Richardson-Lucy in ArrayFire (Algorithm Only)

One big challenge in high performance computing is balancing simple software design with speed, as the two often work against one another. At one extreme are interpreted languages, like MATLAB and Python, which facilitate rapid prototyping at the expense of speed and memory efficiency. At the other extreme are low-level C/C++ APIs, like OpenCL and CUDA, which allow a programmer to reach maximum performance at the expense of making the software incomprehensible to novice developers.

I've personally designed an API which balances these extremes. My design considerations include:

- Portability across operating systems
- Seamless integration between native C, CUDA, and OpenCL implementations
- Support for multi-device processing
- Minimizing memory footprint
- Reaching full transfer bandwidth to compute devices
- Achieving close to peak compute performance

It's not a trivial task; however, it's a balance that any high performance software engineer will have to make. Fortunately there is a variety of software packages that offer developers different balances between efficiency and ease of use. Here's a few examples:

- MATLAB's parallel toolbox, which supports CUDA and OpenCL.
- PyCUDA and PyOpenCL CUDA and OpenCL wrappers for Python.
- OpenCV offers generic and several computer vision algorithms in OpenCL and CUDA.
- ArrayFire offers generic algorithms in OpenCL and CUDA.

This article is the first in a series about implementing Richardson-Lucy deconvolution in ArrayFire, and will focus on creating a numerically stable implementation.

2D Richardson-Lucy Algorithm:

I'll leave a detailed explanation of the algorithm to Wikipedia (LINK), however the premise is that an image taken by a camera is a distortion of a real-world object. If the distortion is minor then the image is going to appear to be very similar to the actual object. Likewise, a large distortion can cause the image to look nothing like the real object. For example, think of how mirrors in a fun house at a carnival can cause you to appear much taller or wider than you actually are. If the distortion is some known quantity, then it stands to reason that the distortion could be mathematically removed from the image.

Richardson-Lucy aims to accomplish this by modeling the image acquisition process as a mathematical convolution of an object with a point-spread function (PSF). This is the general algorithm without any statistical theory:

1. Convolve an estimate of the object with the PSF to create a simulated image
2. Divide the measured image by this simulated image to get a ratio of how far off you are
3. Convolve this ratio with the "opposite" of the PSF, to make that this ratio is properly spread across the pixels that actually contributed to it
4. Multiply the estimated object against this ratio to get a better estimate
5. Iterate to your hearts content

The ultimate goal is to come up with an estimate that, when convolved with the PSF, creates an image that matches the acquired image. So for an NxN image with an MxM PSF you can expect Richardson-Lucy to cost at minimum:

- 2 - 2D convolutions of an NxN image with an MxM PSF
- 1 - NxN element wise division
- 1 - NxN element wise multiplication

Implementation Considerations

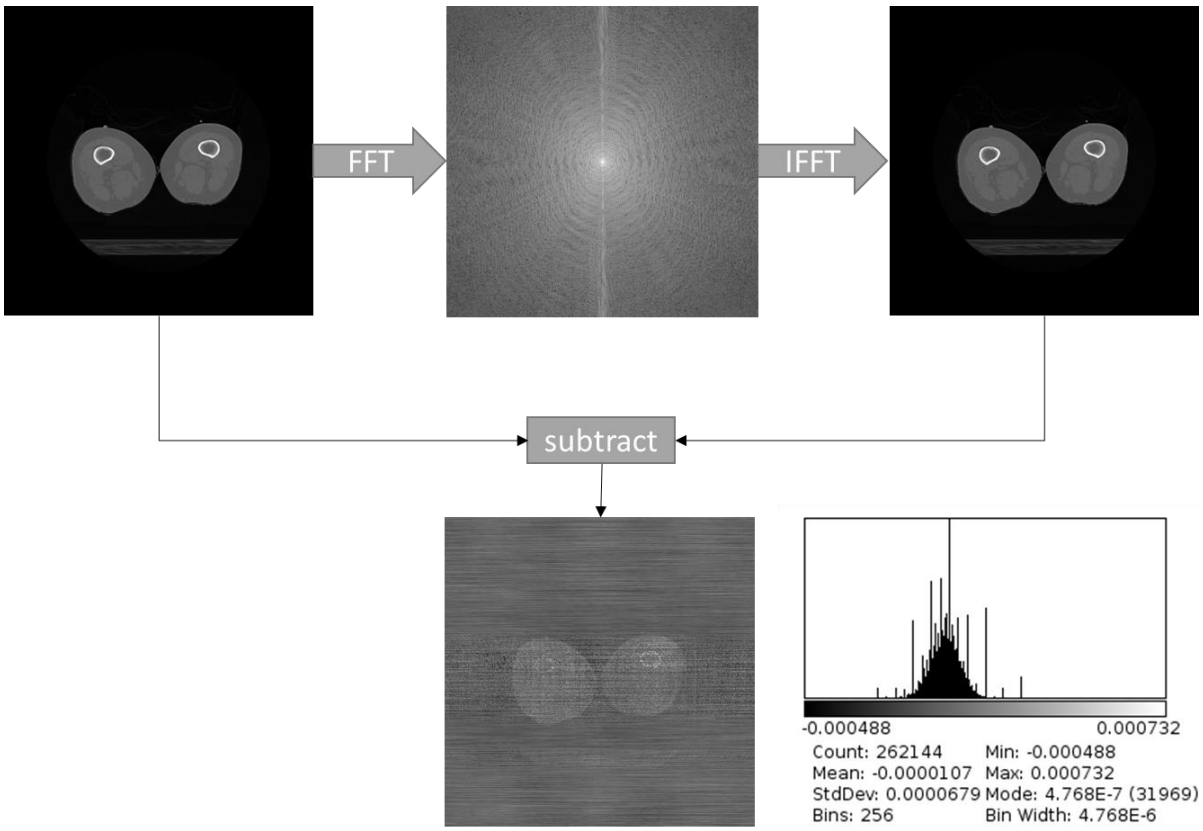
"In theory, there is no difference between theory and practice. But, in practice, there is."
~ Jan L. A. van de Snepscheut

Using Fast Fourier Transform (FFT) accelerated convolution:

ArrayFire automatically chooses between spatial or FFT accelerated convolution depending on the image and PSF dimensions, as using FFTs is not always faster. A key issue with using FFTs is they are more prone to errors cause by precision limitations.

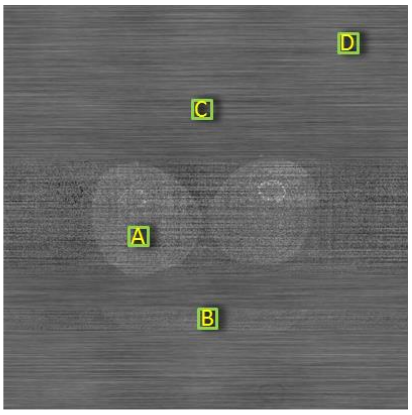
Computers, much like any instrument, have limited accuracy. Computing units like most CPUs and GPUs natively support so-called single (32-bit) and double (64-bit) precisions, and as such most FFT implementations support either degree of accuracy. The balance between single and double precisions is speed against accuracy, but it's important to note that how much the speed trade-off is depends on the compute device.

To illustrate the impact of precision limitations when using FFTs, consider performing a forward FFT on a signal and then an inverse FFT (IFFT) to recover the original signal. In theory you should end up with the original signal, in practice it's usually not exactly the same:



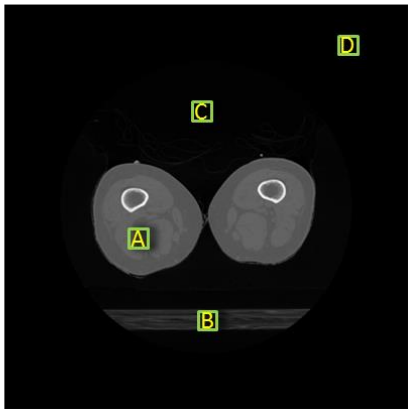
Of course an important question is exactly how much of a problem these precision limitations are:

	Mean
A	4.38E-05
B	2.78E-05
C	-2.04E-05
D	1.47E-05



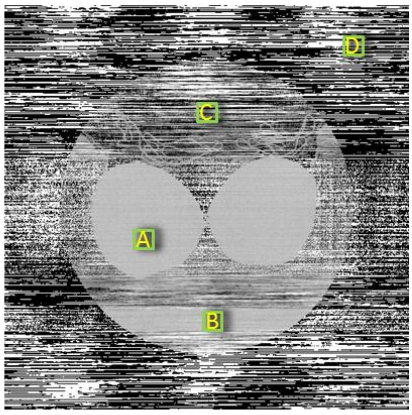
divide

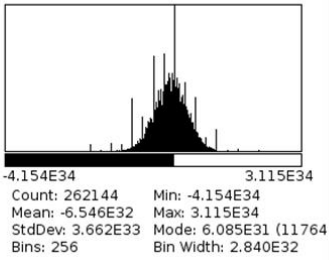
	Mean
A	1019.01
B	395.66
C	22.43
D	0.00



=

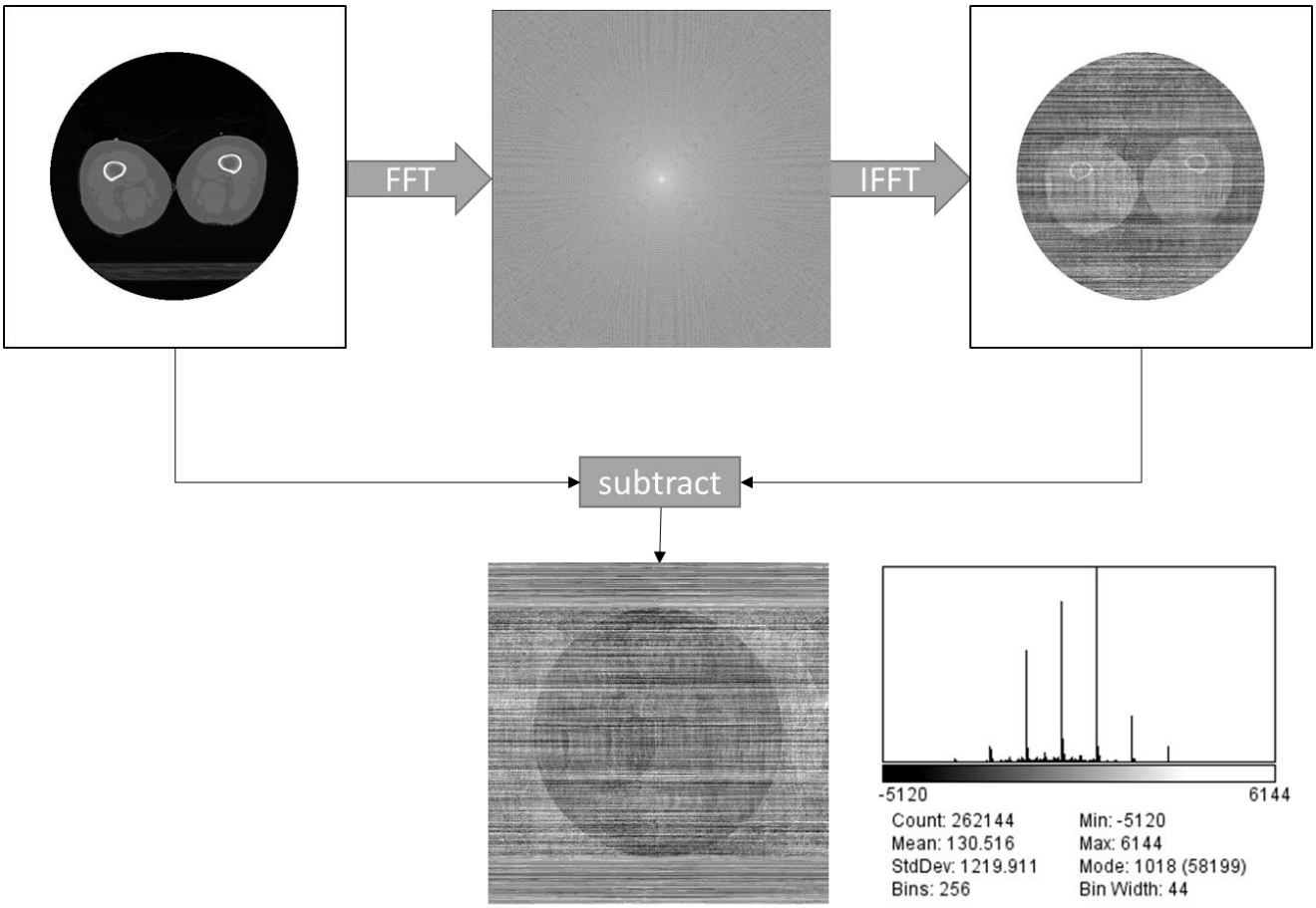
	Mean
A	4.30E-08
B	6.57E-08
C	-9.91E-07
D	NaN





Ignoring region D, which represents the area outside of the so-called circular field of view, the relative errors are around E-07 (or 0.00001%). For this particular image it appears that precision limitations aren't a big deal.

But what happens if the image contained values that really pushed these limitations? One way to do this is to introduce very large numbers into the image. So, running the above experiment again except with region D being filled with a large number (1E+10):



The recovered image becomes corrupted. The large values in region D dominate the precision offered by single precision and cause the rest of the image to get washed away.

A good point to make here is that this experiment has no real-world use other than to show that using FFTs will inherently introduce a certain amount of error. When used to accelerate convolution, there's some math done in the Fourier domain (between the FFT and IFFT) that will exacerbate the issue. But how much of a problem precision limitations create depends on the data being processed.

In regards to Richardson-Lucy, it's possible for the division operation (step two) to create very large values in regions where relatively small values exist (e.g. regions D and C). Per the above experiments, it's possible that said large values can corrupt the signal being processed. Again, one way to address this is to perform the FFT using double precision. Instead I opted to simply clamp the output of step two to be no more than a hundred, and my experiments show that this clamping does not prevent the algorithm from practically functioning.

Negative Numbers:

The underlying physics for Richardson-Lucy prohibits negative values, since in theory you cannot count a negative number of photons. Indeed noise can cause negative values to show up in an image, however that's an art form beyond the scope of this article. I chose to simply clamp the input image to zero and rely on other clamp operations (see below) to enforce positivity during processing.

Divisions by zero:

Nothing sophisticated about this one. The good news is the non-negativity constraint makes the division by zero pretty easy to handle. Clamping the input denominator to the smallest number a 32-bit floating-point value can hold accomplishes both in one step.

Convolutions of two even-sized matrices:

Using an even-sized PSF (e.g. 2x2, 6x6, 4x5) with an even sized image (e.g. 512x512) can cause Richardson-Lucy to “destroy” an image. This stems from how convolutions of two even sized signals are traditionally implemented, and how this implementation creates an inconsistency in Richardson-Lucy’s theory. In lieu of a detailed explanation, it ultimately comes down to how the first and last elements of the output convolution are handled. One of the two must be convolved with an implicit padded zero, and which one gets this padded zero isn't consistent if the signals are flipped. Remember the part where we need to convolve against a flipped PSF? Yup, that’s where things blow-up.

To illustrate, consider this experiment:

For a = [1 2 3 4] and b = [.3 .7]:
octave> conv(a,b,'same') = 1.3000 2.3000 3.3000 2.8000
octave> fliplr(conv(fliplr(a),fliplr(b),'same')) = 0.30000 1.30000 2.30000 3.30000

Superficially it'd seem that flipping both input signals, convolving, and then flipping the output should give the same result as just convolving the two original, unaltered signals. This experiment shows that this is not true. A simple solution is to pad one of the signals with a zero:

For a = [1 2 3 4] and b = [.3 .7 0]:
octave> conv(a,b,'same') = 1.3000 2.3000 3.3000 2.8000
octave> fliplr(conv(fliplr(a),fliplr(b),'same')) = 1.3000 2.3000 3.3000 2.8000

And now both convolutions give the same result, which means that Richardson-Lucy will remain stable. Think of not padding the PSF as being equivalent to using an entirely different PSF in step three instead of using just a mirrored version of the original PSF. It's incorrect and breaks the algorithm.

Implementation:

```
af::timer loopTime = af::timer::start();

//Run iterations
for (int i = 0; i < iterations; i++){
   cout << "Iteration " << i << endl;

   //Forward problem
   forward = af::convolve2(est, psf, af::convMode::AF_CONV_DEFAULT, convType);

   //Clamp denominator to some lower bound
   denom = af::max(minVal, forward);

   //ratio
   ratio = measured / denom;

   //Clamp ratio to some upper bound
   ratio = af::min(ratioMax, ratio);

   //Compute update from ratio and psf
   update = af::convolve2(ratio, psfUD, af::convMode::AF_CONV_DEFAULT, convType);

   //Apply update
   est = update * est;

}

af::sync();
```

Results:

I'm not interested in a detailed image quality investigation, so I'll share the results of one experiment:

Image size: 512x512
PSF Size: 9x9 (Gaussian)
PSF Standard Deviation in X & Y: 5
Iterations: 1000
Platform: Intel OpenCL CPU
Command line (0 indiciates Intel OpenCL CPU device): ./rlaf singleImage.raw 512 0 9 5 1000

Image 1: Simulated Blurring

Image 2: Recovered Image after 1000 iterations

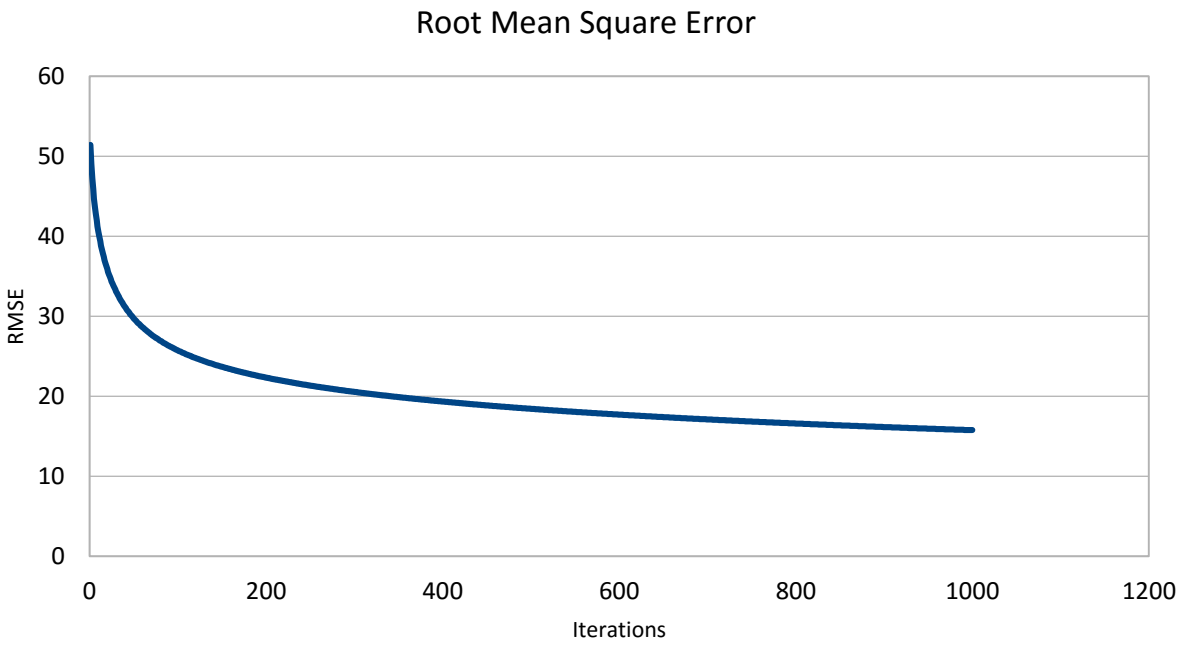


Figure 1: Plot of RMSE of recovered image vs original, unblurred image across iterations

Summary:

My first take away was how quickly I was able to prototype Richardson-Lucy in ArrayFire. It took no more than fifteen minutes to have a functioning implementation. Getting the ArrayFire library setup on Linux took a few hours, but that's mostly my own fault for not reading the instructions. My second time around it only took a few minutes.

I originally wanted to write a single article about algorithm implementation and performance optimization, but it turns out that performance optimization is complicated enough to warrant an article of its own. No joke, I've been looking at it for well over a month now. I don't want to wait any longer to post something, nor do I want a single post reading like a dissertation. My next article will go into analyzing performance quirks of this algorithm when run on the i7-5775c's CPU and its integrated graphics chip.