ECE453 – Deblurring

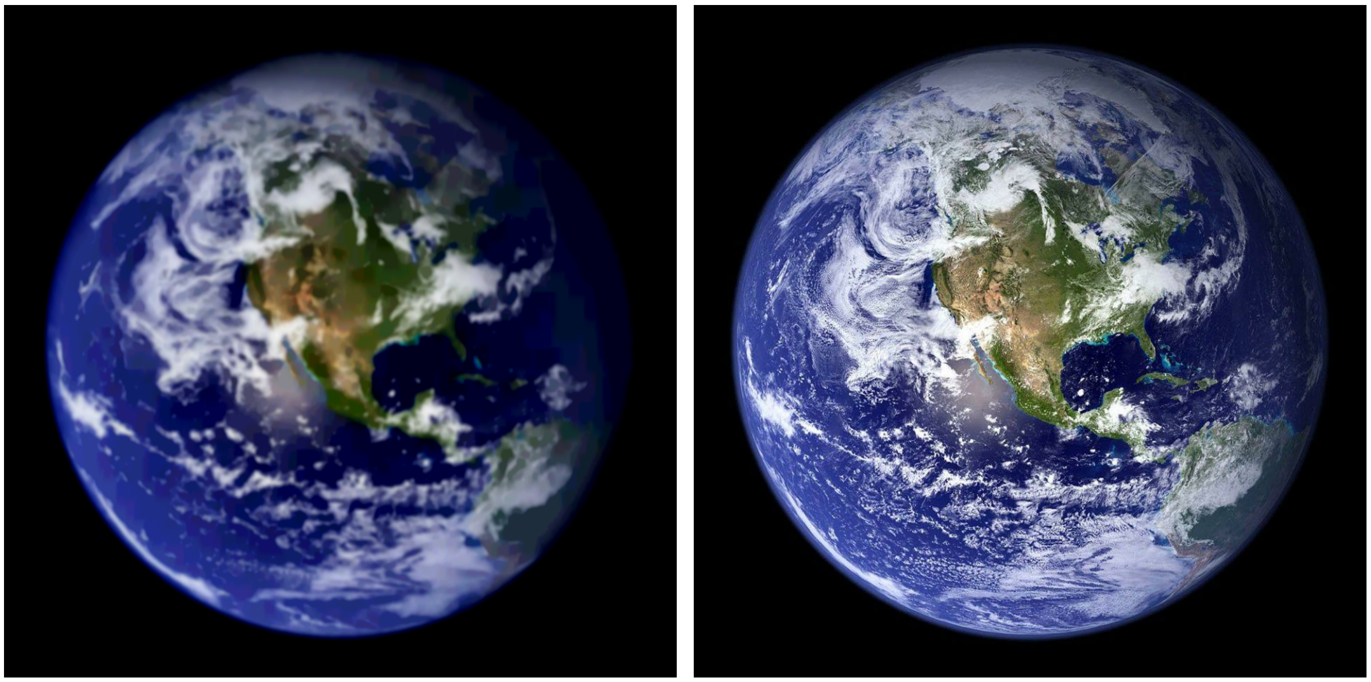
Jonathan Lam

March 21, 2021

1 Introduction

The goal of this project is to become familiar with the CUDA toolchain and use a highly-parallelized CUDA program to achieve a speedup over a traditional CPU program.

The task is to unblur the image. A clear reference image is also provided.



(a) Blurry test image

(b) Clear reference image

Figure 1: The provided images. Both are 960x960 PNG images.

2 Theory

This is not a comprehensive survey of the theory, since this is only an introductory project and the main goal was to learn CUDA, not the theory in full. The full theory is deferred to the research papers mentioned in the following sections.

2.1 Deblurring algorithm

The general idea is that a typical blurred image can be modeled as the convolution of the original image with some distorting filter (this filter is called the point spread function, or PSF). This filter can be used to model blurring caused by factors such as defocus aberration (when an image is slightly out of focus) or motion blur.

Thus the problem is broken into two parts: how to find the PSF, and, once it is known, how to deconvolve the blurry image with the PSF.

The method chosen is the Richardson-Lucy (RL) deconvolution algorithm [1, 2], which, given a known PSF, takes an iterative approach that is guaranteed to converge to the maximum-likelihood deconvoluted image. Fish et. al. developed a method to swap the roles of the original image and the PSF [3] to find the maximum-likelihood estimate for the PSF given the convolved image and the "known" (i.e., estimated) deconvolved image. This can be used iteratively as a blind deconvolution method to get maximum-likelihood estimates of both the PSF and the original deconvolved image by alternatively fixing one and estimating the other.

A paraphrasing of Fish et al's description of the reasoning behind the algorithm in layman's terms goes as follows: by Bayes' theorem, we have:

$$P(x_i \mid y) = \frac{P(y \mid x_i)P(x_i)}{\sum_j P(y \mid x_j)P(x_j)}$$

We can interpret this (very roughly) as: $P(x)$ being the true distribution (i.e., the pristine image); $P(y)$ being the convolved distribution (the blurry image); $P(y \mid x)$ is the PSF centered at x. Note that $P(x_i)$ appear on both sides of the equation – we can use this to improve our estimate using the previous estimate. By multiplying by $P(y)$ and rearranging terms we obtain:

$$P(x_i) = \sum_k P(x_i \mid y_k)P(y_k) = \sum_k \left[\frac{P(y_k)}{\sum_j P(x_j)P(y_k \mid x_j)} P(y_k \mid x_i)\right] P(x_i)$$

The additional sum is added because the pixel at x is based on a range of pixels in the deconvolved image. The summations translate to convolutions, and this can be written as:

$$f_{i+1}(x) = \left\{\left[\frac{c(x)}{f_i(x) \otimes g(x)}\right] \otimes g(-x)\right\} f_i(x)$$

following Fish et al's notation.

The "blind" part of the algorithm, i.e., estimating the PSF works the same way, but swaps the roles of g (the PSF) and f (the image). The formula as stated in Fish et al. is

$$g_{i+1}(x) = \left\{ \left[\frac{c(x)}{g_i(x) \otimes f(x)} \right] \otimes f(-x) \right\} g_i(x)$$

Now that we have a method of solving for the (next iteration of) the deblurred image and the (next iteration of) the PSF, we alternate between solving for one and then the other.

I was originally going to try the method used in Fish et al. to do the full blind deconvolution (i.e., also estimating the PSF), but this proved too time-consuming a task, so I stuck with the ordinary RL method using a fixed (chosen) PSF. For this implementation, circular (2D) Gaussian filters with various standard deviations were tested, and for the given test image a standard deviation of roughly $\sigma = 3$ appeared to produce the best results. The filter dimension used for this value of σ is roughly $\lceil 6\sigma \rceil$ so as to not lose too much power (i.e., so that the filter roughly has unity norm).

2.2 Evaluation

To evaluate correctness of the algorithm, a program was written to calculate the elementwise squared error between two images (where the "element" is each data channel of each pixel). This is used both to calculate the error between the blurry and deblurred images and the original (to show that deblurring does not make the image substantially less faithful to the original) and between the CPU and CUDA versions (to show that the two implementations are equivalent in output).

To evaluate the effectiveness of the deblurring, a method similar to that in an answer by @Niki on Stack Overflow that suggests convolution of the image by a Laplacian of Gaussian (LoG) filter. This filter is essentially an edge detection filter and will have values of larger magnitude near areas with a steeper gradient. While @Niki's use suggests the idea of a "general blurry" metric that works across dissimilar images, we are only comparing similar images; as a result, I chose instead to use the average value of the image convolved with the LoG filter (as opposed to the 99.9th percentile value) to get a better estimate of the relative blurriness across the image.

3 Implementation

The source code for this project is located at `https://github.com/jlam55555/ece465-adv-comp-arch`.

3.1 CUDA version

The general algorithm flow involves: loading in the image to a (contiguous) buffer; stripping alpha values (for PNG images); converting integers to floats; allocating space of the same size on the GPU for the image buffer, as well as temporary buffers for intermediate values; iterating over the RL algorithm; copying the image back from device memory; and writing the image to a file.

The RL algorithm is fairly straightforward (requiring two convolution operations and two elementwise multiplication/division operations). The convolution takes two arguments (an input image and a filter) and outputs to a third buffer; the output has the same size as the input image (because all operands in the algorithm must be the same size for the pointwise multiplication to work), and the filter must be centered at index zero so that the convolution does not shift the original image. This requires bounds checking and a transformation of the filter's coordinates to zero-centered coordinates.

There are four (CUDA) kernels used in total: two to convert images to and from bytes and floats (and also strip/restore the alpha value); the 2D convolution; and the elementwise multiplication/division.

3.2 CPU version

The CPU-only version is very similar to the CUDA version, and the data types are identical (all calculations are done in 32-bit floating-point and cast back to 8-bit pixel values). Thus we expect a very small error between the outputs of the two algorithms, and this is indeed the case.

The code is somewhat shorter because there is no need to create separate host- and device-versions of the input buffer.

3.3 Benchmarking and image I/O

A very simple benchmarking implementation was created using the standard library's `time.h` utilities. It allows the user to create several timer objects with a stopwatch "lap" function, and stores the average time taken for different operation types in an array.

There was some trouble getting this to work on the CUDA implementations, since the CUDA kernels run asynchronously to the C code – this required the use of the `cudaDeviceSynchronize()` method. This likely adds a small but unnoticeable decrease in the CUDA implementation's performance. While this worked on my local machine, this method did not successfully keep track of time on the Jetson Nano for unknown purposes – this requires future investigation.

For the time being, a less accurate estimate of the time taken is found using the Unix `time` shell command.

Image I/O was handled through the use of the libpng library using sample I/O code by Guillaume Cottenceau. This was chosen because libpng is fairly standard and easy to install (as opposed to more capable libraries like ImageMagick's C/C++ API). However, this means that only PNG images are supported in this version.

3.4 Evaluation of error/correctness

This was implemented quite simply by looping through all pixels of two images and calculating the mean squared error between each corresponding data value (excluding the alpha channel).

3.5 Evaluation of blurriness

The bluriness metric was implemented by convolving (reusing the 2D convolution function) the input image with a 3×3 LoG kernel, and then taking the mean squared value of the convolved image.

4 Results

See the GitHub repository README for detailed build and run instructions.

The implementations were tested on a local machine and the school's Jetson Nano. The local machine has an Intel i7-2600 CPU (x86_64 architecture, up to 3.8GHz single-core clock speed) with a Nvidia GT740 (Kepler architecture, 384 CUDA cores) GPU. The Nano has a Cortex-A57 CPU (1.4GHz, aarch64, up to 1.4GHz) with a Maxwell architecture, 128 CUDA cores GPU.

The performance results of running both the CPU and GPU (CUDA) versions with 25 iterations and the best value of σ are shown in Table 1. Performance depends on the exact model of GPU and CPU, but there is a huge gap between the performance of the two CPUs and those of the two GPUs. For the two CPU-GPU pairs on the test systems, there are 37× and 110× speedups. (The speedup on the Nano is expectedly larger, as this is a machine dedicated to GPU-heavy tasks). Most of the time and computation spent in each iteration occurs in the convolutions, and this is also where the large speedup occurs.

The MSE between the CPU and CUDA version for $\sigma = 3$ is 0.000014, which is small enough to be unrecognizable. Thus we conclude the two are equivalent.

The MSE between each image and the clear reference image, as well as the sharpness metric, is shown in Table 2. Increasing σ monotonically decreases the fidelity of the deblurred image, but this is fine as we don't expect a very good result if the image was initially blurred with a large radius. All that matters for the MSE is that it is sufficiently close to that of the input blurry image.

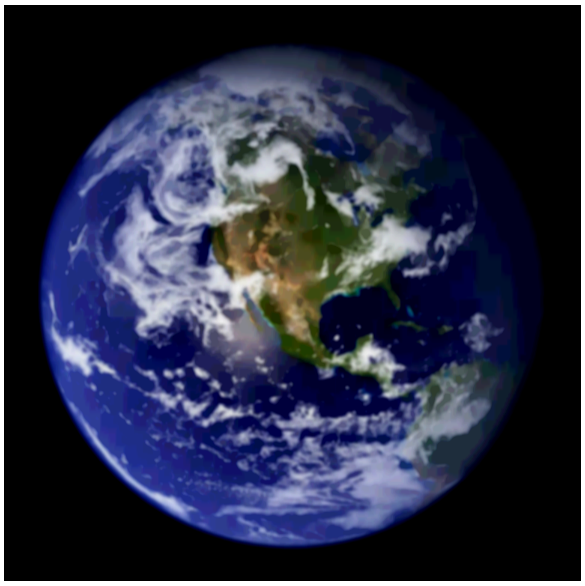
The sharpness is not monotonic with respect to σ. The best σ should be that which best matches the gaussian estimate of the original PSF.

	i7-2600 (CPU)	**Cortex-A57 (CPU)**	**GT740 (CUDA)**
Overall	219	880	5.93
RL Iteration	8.76	35.2	0.237
2D Conv kernel	4.37	17.6	0.114
Mult/Div kernel	0.0107	0.0350	0.00438

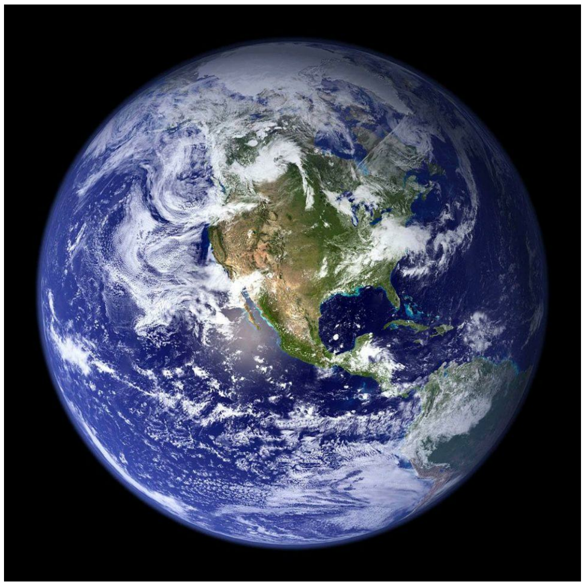
Table 1: Average time (s) of operations on the CPU and GPU implementations. Tests were performed with $\sigma = 3$ (19×19 filter). Timing did not work correctly on the Nano CUDA implementation; a rough measurement taken is 8s overall.

	Original	**Blurry**	**CPU**	**CUDA**			
			$\sigma = 3$	$\sigma = 1$	$\sigma = 2$	$\sigma = 3$	$\sigma = 4$
MSE	0	1247	1380	1264	1274	1380	1538
Sharpness	1695	8	53	23	37	53	51

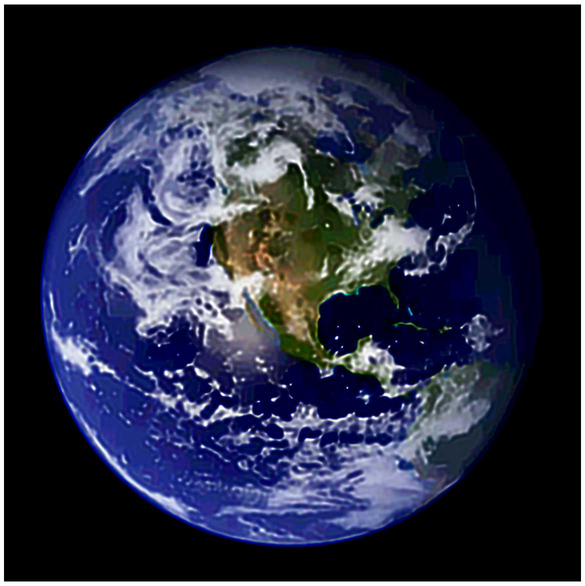
Table 2: MSE and sharpness values, as defined in 2.2, of the input (clear and blurry) image and the input images.



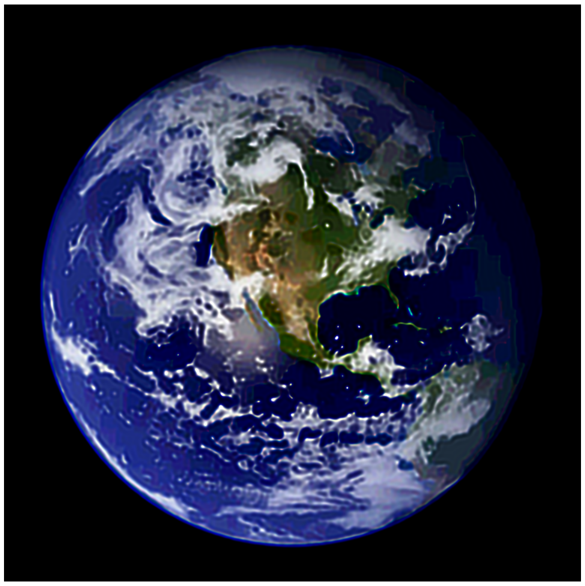
(a) Blurry test image



(b) Clear reference image

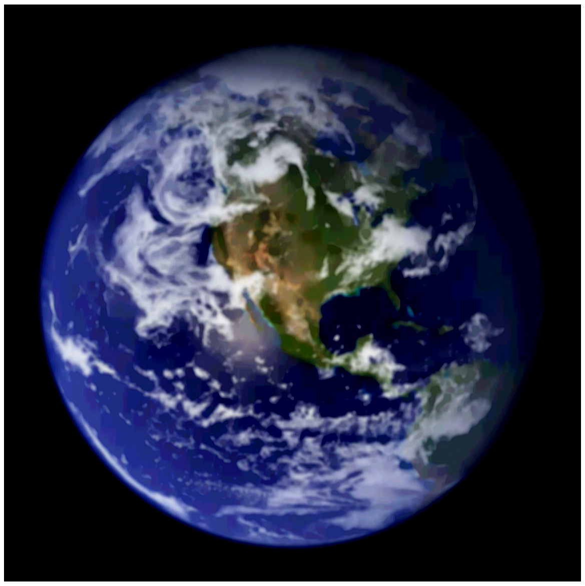


(c) CUDA deblurring

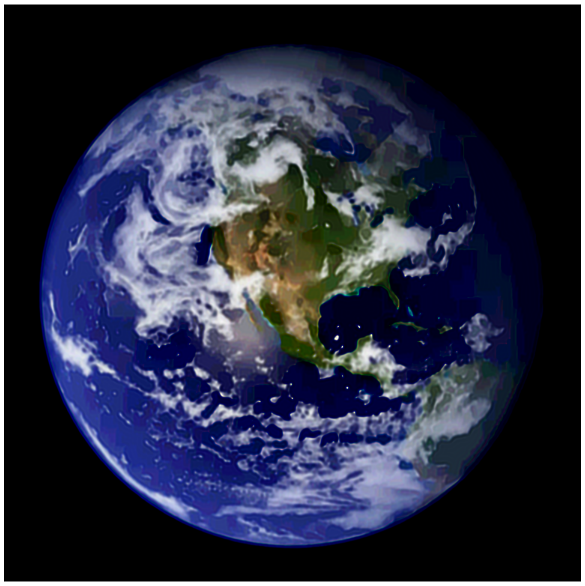


(d) CPU deblurring

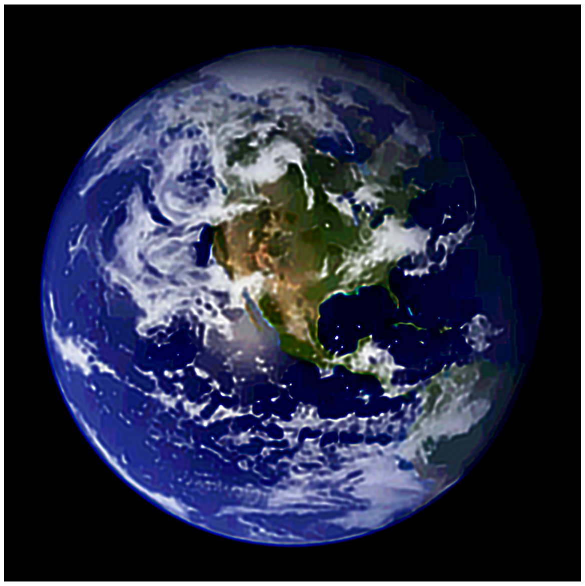
Figure 2: The provided images vs. the deblurred images with 25 iterations and $\sigma = 3$. The CUDA and CPU versions are indistinguishable. (The difference is best seen on a large screen with the images on GitHub.)



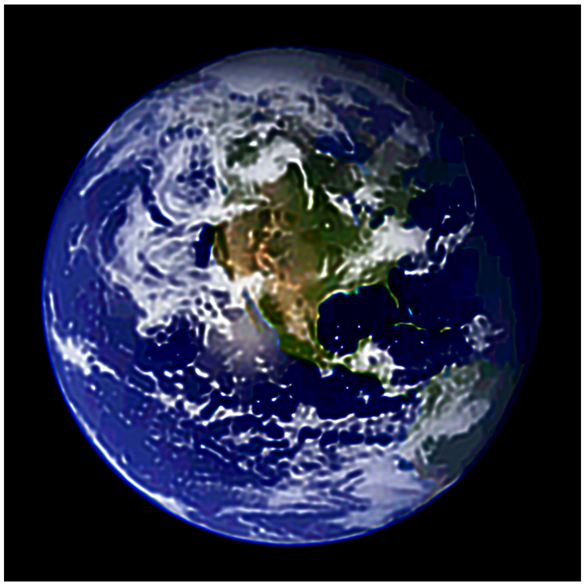
(a) $\sigma = 1$



(b) $\sigma = 2$



(c) $\sigma = 3$



(d) $\sigma = 4$

Figure 3: Deblurred images with 25 iterations of RL using the CUDA implementation, but with different values for σ. The image gets slightly clearer going from $\sigma = 1$ to $\sigma = 2$ and from $\sigma = 2$ to $\sigma = 3$, but excessive ringing begins to appear at $\sigma = 4$ (which likely contributes to the decrease in the sharpness.

5 Conclusions

The programs were fairly successful at demonstrating the speedup that GPUs can offer. This project was also successful in getting the student more familiar with writing kernel functions in CUDA rather than using a loop-based or CPU multi-threaded approach. It was also informative learning about blind deconvolution, the RL-method, and LoG filters.

References

[1] L. B. Lucy. An iterative technique for the rectification of observed distributions. *Astronomical Journal*, 79:745, June 1974.

[2] William Hadley Richardson. Bayesian-based iterative method of image restoration∗. *J. Opt. Soc. Am.*, 62(1):55–59, Jan 1972.

[3] D. A. Fish, A. M. Brinicombe, E. R. Pike, and J. G. Walker. Blind deconvolution by means of the Richardson-Lucy algorithm. *Journal of the Optical Society of America A*, 12(1):58–65, January 1995.