BinShrink: A multi-resolution filter with cache efficient averaging.

Release 1.00

Bradley C. Lowekamp[1] and David T. Chen[1]

November 18, 2013

[1]National Library Of Medicine

Abstract

We present a new filter for the Insight Toolkit (ITK) for reducing the resolution of an image by an integer factor while averaging called *BinShrink*. This filter provides a new level of performance to ITK for reducing resolution and noise present in an image. The filter supports streaming, multi-threading and most of ITK's pixel types including scalars, *Vector*s, *SymmetricSecondRankTensor*s, and *RGBPixel*s. The filter has been optimized to efficiently access the input image thereby greatly increasing performance over conventional methods.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/10380]

Contents

Our new *BinShrink* filter reduces the resolution of an input image by a integer shrinking factor, while performing averaging of an input neighborhood. It can be used in streaming processing to reduce the size of an image that is larger the main memory. The averaging can effectively reduce uncorrelated noise by reducing the expected standard deviation by a factor $1/\sqrt{n}$ where n is the number of samples in the neighborhood, based on the Bienaym formula.

Electron microscopy pushes the limits of resolution and detectable signal, sometimes having less than a 1:4 signal to noise ratio. The "binning" algorithm is commonly used in processing high resolution electron microscopy images. It is available in packages such as The Boulder Laboratory for 3-D Electron Microscopy of Cell's IMOD[2], and BSoft [1].

In this paper we demonstrate the effective performance improvements which can be achieved by changing the way an algorithm accesses the data. We show that by accessing the data is a coherent scan-line order a 10 times speedup can be realized in some cases.

1 Implementation

The algorithm implemented in the *BinShrink* filter can be described with (Equation 1) for the 2D case.

$$\mathsf{I}_{out}(x_o, x_1) = \frac{\sum_{i=0}^{f_0} \sum_{j=0}^{f_1} \mathsf{I}_{in}(f_0 x_o + i, f_1 x_1 + j)}{f_0 f_1} \tag{1}$$

The output image is only defined when all the required input image pixels are defined. The vector variable $\bar{\mathbf{f}}$ is a user specified shrink factor.

The method operates only on local input and output regions making the filter appropriate for streaming. Also the only operations required are pixel wise addition and division by a scalar, so the filter can readily work with a wide variety of pixel types.

1.1 Geometry

The geometry of an image includes its pixel spacing, origin, and image orientation. For the *BinShrink* filter the orientation is unchanged from the input image to the output image, but the output image's spacing is scaled from the input spacing by the shrink factor. Also, there is some complication when defining the origin. When the input image size is not evenly divisible by the shrinking factor, a choice of how to "round" the pixel locations needs to be made. We have chosen to maintain the physical location for the image signal and truncate the odd pixels at the boundaries. As an ITK image can have non-zero starting indexes, we define that the output origin pixel must lay on the same lower boundary as the input, and therefore the origin must be adjusted accordingly (See figure 1).

While this approach maintains the physical location of the image signal, it does not always maintain either the full extent or the center of the image. Therefore for algorithms such an image pyramids or certain neuro-imaging analysis, this method may not be appropriate, or assurances must be made that the input size is divisible by the shrink factor.

1.2 Initial Implementation

We have included the initial, naive implementation of the filter in the submission for performance comparison purposes and have named it *BinShrink2*. The above description of geometry applies only to the *BinShrink* filter. The original *BinShrink2* was derived from ITK's *Shrink* filter, which has different geometry characteristics than described above. The original *BinShrink2* follows the same geometry as the *Shrink* filter.

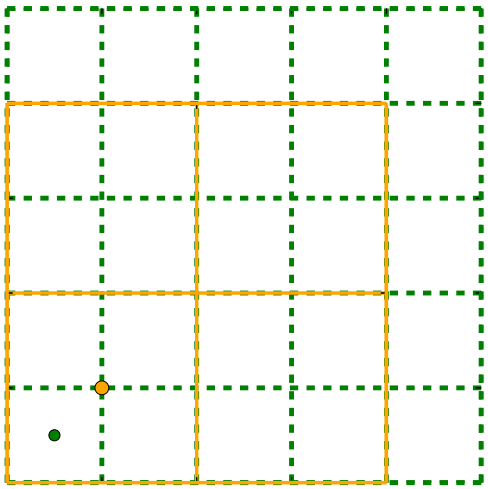

Figure 1: The change in image geometry from a 5x5 image binned by a factor of 2x2. The green dotted lines are the input image. The yellow grid is the result of the filter. The points represent the respective origins.

This original implementation used ITK's conventional *NeighborhoodIterator* to average the input for each output pixel (See Listing 1).

```
while ( !outIt.IsAtEnd() )
  {
  outputIndex = outIt.GetIndex();

  inputIndex = outputIndex * factorSize + offsetIndex;

  inputIt.SetLocation( inputIndex );

  AccumulatePixelType sum = NumericTraits<AccumulatePixelType>::Zero;
  typename ConstNeighborhoodIteratorType::ConstIterator ci = inputIt.Begin();

  for (ci.GoToBegin(); !ci.IsAtEnd(); ++ci)
    {
    sum += AccumulatePixelType( ci.Get() );
    }
  sum = sum / double( inputIt.GetActiveIndexListSize() );

  outIt.Set( sum );
  ++outIt;
  }
```

Listing 1: A selected section of code from *BinShrink2* filter using the neighborhood iterator.

1.3 Optimization

The implementation we suggest for inclusion in ITK is the *BinShrink*, which follows the geometry described above (Section 1.1) and is not derived from the *ShrinkFilter*. The initial implementation accesses the memory in an incoherent fashion based on the input neighborhood. However, it is faster to access memory in a linear and coherent fashion. Therefore we designed *BinShrink* to access the input image on a per scan-line basis and to operate the averaging on whole scan-lines, not individual pixels. By utilizing ITK's *ScanlineIt-*

erators, we improved the algorithm to work on a scan-line for the innermost loops (See Listing 1).

```
while ( !outputIterator.IsAtEnd() )
  {
  const OutputIndexType outputIndex = outputIterator.GetIndex();

  typename std::vector<OutputOffsetType>::const\_iterator offset = offsets.begin();
  const InputIndexType startInputIndex = outputIndex * factorSize;

  while ( ++offset != offsets.end() )
    {
    inputIterator.SetIndex( startInputIndex+*offset );

    for( size\_t i = 0; i < ln; ++i )
      {
      for( size\_t j = 0; j < factorSize[0]; ++j)
        {
        accBuffer[i] += inputIterator.Get();
        ++inputIterator;
        }
      }
    }

    for ( size\_t j = 0; j <ln;++j)
    {
      accBuffer[j] = accBuffer[j] * inumSamples;
      outputIterator.Set( static\_cast<OutputPixelType>(accBuffer[j]) );
      ++outputIterator;
    }

    outputIterator.NextLine();
  }
```

Listing 2: A selection of code from the *BinShrink* filter demonstrating scan-line averaging.

2 Results

To test our BinShrink filter we have used a synthetic function created by Marschner and Lobb[3], along with additive noise to create test images. This function is often used to evaluate volume rendering reconstruction filters. We have rendered it into a 128 pixel cubic volume such that the majority of the frequencies in the function are sampled just above 4 times the Nyquist frequency. Such an image makes for a challenging theoretical data set when the shrinking factor is also 4.

The code used to generate these images was written in Python with SimpleITK. The original Marshner-Lobb volume was normalized with the *Normalize* filter so that the volume had a mean of 0 and a standard deviation of 1. Then Gaussian distributed random noise was added with a 0 mean and a sigma to achieve the targeted signal to noise ratio. After the shrinking operation was performed, the volume was again normalized for contrast. Then the center slice was extracted and tiled. Finally it was colorized by the *ScalarToRGBColormap* filter[4].

We examined the results of the *BinShrink* filter by varying the signal to noise ratio in our Marshnel-Lobb test image and then shrinking those noisy images by factors of 2 and 4 (See figure 2). For comparison we also used the *SmoothingRecusiveGaissian* filter in conjunction with the *Shrink* filter on the same test noisy images. We set *sigma* of the smoothing Gaussian kernel at 0.7 times the shrink factor.

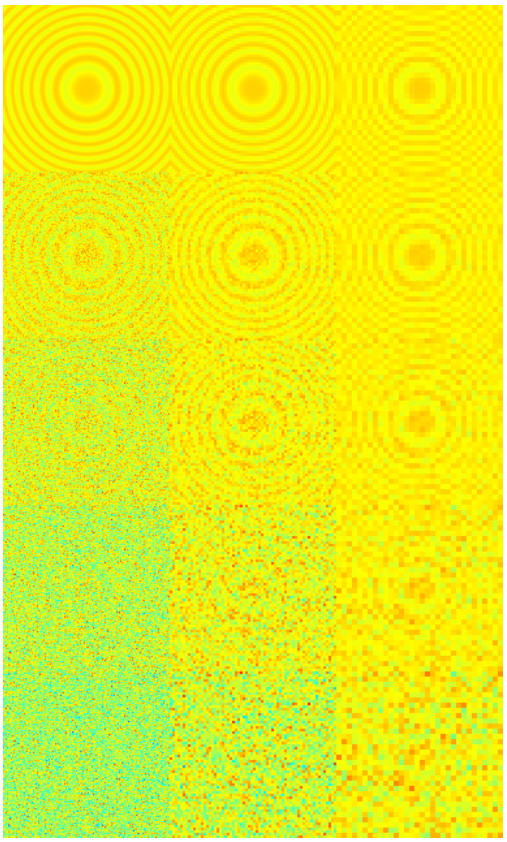

Figure 2: Outputs of the *BinShrink* filter on a Marchner-Lobb function with additive Gaussian noise. The (Left) column is the original image, the (Center) column is binned by 2, and the (Right) column is binned by 4. The signal to noise ratio varies by row with the values 100:1, 2:1, 1:1, 1:2, and 1:4, respectively.

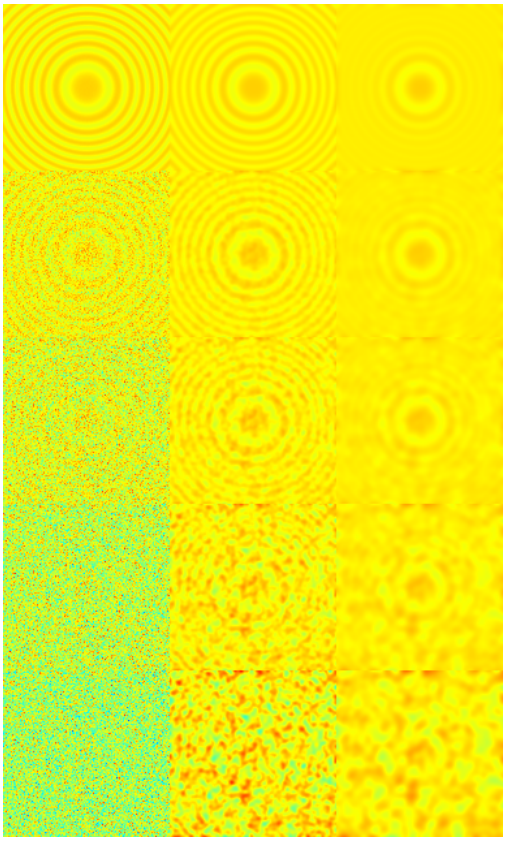

Figure 3: Outputs of the *SmoothingRecusiveGaussian* and *Shrink* filters on a Marchner-Lobb function with additive Gaussian noise. The (Left) column is the original image, the (Center) column is smooth with a sigma of 1.4 and shrunk by 2, and the (Right) column is smoothed by 2.8 and shrunk 4. The signal to noise ratio varies by row with the values 100:1, 2:1, 1:1, 1:2, and 1:4, respectively.

Algorithm	Shrink Factor						
	2	3	4	6	8	12	24
BinShrink	0.0149	0.0116	0.0118	0.0113	0.0112	0.0113	0.0110
BinShrink2	0.0862	0.0465	0.0714	0.0797	0.0940	0.1217	0.7451
GaussianShrink	0.4850	0.4779	0.4778	0.4803	0.4791	0.4752	0.4767
MeanShrink	0.7440	3.9093	3.9264	12.818	31.602	121.80	1412.3

Table 1: The execution time in seconds of algorithms verses various shrink factors.

Comparing the results of Gaussian filtering (Fig. 3) with *BinShrink*'s box filtering (Fig. 2) reveals the aliasing that the latter can produce. This artifact is most apparent in the images with signal to noise ratios of 100:1, the images in the upper right-hand corner of the two figures. The Gaussian filter image is much smoother without the banding artifact.

2.1 Performance

To analyze the performance of our two bin shrinking methods (both the optimized and naive versions) we compare them against similar processes which can be performed in ITK with other pairs of filters.

Running a *Mean* filter followed by a *Shrink* filter is a close approximation to *BinShrink*. The *Mean* filter computes an average for each input pixel's neighborhood in a brute force fashion. This approach wastes computation on input pixels that are not used by the *Shrink* filter.

Using a Gaussian kernel to reduce aliasing is another alternative to the box kernel implicitly used with the *BinShrink* filter. Gaussian filtering can be performed in constant time and independent of the size of the Gaussian with the *SmoothingRecusiveGaussian* filter.

We generated a 384 pixel cubic image of Gaussian distributed noise for performance evaluation. We set the number of threads used to be 16. Utilizing SimpleITK and Python's *timeit* module, we report the median of 3 runs for each algorithm across varying shrink factors (See Figure 4 and Table 1). The image size, the number of threads, and the shrink factors were carefully chosen such that the output image was always evenly divided for multi-threading. As expected the *Mean* approach suffers from exponential cost as a function of shrink size, while the *SmoothingRecursiveGaussian* method remains constant. The *BinShrink2* implementation only touches each input pixel once, but it also suffers from exponential growth likely due to its memory access pattern being inefficient and not cache coherent. On the other hand the *BinShrink* implementation execution time decreases as the shrink factor increases.

Analyzing the difference in performance between the *BinShrink* and *BinShrink2* filters is quite interesting. Both these filters access the same number of input pixels and output pixels to perform the same computation. The difference between them is the type of iterator used and the order in which the images are accessed. The differences in the iterator should account for the time difference for a shrink factor of 2. However, the large increase in the *BinShrink2* execution times can not be explained by the difference in the iterator operation costs. Specifically our performance results indicate that for a shrink factor of 24 the *BinShrink* is 67X faster. We conjecture that this disparity is due to *BinShrink2*'s inefficient memory access pattern causing decreased cache hits.

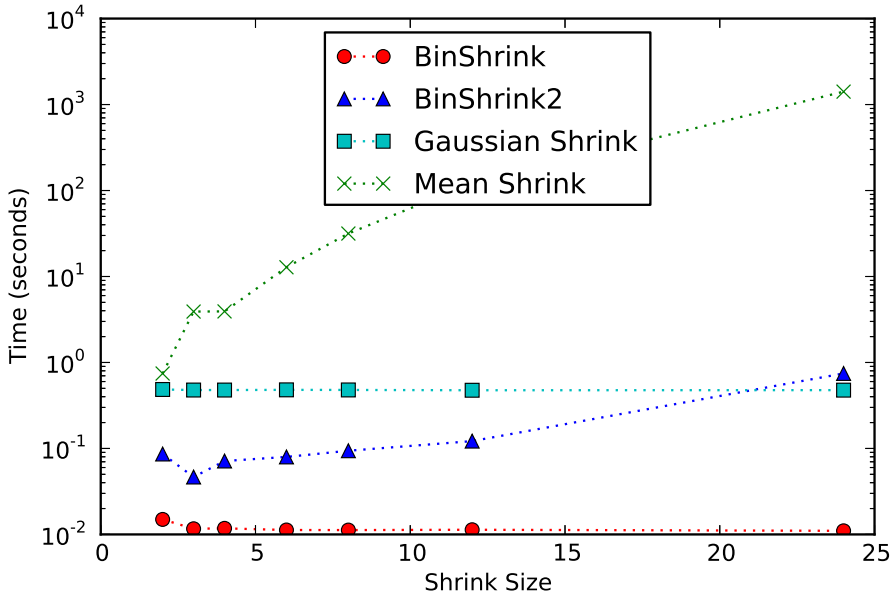

Figure 4: The execution time of the *BinShrink* filter compared to the original *BinShrink2* implementation and other shrinking approaches.

3 Conclusion

We have demonstrated that the *BinShrink* filter is a fast filter for multi-resolutional work. We have also shown that by coherently accessing images in scan-line order the performance can be improved by a factor of 10. *BinShrink* may not always be the best method for image quality as it may result in aliasing. However, features such as wide pixel type support and streaming make it quite practical for working with large data sets.

References

[1] Heymann J.B. and Belnap D.M. Bsoft: Image processing and molecular modeling for electron microscopy. *J. Struct. Biol.*, 157(1):3–18, 2007. (document)

[2] Kremer J.R., Mastronarde D.N., and McIntosh J.R. Computer visualization of three-dimensional image data using IMOD. *J. Struct. Biol.*, 116:71–76, 1996. (document)

[3] Stephen R. Marschner and Richard Lobb. An evaluation of reconstruction filters for volume rendering. In *IEEE Visualization*, pages 100–107, 1994. 2

[4] N. Tustison, H. Zhang, G. Lehmann, P. Yushkevich, and J. Gee. Meeting Andy Warhol somewhere over the rainbow: RGB colormapping and ITK. *The Insight Journal*, 01 2009. 2