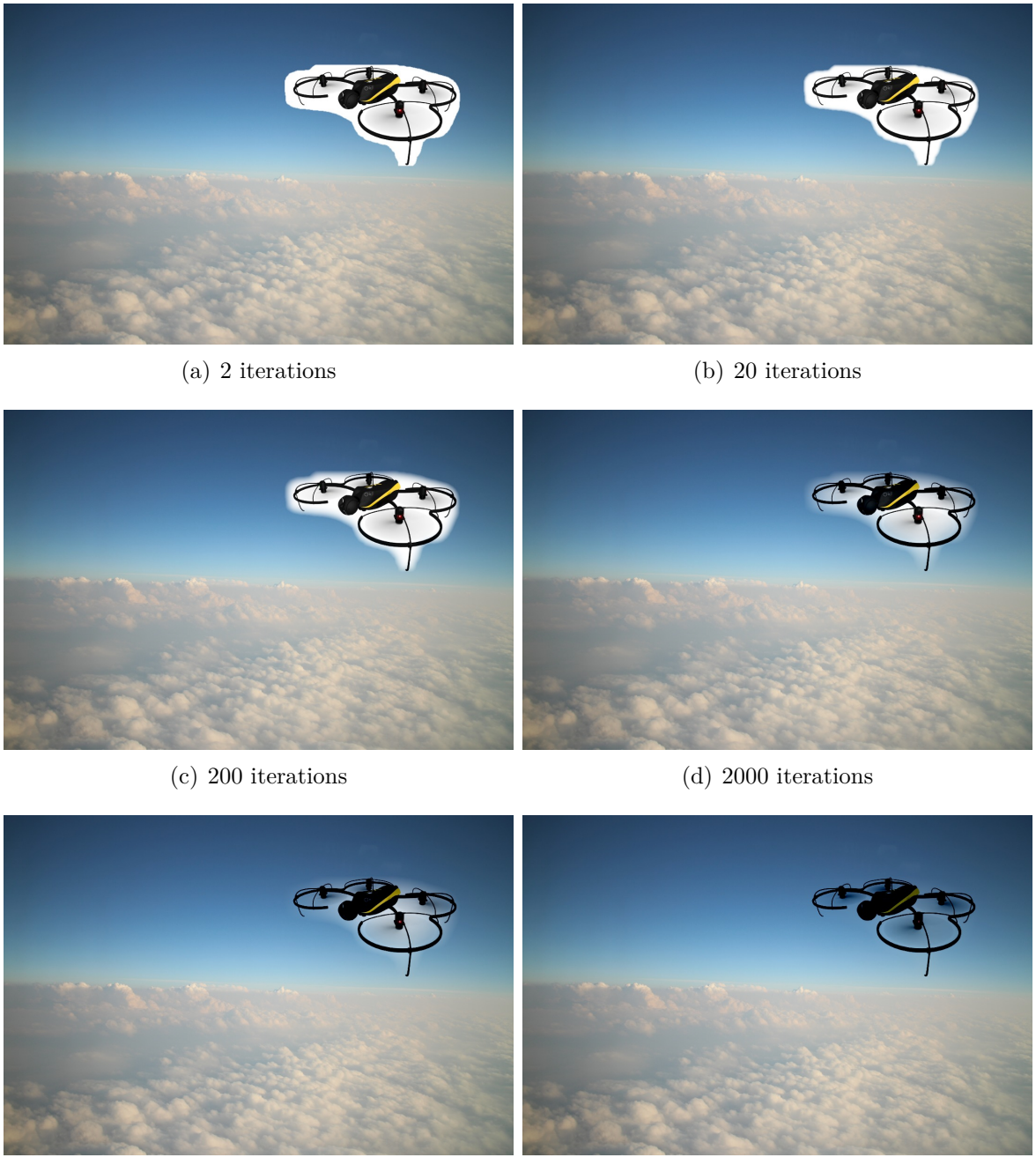
SOR is just a interpolation/extrapolation between current values and the values of the next iteration. Sadly, since the linear system is not **diagonally dominant**, the SOR Jacobi iteration diverges. (In fact, even naïve Jacobi iteration is not guaranteed to converge.) However, some students of 2017 accidentally found that the following alternative formula works perfectly in this assignment.

$$C''_{b,SOR} = \omega C'_b + (1 - \omega) C_{b,SOR}. \tag{4}$$

The new formula extrapolates against the C_b two steps before, instead of one, and this can still be implemented without an third buffer. We have confirmed that $\omega = 1.9$ works fine.

3 Grading

1. 100 points, when you finish the baseline implementation (Jacobi, no acceleration).
2. Up to 20 bonus points for SOR Jacobi.
3. Up to 40 bonus points for hierarchical Jacobi.
4. Up to 50 bonus points if you can tell why Equation 4 works magically.
5. Up to 50 extra bonus points for any other speed-up implementation.



(a) 2 iterations (b) 20 iterations

(c) 200 iterations (d) 2000 iterations

(e) 6000 iterations (f) 20000 iterations

Figure 4: The convergence with Jacobi Iteration.

4 Submission

- The submission deadline is 2017/05/21 23:59.
- Please submit the result in loseless PNG format `lab3/results/***.png`.
- Submit your source code `lab3/lab3.cu`. Again, you should only modify this file in the homework.
- We will test your code with the command listed in Listing 2.
- If you implement hierarchical, SOR, or any other speed-up algorithm, you need to submit a report `lab3/report.pdf` to be considered for the bonus.

5 Hint

Listing 3 contains part of TA's code. Feel free to use it.

```cpp
void PoissonImageCloning(
    const float *background,
    const float *target,
    const float *mask,
    float *output,
    const int wb, const int hb, const int wt, const int ht,
    const int oy, const int ox
) {
    // set up
    float *fixed, *buf1, *buf2;
    cudaMalloc(&fixed, 3*wt*ht*sizeof(float));
    cudaMalloc(&buf1, 3*wt*ht*sizeof(float));
    cudaMalloc(&buf2, 3*wt*ht*sizeof(float));

    // initialize the iteration
    dim3 gdim(CeilDiv(wt,32), CeilDiv(ht,16)), bdim(32,16);
    CalculateFixed<<<gdim, bdim>>>(
        background, target, mask, fixed,
        wb, hb, wt, ht, oy, ox
    );
    cudaMemcpy(buf1, target, sizeof(float)*3*wt*ht, cudaMemcpyDeviceToDevice);

    // iterate
    for (int i = 0; i < 10000; ++i) {
        PoissonImageCloningIteration<<<gdim, bdim>>>(
            fixed, mask, buf1, buf2, wt, ht
        );
        PoissonImageCloningIteration<<<gdim, bdim>>>(
            fixed, mask, buf2, buf1, wt, ht
        );
    }

    // copy the image back
    cudaMemcpy(output, background, wb*hb*sizeof(float)*3, cudaMemcpyDeviceToDevice);
    SimpleClone<<<gdim, bdim>>>(
        background, buf1, mask, output,
        wb, hb, wt, ht, oy, ox
    );

    // clean up
    cudaFree(fixed);
    cudaFree(buf1);
    cudaFree(buf2);
}
```

Listing 3: Hint