Constant Time Gaussian Blur

Lin Nan
calin@nvidia.com

February 2011

Document Change History

Version	Date	Responsible	Reason for Change
1	Jan 2011	Lin Nan	Initial release

Abstract

This sample implements the 2D Gaussian filtering using repeated integration. The term "constant time" refers to the invariant execution time for any filter width, which is the main advantage of this method. Compute shaders and group shared memory are used to perform per row and per column scan operations in the sample.

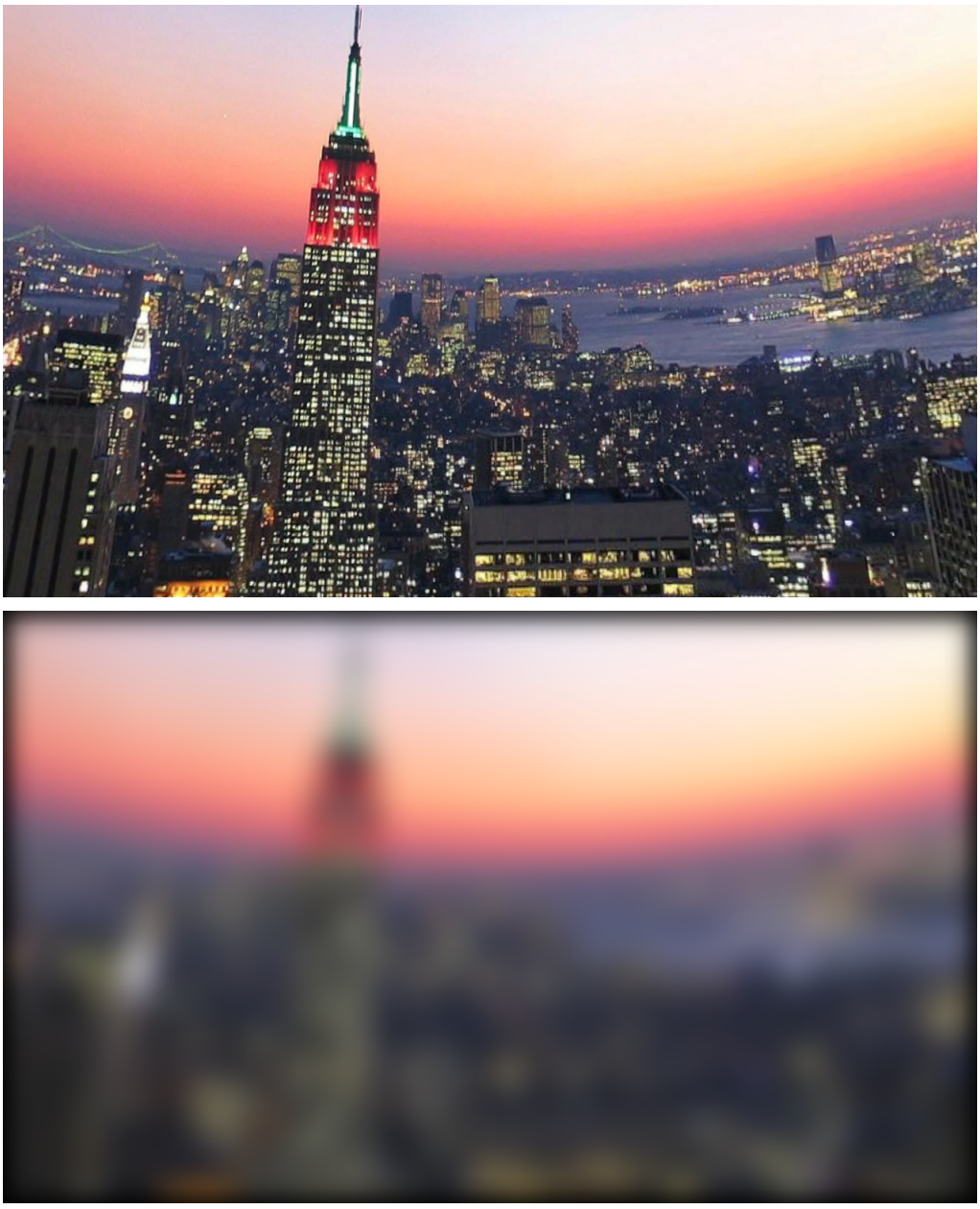

Figure 1. The input image (top) and filtered result (bottom) with filter width = 120

Introduction

Two-pass direct convolution is most commonly used Gaussian filtering method in today's real-time graphics, but which becomes time consuming when operating on large kernels due to that the per-pixel cost increases linearly with the kernel size. Many applications down-sample the input image before the filtering to trade off quality for performance. To address this issue, several filter width independent algorithms were proposed in the past, most of which involved with frequency domain transform [1], recursive computing [4] or repeated integration [5] that cannot be conveniently implemented on traditional GPU pipeline until the recently introduced GPU general computing. Among these three techniques, repeated integration is usually considered intuitive and effective.

In this code sample we demonstrate how to perform Gaussian filtering by repeated integration using compute shader. The algorithm is based on the following simple fact: repeatedly applying a box filter (average filter) on the input data will quickly approximate the result of Gaussian filtering, while the box filtering can be handled by scan operation, thus makes the per-pixel cost independent to the kernel size. As the traditional two-pass convolution method, we also employ a vertical pass and a horizontal pass, in each of which a column or a row of pixels is fetched into shared memory first, then applied with a box filter repeatedly and written to the global memory at last.

In the rest of the document, we will explain the details of the algorithm, then describe the implementation using compute shader and group shared memory.

Algorithm

Gaussian by Repeated Integration

The shape of Gaussian function can be approximated by repeating a box integration, i.e. average, over a delta function for multiple times. Figure 2 shows the equivalent kernels of repeating a set of box filters for 1~5 iterations in 1D situation. The filter curve converges quickly to a Gaussian function after the 4th iteration. Actually, Heckbert has shown that any piecewise polynomial kernel of degree $n - 1$ can be computed by integrating the signal n times and point sampling it several times for each output sample [5], but here in this document we will only concern the case of Gaussian filtering.

Given the standard deviation σ, the normalized Gaussian kernel is

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

To approximate G, we use Kovesi's method [6] to pick a box filter of width w and apply it for n times. The box width w is determined by

$$w = \sqrt{\frac{12\sigma^2}{n} + 1}$$

The value of w usually turns out to be a real quantity. We linearly interpolate between two pixels at the filter window boundaries for the fracture part of w, which gives better accuracy than Kovesi's original approach [6].

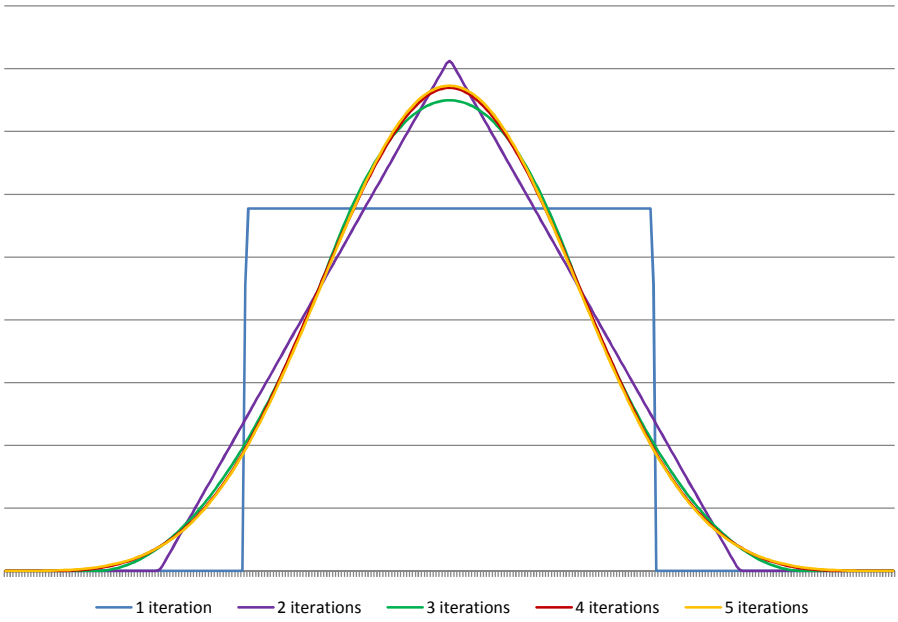

Figure 2. The equivalent kernels of repeated box filter in 1D

The technique of repeated integration can be generalized to 2D straightforwardly, but it is worth to note that the lower iteration count will introduce undesired anisotropy in higher dimensions, as Young et al. pointed out [8]. Figure 3 visualizes the equivalent 2D kernels of repeating the box filter for 2, 3, 5 and 12 times. Within the cutoff range 3σ (marked by thin blue circle), 5 iterations give close enough result to the isotropic Gaussian kernel. In real applications 4 iterations can be considered good enough because the weights on outskirt of the kernel have very limited contribution to the final value.

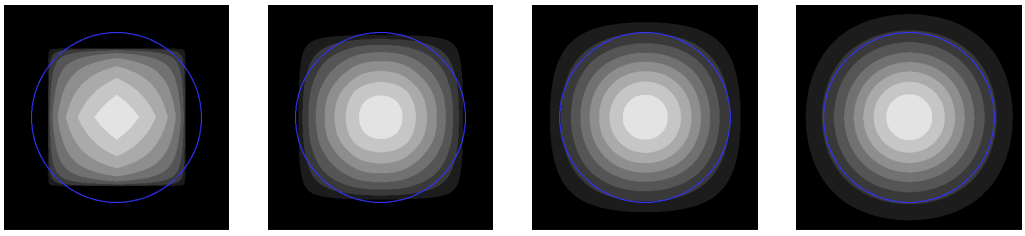

Figure 3. The equivalent kernels of repeated box filter in 2D, visualized by logarithmic contour regions. The thin blue circle marks 3σ boundary.

Box Filtering by Prefix Sum

With the filter width and repeating count setup, the next step is to find an efficient way to executing the box filtering multiple times. It is well known that box filtering can be easily

computed from summed area table [3], which is the 2D equivalence of "inclusive scan" or "inclusive prefix sum" in parallel computing. For algorithm details of scan operation, please refer to [1, 7]. Here we only give a brief description for the usage in this code sample:

Given an input data sequence $a = \{a_0, a_1, a_2, \ldots, a_n\}$, an inclusive scan produces an output sequence $b = \{b_0, b_1, b_2, \ldots, b_n\}$, where $b_i = a_0 + a_1 + a_2 + \ldots + a_i$. (The math operator is not necessarily to be addition though.) Then the sequence b can be conveniently used to compute the average value of any local region of a, as $(a_i + a_{i+1} + a_{i+2} + \ldots + a_{i+N-1}) / N = (b_{i+N-1} - b_{i-1}) / N$, therefore the N samples in a are reduced to two samples in b.

When generalized into 2D, the scan operation on an input image produces a summed area table S, in which a point at position (u, v) takes the value of cumulative sums within the rectangle region $[0, u; 0, v]$. For any rectangular region $[u, u + w; v, v + h]$ in the image, the average value can be calculated using $(S_{u+w,v+h} - S_{u+w,v} - S_{u,v+h} + S_{u,v}) / (w * h)$. Thus S has the ability to perform arbitrary sized box filtering with the fixed cost of four samples per pixel.

Although intriguing on paper, such 2D filtering scheme faces a couple of drawbacks in practice: the precision of floating point number becomes insufficient at higher resolution, and the additional memory bandwidth required for writing back the temporary result after each iteration is considerably high. The former raises noise artifacts which is extremely hard to eliminate despite of many past efforts to improve the precision, while the latter turns into the main bottleneck of performance. We thereby do not employ 2D SAT for the work, but perform two passes of 1D prefix sum instead, first on columns and then on rows, so that the accumulated values will not run out of precision and the per column/row data can be stored in group shared memory to reduce off-chip traffic. We will address the implementation details in the next section.

Implementation Details

The sample renders a cubemap as the skybox, and then applies the constant time Gaussian blur using compute shaders.

The D3D resources used for filtering:

- A DXGI_FORMAT_R16G16B16A16_FLOAT texture as the main render target and the input image for filtering.
- An RWTexture object mapped to a UAV (unordered access view) as the output buffer of compute shaders. Due to the constraint that unordered access must output to a single 32-bit channel format, we use DXGI_FORMAT_R32_UINT to pack color data or DXGI_FORMAT_R32_FLOAT for grayscale data.

We setup the computer shaders to handle one column (row) of pixels per thread group. The optimal number of threads per group is either 128 or 256 according to the test results. All pixel data from the same column (row) are stored in the group shared memory until the processing finished. This actually sets a resolution cap for the algorithm depending on the size of shared memory. D3D11 requires at least 32KB per thread group, which means the maximum input resolution is 8192x8192 for grayscale image, or 2730x2730 for RGB color image. The first compute shader pass is column filtering, in which the shader executes following steps (assuming 128 threads per group):

1. Fetch in the pixel values from the corresponding column in the input image with one pixel per thread. That way the group reads 128 pixels a time. Loop until the entire column being fetched.
2. Start the first prefix sum operation which is done on the newly fetched data, and then hold the results in the register space. After calling a barrier function to ensure all elements have been processed, the results are sent back to shared memory and overwrite the old data.
3. Loops several times to approximate the Gaussian filtering. Each iteration applies an averaging operation and then a prefix sum as described in the previous section. Same as step 2, we need to synchronize all threads in the group before overwrite the old data.
4. Output the filtering results to global memory (the RWTexture object mapped to the UAV). Since the data in shared memory are still in prefix summed form, we have to perform one more averaging. For RGB color data, this step also packs the three channels into R11B11G10 format due to the INT32 output format.

The next pass is row filtering, which is almost identical to the column filtering except that it uses the intermediate UAV as input. Such two-pass scheme, as mentioned in the previous section, utilizes the separation property of Gaussian function, avoids precision issue and fits the data into shared memory.

Figure 4 shows the filtering results after 1~5 iterations. Notice that there are almost no perceptible differences after the 4th iteration. In an extreme case, an HDR input image containing a single bright spot against a dark region, the anisotropy artifact may be spotted if the iteration number is less than 5.

Running the Sample

- Use left mouse button for panning the camera.
- Slide "Filter Radius", which is defined as 3σ of the normalized Gaussian function, to change the filter width.
- Use "Iteration" slide bar to set iteration times.

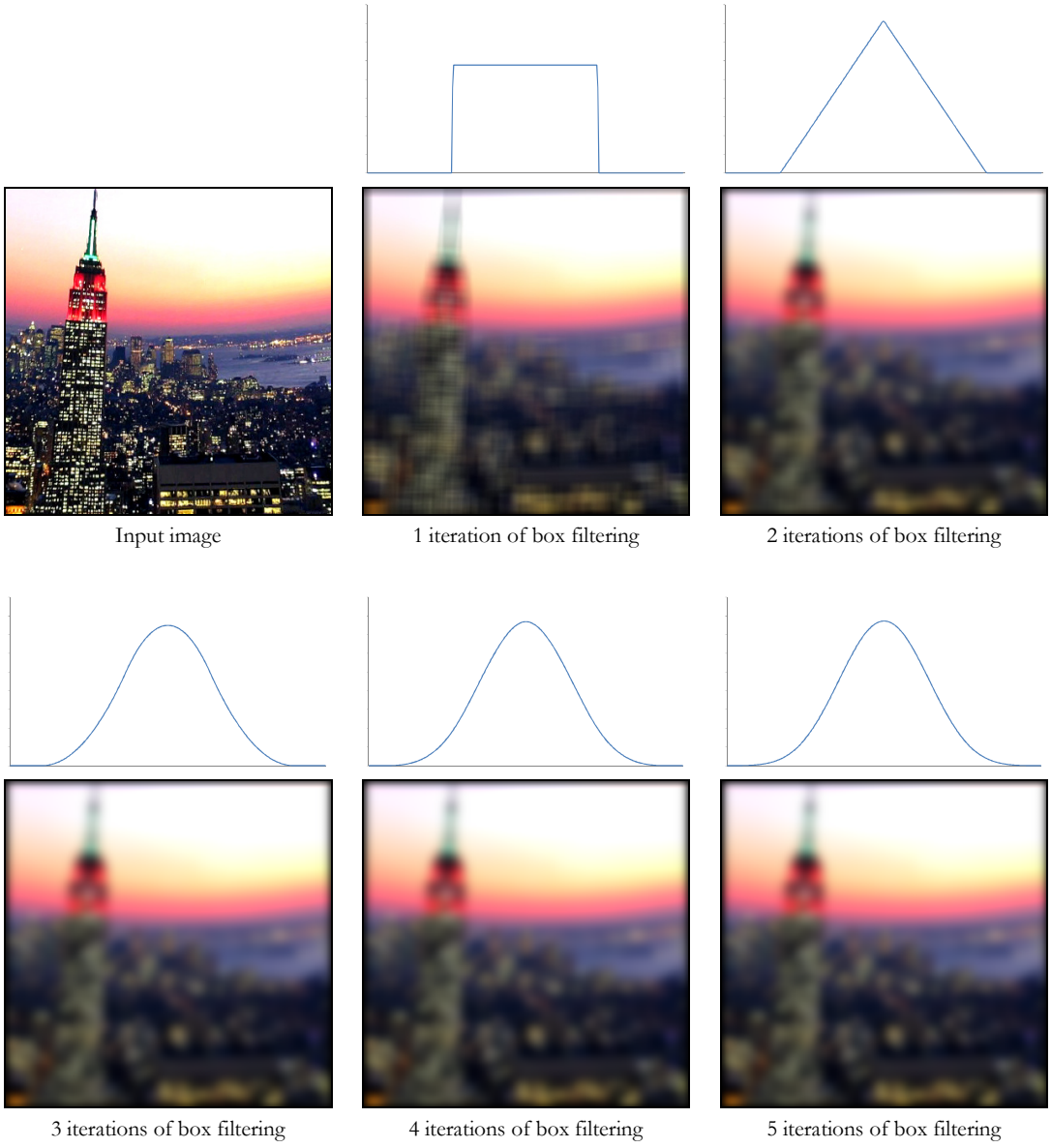

Figure 4. Filtering result after 1~5 iterations and their equivalent 1D filters

References

1. Blelloch, G.E., 1989, "**Scans as Primitive Parallel Operations**", IEEE Transactions on Computers": C-38(11):1526–1538.
2. Brigham, E. Oran, "**The Fast Fourier Transform**", Prentice-Hall, Englewood Cliffs, NJ, 1974.

3. Crow, F., 1984, "**Summed-Area Tables for Texture Mapping**", SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pp. 207–212.
4. Deriche, R., 1993, "**Recursively Implementing the Gaussian and Its Derivatives**", Proceedings of the 2nd International Conference on Image Processing, Singapore, pp. 263–267.
5. Heckbert, P.S., 1986, "**Filtering By Repeated Integration**", SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques.
6. Kovesi, P., 2010, "**Fast Almost-Gaussian Filtering**", 2010 International Conference on Digital Image Computing: Techniques and Applications, pp.121-125.
7. Sengupta S., Harris M., and Garland M, 2008, "**Efficient Parallel Scan Algorithms for GPUs**", NVIDIA Technical Report NVR-2008-003.
8. Young, I.T., and Lucas J. van Vliet, 1995, "**Recursive Implementation of the Gaussian Filter**", Signal Processing, Volume 44 Issue 2, June 1995

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com