Bilateral Filtering via Compute Shaders

Varun Malladi
Dartmouth College
Bachelor
varunmalladi.github.io

Abstract

The bilateral grid is a data structure intended to accelerate operations such as bilateral filtering. The major benefit of the approach is its parallelizability. Previous implementations leveraged this by implementing grid creation and operations on the traditional rasterization pipeline, via vertex and fragment shaders. However, this is wasteful by the nature of the rasterization pipeline being optimized for real-time rendering. Most modern GPUs now support compute shaders, which allow general-purpose code to be parallelized on the GPU. We utilize this to implement bilateral grid construction and various operations via compute shaders.

1. Introduction

Gaussian filtering is very effective at reducing noise globally. However, it does not recognize edges, so applying Gaussian blur to an image globally will result in a loss of detail at the edges. This is typically not desired. Bilateral filtering addresses this by modifying the Gaussian kernel locally to account for edges. However, doing so greatly increases the computational complexity of the blurring procedure, as the filtering kernel is now dependent on the values of the pixels it is being applied to, rather than being globally constant.

As Chen *et al*. [1] demonstrated, one can approach this problem by reconsidering the data structure we are convolving against. The issue with the bilateral filtering approach described above was the the introduction of edge-awareness changed the kernel locally. The idea of the bilateral grid is to represent the image as a 3 dimensional object, whose depth-hyperplanes correspond pixel intensity. In this object, pixels spanning an edge will theoretically be located on distant hyperplanes. As a result, convolving this structure with an ordinary 3D Gaussian kernel automatically accounts for edge-awareness.

While it may seem that we have merely relegated the computational complexity to the pre-processing stage, the advantage of this approach is its amenability to GPU implementation. Grid construction and image reconstruction is highly parallelizable, and as a result we can apply bilateral filtering in real-time.

2. The bilateral grid

In what follows, let I be the intensity of our input image. We assume that all intensity values are normalized to fall within $[0, 1]$. We will write $I(x, y)$ to refer to the value of the pixel at (x, y) in Cartesian coordinates. The process takes two parameters, s_s and s_t. The parameter s_s corresponds to spatial sampling, and is ≥ 1. The parameter s_t corresponds to range sampling, and directly effects the depth (or the number of layers) in our grid. Typically $s_t < 1$.

2.1. Grid creation

Let Γ be the bilateral grid. The dimensions of Γ are

$$\left(\left[\frac{I.w-1}{s_s}\right], \left[\frac{I.h-1}{s_t}\right], \left[\frac{1}{s_t}\right]\right), \tag{1}$$

where $I.w, I.h$ are the pixel width and height of the image, respectively, and $[-]$ denotes the nearest integer operation.

First we initialize every voxel in the grid to have value 0. Then, for each x, y in the input image, we do

$$\Gamma([x/s_s], [y/s_s], [I(x, y)/s_t]) \mathrel{+}= (I(x, y), 1). \tag{2}$$

Depending on what we may store, we may alteral the first component of the value the grid voxel is being incremented by. For example, we may store the RGB value of the image at (x, y) rather than the intensity.

2.2. Bilateral filtering

To do bilateral filtering we do bilateral filtering on our grid. Assuming grid creation as above, this corresponds to convolving the grid with a 3D Gaussian kernel. The kernel takes two components: a spatial and range Gaussian. We suggest taking the spatial σ_s to be s_s, and the range σ_t to

be s_t. Then produce two 1D Gaussian kernels k_s, k_t corresponding to σ_s, σ_t respectively. The convolution can then be implemented as first applying k_s along the x-dimension, then applying k_s along the y-dimension, and finally applying k_t along the depth dimension. We are able to do this because the Gaussian kernel is separable.

2.3. Slicing

Let O denote the output image, and $O(x, y)$ the (x, y) pixel value. The pixel value should correspond to the one being stored in the bilateral grid (e.g. image intensity, RGB). Then we compute the output image such that

$$O(x, y) = \Gamma(x/s_s, y/s_s, I(x, y)/s_t). \quad (3)$$

Note that we do not round the coordinates of the grid here; instead, we utilize trilinear interpolation on the grid to obtain the value at the specified location.

3. Implementation

Our implementation targets macOS platforms with support for the Metal API. We used a 14-inch, 2021 MacBook Pro with 16 GB of memory and an Apple M1 Pro processor. You can find our implementation at `https://github.com/treemcgee42/bilateral-grid`.

3.1. Data representation

The bilateral grid is represented as an array of 2D textures. The length of the array corresponds to the number of possible depth levels. We explored the possibility of representing the grid in a unified 3D texture, but would caution other implementations to note the assumptions the graphics API makes on such textures. At the time of writing, the Metal API regards depth in the 3D texture as mipmap levels. Hence sampling the 3D texture does not result in the desired trilinear interpolation.

3.2. Grid creation

To zero-initialize the grid, we wrote a compute kernel to write a zero-vector to each texel in the grid. The virtual grid of GPU threads corresponded to the dimensions of the grid.

This was immediately followed by another compute kernel responsible for writing to the grid. The grid of GPU threads corresponded to the dimensions of the reference image (texture). Each thread performs a read to the reference image, a read to the grid texture, and a write to the grid texture.

It is very important to consider synchronization issues with this scheme. Namely, if $s_s > 1$ then it may be possible for two threads to be accessing a particular texel in the grid simultaneously. The Metal API does not support atomic read and write for textures, so alternative synchronization methods needed to be used. Even if it did, this would not be ideal as the entire texture would have to be have to be passed between threads synchronously. But there only a portion of pixels in the input image could possibly affect a given voxel in the grid.

Our approach was to construct the grid in two passes. In the first pass, we construct the grid with no downsampling in the xy-dimensions. This ensures that each voxel in the grid can be incremented by at most one pixel. In the second pass, we downsample the grid to our desired dimensions. It may be possible to implement this logic in a single pass, though by utilizing two passes we can effectively bypass branching in our shader code.

3.3. Bilateral filtering

As discussed in [1], bilateral filtering on the bilateral grid corresponds to Gaussian filering on the grid with a 3D kernel. To implement this we first construct another bilateral grid texture, mocking the properties of the original grid. We then pass both the original and new grid, along with precomputed Gaussian kernels, to a compute shader which implements the convolution. Since the kernel is separable, we can 3D filtering one dimension at a time. In particular, we first convolve the grid horizontally against k_s, storing the result in the new grid. We then convolve the new grid with k_s vertically, storing the result back into the new grid. Finally, we convolve the new grid with k_t depth-wise, storing the results back into the new grid.

Unlike grid creation, there should not be any concerns with synchronization as only one thread will be reading from and writing to a given texel in the grids.

3.4. Slicing

To perform trilinear interpolation on our bilateral grid, which is represented as an array of 2D textures, we first compute the coordinates (x, y, z) at which we wish to obtain the interpolated value. We then fetch the two nearest 2D textures corresponding to the coordinate, *e.g.* the ones whose indices correspond to the floor and ceiling of z. Since each of these are ordinary 2D textures, we can take advantage of hardware/API-supported bilinear interpolation by sampling these textures at (x, y). Finally, we perform 1D interpolation on the two resulting values to obtain our final trilinearly-interpolated result.

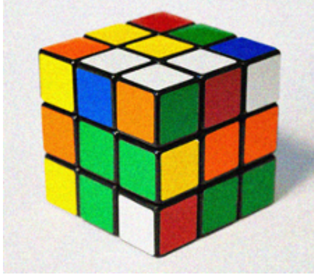
We did this with another compute kernel, with thread grid shape corresponding to the shape of the input (and desired output) dimensions.

4. Results

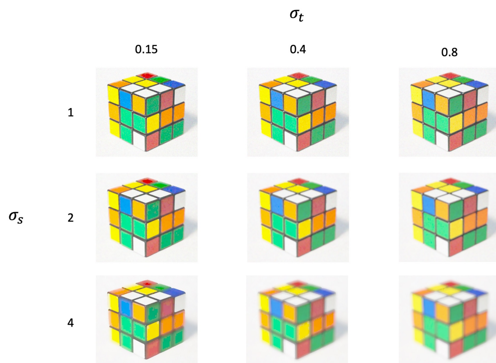
A major strength of the bilateral grid is its applicability to a wide range of applications. While we only cover two here, we refer the reader to [1] for further exploration.

4.1. Bilateral filtering

Given a test image



our results were as follows:



We used a kernel diameter of 5 in all dimensions. Compared to the corresponding call using the OpenCV library in Python, our method was approximetely twice as fast, at around 4 milliseconds. This number includes the creation of the grid, but does not include the conversion of the input image to a texture, or the conversion of the output texture to an image. Another factor to consider is that the OpenCV library likely does not support GPU-acceleration on our system, though it might support it on other hardware.

4.2. Cross-bilateral filtering

Cross-bilateral filtering is a simple extension of bilateral filtering, where the spatial and range Gaussians are applied to different images to generate a sort of combined image. Implementation-wise, this amounts to indexing the grid during creation with (intensity) values coming from the spatial image, but storing values coming from the range image in the grid. Slicing is also done by indexing into the grid using the spatial image. In this work, we applied the spatial Gaussian to a flash image, and the range Gaussian to a non-flash image of the same scene. Our results are as follows:

5. Conclusion

5.1. Limitations

The largest drawback of the bilateral-grid is its (texture) memory consumption. Not only does the grid occupy a considераl amount of memory, the operations on the grid may also require additional memory. For instance, the grid creation process utilized an intermediate grid texture and the downsampled it into a separate texture. The result of Gaussian filtering was a separate grid texture.

A consequence of the heavy memory requirements is that the bilateral grid will likely (as in our implementation) index depth with a single value. In the above, we described indexing depth using pixel intensity. As a result, techniques such as bilateral filtering on the grid are edge-aware with respect to pixel intensity, though in practice we may desire the algorithm to account for edges not apparent in the pixel-intensity domain.

5.2. Future research

To the extent of reducing memory usage, we are interested in exploring single-pass techniques for grid creation. We noted that while it would be possible to implement the logic we described above into a single pass, doing so would require branching code. We suspect, though haven't confirmed, that this would signficantly affect performance. Another solution we considered was leveraging atomic operations. For one, the Metal API does not support atomic operations on 2D textures. The larger issue, though, is that doing so may mean acquiring a lock for the entire thread. But given a voxel in the grid, there are only a relatively small amount of pixels in the input image which could affect the value of that voxel. Thus many threads would be unnecessarily waiting to acquire the lock.

We would also like to apply this method to videos. Since computation is dominated by grid creation, we are interested in finding ways to leverage similarities between frames to avoid reconstructing the entire grid every frame. Indeed, the strength of the bilateral grid comes from the flexibility of the grid structure.

Finally, we are interested in utilizing hardware support for 3D textures and trilinear interpolations. As discussed, this data structure is intended to be used for mipmaps. It would be interesting to consider whether the bilateral grid

could be represented (or approximated) in such a way. This may also reduce texture memory usage.

References

[1] Jiawen Chen, Sylvain Paris, and Frédo Durand. Real-time edge-aware image processing with the bilateral grid. *ACM SIGGRAPH 2007 papers*, 2007. 1, 2

[2] Frédo Durand and Julie Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002.

[3] James Tompkin. Bilateral filter lab.