GPGPU LAB 3
Poisson Image Editing

宋子維
b03902042@ntu.edu.tw

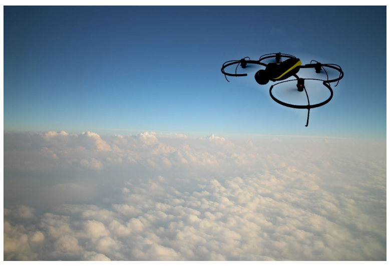
1 Baseline Method

Baseline method refers to the pure Jacobi iterative method to solve the Poisson equation. In Figure 1, you can see that it is very slow to converge, and thus, I implement two acceleration method in the following sections.

(a) 600 iterations

(b) 1000 iterations

(c) 2000 iterations


(d) 10000 iterations

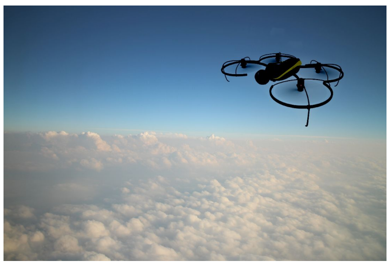

(e) 20000 iterations

Figure 1: The convergence with baseline method.

2 Successive Over-relaxing (SOR) Method

As shown in Figure 2, the convergence speed of SOR method is significantly faster than the one of baseline method. About 1000 iterations are enough to generate the same degree of seamless cloning image compared with baseline method.

As for the magical equation that TA lists, basically, **I don't think it works magically**. It is just how **Parallel SOR iterative methods** [1] works! The following equation depicts how parallel SOR works on interior grids:

First pass:

$$\phi_{i,j}^{(n+1)} = (1-\omega)\phi_{i,j}^{(n)} + \frac{\omega}{4}(\phi_{i+1,j}^{(n)} + \phi_{i,j+1}^{(n)} + \phi_{i-1,j}^{(n)} + \phi_{i,j-1}^{(n)}) \quad \text{for } i+j \text{ is odd} \tag{1}$$

Second pass:

$$\phi_{i,j}^{(n+1)} = (1-\omega)\phi_{i,j}^{(n)} + \frac{\omega}{4}(\phi_{i+1,j}^{(n+1)} + \phi_{i,j+1}^{(n+1)} + \phi_{i-1,j}^{(n+1)} + \phi_{i,j-1}^{(n+1)}) \quad \text{for } i+j \text{ is even} \tag{2}$$

Note that the the whole procedure is a two-pass process.

Back with my CUDA implementation in Listing 1, and consider two buffers a and b. To be simplified, the mathematical formula without fixed terms for each iteration can be written as follows:

First pass:

$$b_{i,j}^{(n+1)} = (1-\omega)b_{i,j}^{(n)} + \frac{\omega}{4}(a_{i+1,j}^{(n)} + a_{i,j+1}^{(n)} + a_{i-1,j}^{(n)} + a_{i,j-1}^{(n)}) \tag{3}$$

Second pass:

$$a_{i,j}^{(n+1)} = (1-\omega)a_{i,j}^{(n)} + \frac{\omega}{4}(b_{i+1,j}^{(n+1)} + b_{i,j+1}^{(n+1)} + b_{i-1,j}^{(n+1)} + b_{i,j-1}^{(n+1)}) \tag{4}$$

Note the the whole procedure is still a two-pass process, but, in each pass, the value of a or b is updated for all i, j.

It seems that the formula is a little bit different with Equation 1 and 2. However, if we do the replacement below, the whole thing makes sense:

$$a_{i,j}^{(n)} \leftarrow \phi_{i,j}^{(n)} \quad \text{for } i+j \text{ is even} \tag{5}$$

$$b_{i,j}^{(n)} \leftarrow \phi_{i,j}^{(n)} \quad \text{for } i+j \text{ is odd} \tag{6}$$

Thus, the following formula can be derived from Equation 3 and 4:

First pass:

$$\phi_{i,j}^{(n+1)} = (1-\omega)\phi_{i,j}^{(n)} + \frac{\omega}{4}(\phi_{i+1,j}^{(n)} + \phi_{i,j+1}^{(n)} + \phi_{i-1,j}^{(n)} + \phi_{i,j-1}^{(n)}) \quad \text{for } i+j \text{ is odd} \tag{7}$$

Second pass:

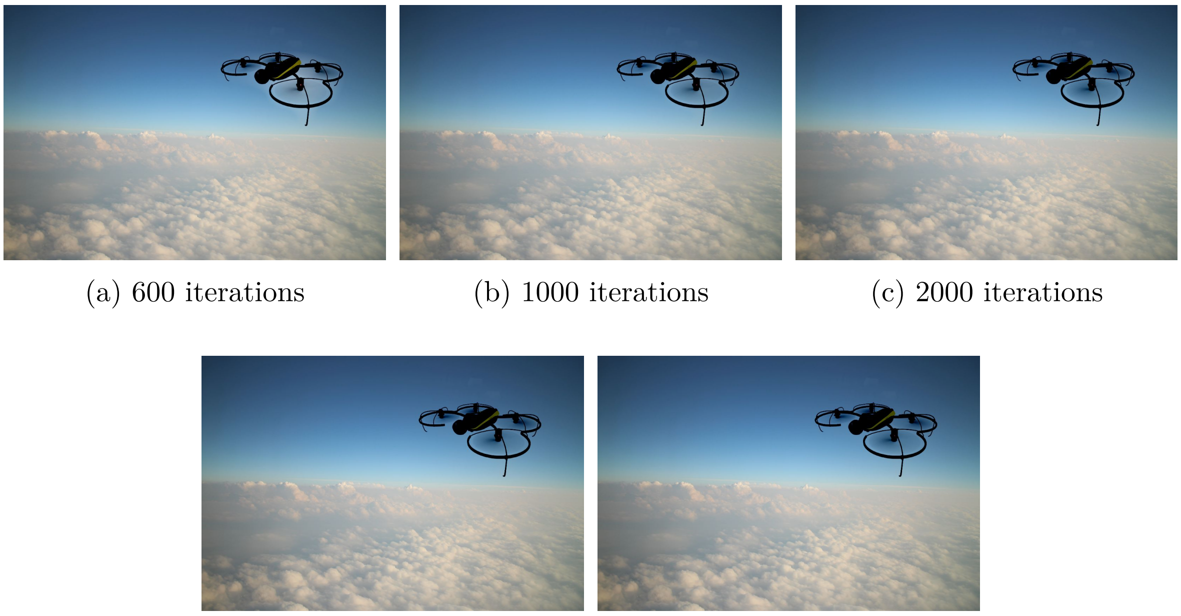
$$\phi_{i,j}^{(n+1)} = (1-\omega)\phi_{i,j}^{(n)} + \frac{\omega}{4}(\phi_{i+1,j}^{(n+1)} + \phi_{i,j+1}^{(n+1)} + \phi_{i-1,j}^{(n+1)} + \phi_{i,j-1}^{(n+1)}) \quad \text{for } i+j \text{ is even} \tag{8}$$

which are exactly the same as Equation 1 and 2. The final image we should clone back to background is the one with even-indexed pixels in a and odd-indexd pixels in b. However, in my experiment, simply cloning a back to background still works well. I think it is because a and b may be numerically equal after some iterations. Hence, I think the magical equation is not magical at all, and it is just the parallel version of SOR iteration (and does more works).

3 Hierarchical Method

In this method, we can simply follow the steps down below for each downsampling level:

1. Downsample
2. Calculate fixed values
3. Run Jacobi iterative method
4. Upsample



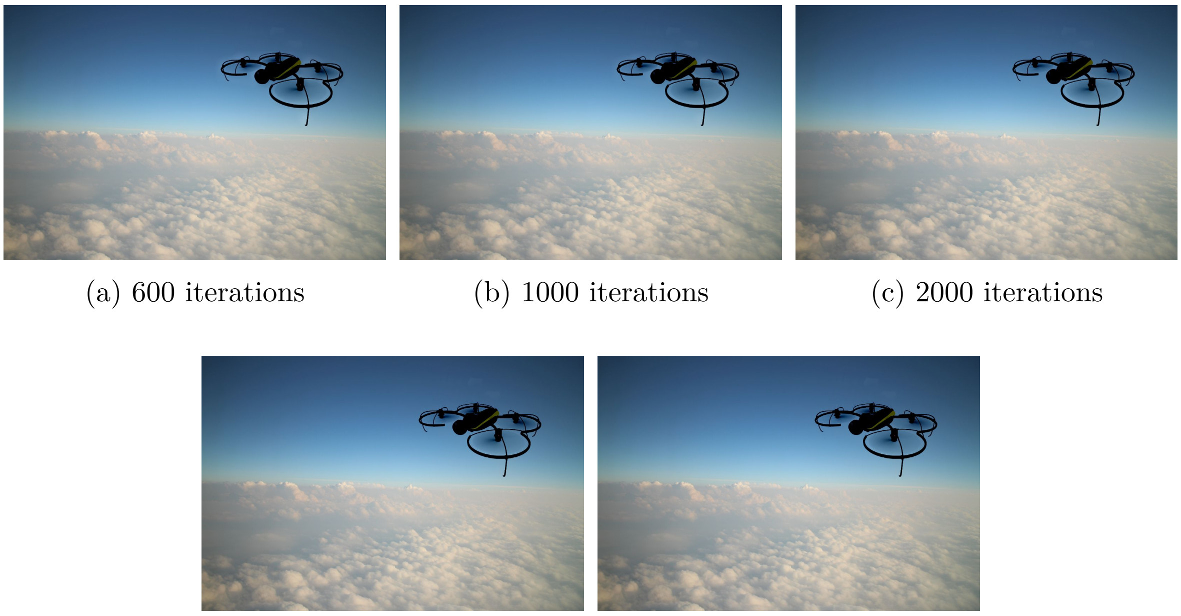
(a) 600 iterations (b) 1000 iterations (c) 2000 iterations

(d) 10000 iterations (e) 20000 iterations

Figure 2: The convergence with SOR method.

Suppose that 4-level downsampling scale ($\frac{1}{8}$x, $\frac{1}{4}$x, $\frac{1}{2}$x, 1x scale) and 10000 total iterations are applied, the iterations are equally divided for each scale in my implementation, that is, 2500 iterations for each scale in this case. As shown in Figure 3, hierarchical method with 6-level downsampling scale can converge at 600 iterations, at which, SOR method still remains some white regions.

Moreover, integration SOR method into hierarchical Jacobi iterative method can even converge faster than each individual. In my experiment, 200 iterations are large enough for the integrative method to generate comparable images.



(a) 600 iterations (b) 1000 iterations (c) 2000 iterations

(d) 10000 iterations (e) 20000 iterations

Figure 3: The convergence with hierarchical Jacobi iterative method.

References

[1] David J Evans. “Parallel SOR iterative methods”. In: *Parallel computing* 1.1 (1984), pp. 3–18.

```
__global__ void PoissonImageCloningIteration(
    const float *fixed,
    const float *mask,
    const float *input,
    float *output,
    const int wt, const int ht,
    const float omega
)
{
    const int yt = blockIdx.y * blockDim.y + threadIdx.y;
    const int xt = blockIdx.x * blockDim.x + threadIdx.x;
    const int center_idx = to_1dindex(wt, xt, yt);
    if (in_boundary(wt, ht, xt, yt) && mask[center_idx] > 127.0f) {
        int neighbor_idx;
        int xn, yn;
        float r = fixed[3 * center_idx + 0],
              g = fixed[3 * center_idx + 1],
              b = fixed[3 * center_idx + 2];
        for (int i = 0; i < 4; i++) {
            xn = xt + dir[i][0], yn = yt + dir[i][1];
            neighbor_idx = to_1dindex(wt, xn, yn);
            if (in_boundary(wt, ht, xn, yn) && mask[neighbor_idx] > 127.0f) {
                r += input[3 * neighbor_idx + 0];
                g += input[3 * neighbor_idx + 1];
                b += input[3 * neighbor_idx + 2];
            }
        }
        output[3 * center_idx + 0] = omega * r / 4 +
                                     (1 - omega) * output[3 * center_idx + 0];
        output[3 * center_idx + 1] = omega * g / 4 +
                                     (1 - omega) * output[3 * center_idx + 1];
        output[3 * center_idx + 2] = omega * b / 4 +
                                     (1 - omega) * output[3 * center_idx + 2];
    }
}
...
    cudaMemcpy(a, target, 3 * wt * ht * sizeof(float), cudaMemcpyDeviceToDevice);
    cudaMemcpy(b, target, 3 * wt * ht * sizeof(float), cudaMemcpyDeviceToDevice);
    for (int i = 0; i < 10000; i++) {
        PoissonImageCloningIteration <<<gdim, bdim>>> (
            fixed, mask, a, b, wt, ht, 1.9
        );
        PoissonImageCloningIteration <<<gdim, bdim>>> (
            fixed, mask, b, a, wt, ht, 1.9
        );
    }
    PlaceCorrectValue <<<gdim, bdim>>> (buf1, buf2, wt, ht);
```

Listing 1: SOR