HIGH-DIMENSIONAL GAUSSIAN FILTERING FOR COMPUTATIONAL PHOTOGRAPHY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Andrew B. Adams
July 2011

Abstract

Over the last decade, digital imaging has become ubiquitous. The advent of cheap digital cameras, and the inclusion of cameras in almost all mobile devices, has made photography one of the basic ways in which people record and communicate experiences.

The ubiquity of cameras has imposed new constraints on their physical form. Camera modules are expected to be thin, light, and cheap. These restrictions make the production of high-quality images challenging. We turn to increasingly sophisticated algorithmic tools to transform the raw data captured by a camera into a photograph.

This dissertation focuses on one such family of algorithmic tools: those expressible as a Gauss transform. One popular technique in this family is the bilateral filter, which smooths the fine detail in an image without crossing strong edges. It can be used to isolate and control the sharpness, tone, and contrast of a photograph at various scales. Its relatives, the joint-bilateral filter and the joint-bilateral upsample, allow for the fusion of data from multiple images. Another popular technique in the same family is non-local means, which denoises an image by replacing each pixel with the average color of all other pixels in the image with a similar local neighborhood.

A naive implementation of these algorithms is prohibitively slow. This dissertation unifies these algorithms under a common framework, describes a variety of applications of the transform in photographic image processing, and presents two new data structures to accelerate the computation of such transforms: the permutohedral lattice, and the Gaussian kd-tree.

The permutohedral lattice implements a Gauss transform as a resampling using the lattice A^*, which tessellates space with uniform simplices. For n d-dimensional

inputs, the permutohedral lattice performs a Gauss transform using $O(d^2 n)$ time and memory. The Gaussian kd-tree enacts a Gauss transform by clustering inputs at the leaves of a space-partitioning tree, and performing importance-sampled queries down the tree. It uses $O(dn \log n)$ time and $O(dn)$ memory. Previous methods for computing Gauss transforms are typically quadratic in n, or exponential in d.

Acknowledgements

Performing the research that goes into this dissertation would have been not nearly as enjoyable without the help of my many wonderful collaborators. In particular, the permutohedral lattice described in Chapter 3 is joint work with Jongmin Baek and Abe Davis. The Gaussian kd-tree described in Chapter 4 is joint work with Natasha Gelfand, Jennifer Dolson, and Marc Levoy.

My name is on this dissertation, but I am the product of all those who have inspired me, guided me, and nurtured me. This work therefore owes a great deal to many people, some of whom I will thank here.

Thanks to my lab mates Jongmin Baek, Abe Davis, Jen Dolson, Dave Jacobs, Sung-Hee Park, Eddy Talvala, and Zhengyun Zhang for their good friendship, good humor, and keen intelligence. It has been a pleasure to work with all of you. Thanks particularly to Eddy, who collaborated with me on everything not described in this dissertation, and shared an office with me for nearly seven years. I will always admire your terrifying ability to trawl the depths of the English language for puns.

Thanks to my mentor Marc Levoy for his enthusiasm and encouragement, and for the high standard he sets. Marc has driven me to be all that I can. Thanks also to my readers Leonidas Guibas, who is a repository of accumulated wisdom, and Mark Horowitz, who knows how to cut right to the heart of a problem. Thanks also to Vaibhav Vaish, Natasha Gelfand, Kari Pulli, Jeremy Sugerman, Björn Hartmann, Kayvon Fatahalian, Pat Hanrahan, and other graphics lab gurus who have shared their sage advice.

Thank you to my father Don, who taught me how to detect and discard nonsense, and who taught me that there is good, and how to look for it. Thank you to my

mother Helen, who instilled in me a love of learning and rigorous thought.

Thanks to our American friends Carol, Karey, Nikki, and Stuart, who have taken us in for the holidays and shared with us their incredible generosity and hospitality.

Thanks to the figure subjects that appear in this dissertation: Tony and Polly the labradors, the husky that helped pull our dog-sled, the bison who stopped to stare at us, the boxer dressed as a boxer, Ian, Malcolm, Hercules the kitten, and Annika.

Reaching further back, thanks to Richard Buckland and Ada Lim for demonstrating the value of enthusiasm, and for teaching me how to teach. Thanks to Dr Windon for showing me that clarity of expression is necessary for clarity of thought, and to Dr Bishop for his vote of confidence in me. He told me to be famous so he could "bask in reflected glory". Of course the old bastard died before I could achieve it.

Finally, thanks to my constant companion Elena, with whom life is a grand adventure.

Contents

List of Figures

Chapter 1

Gaussian Filtering and its Applications

A surprising number of photographic image processing operations can be expressed by a single equation:

$$\vec{\hat{v}}_i = \sum_j e^{-|\vec{p}_i - \vec{p}_j|^2/2} \vec{v}_j \tag{1.1}$$

This is a Gauss transform, in which output *values* $\vec{\hat{v}}_i$ are a weighted sum over input values $\vec{v}_j$, with the weights given by a Gaussian in the distance between two associated *positions* $\vec{p}_i$ and $\vec{p}_j$. Put simply, this equation mixes together *values* that have similar *positions*.

For image processing, our values will almost always be pixel colors, so that $\vec{v}_i$ represents the color of pixel i. We use a homogeneous representation for color, so that the weighted sum in Equation 1.1 performs a weighted average. This we describe as Gaussian *filtering*. Thus a pixel with red, green, and blue components r, g, and b, has a input value $[r, g, b, 1]$, and an output value of $[a, b, c, d]$ should be understood as the color $[\frac{a}{d}, \frac{b}{d}, \frac{c}{d}]$.

The positions associated with these values will vary according to the task at hand. For example, if we set the position vector of pixel i to be the (x, y) location of that

Gaussian Blur: $\vec{p}_i = \left(\frac{x_i}{\sigma} \ \frac{y_i}{\sigma}\right) \quad \vec{v}_i = \begin{pmatrix} r_i & g_i & b_i & 1 \end{pmatrix}$

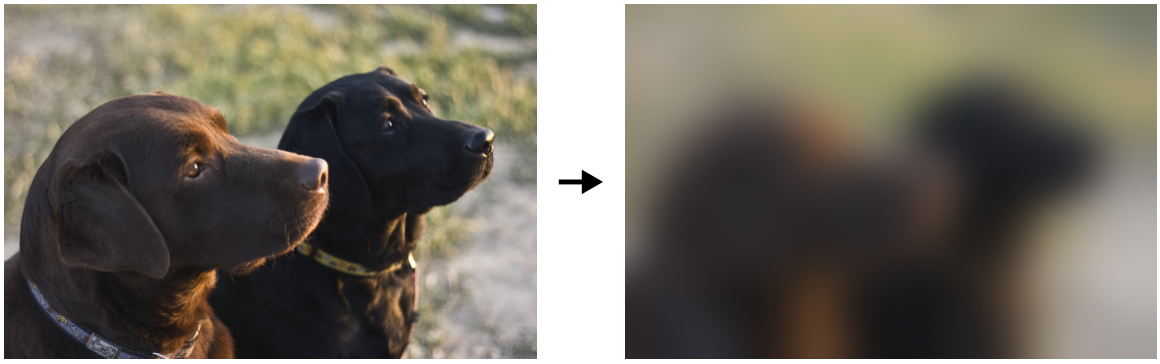

Figure 1.1: A Gauss transform mixes together *values* $\vec{v}_i$ that have similar *positions* $\vec{p}_i$. The simplest use of this in image processing is the Gaussian blur, in which the values are pixel colors, and the position of each pixel is its spatial location within the image. Mixing *values* that have similar *positions* thus mixes together pixel that are nearby in the image. We can set the standard deviation of the filter by scaling down the position vectors by some constant σ. We use *homogeneous* coordinates for pixel colors (in the projective geometry sense), which turns the Gauss transform's weighted sum into a weighted average.

pixel within the image, then Equation 1.1 expresses a Gaussian blur of standard deviation 1. If we wish to perform a larger blur we could modify the equation, but it is more convenient to instead scale down the position vectors (Figure 1.1).

1.1 The bilateral filter

While Gaussian blurs are common-place in image processing, they can already be efficiently implemented using a wide variety of methods. A more interesting example is the bilateral filter, which mixes together nearby pixels which also have similar colors, producing a piecewise flattening of the image (Figure 1.2). The two notions of spatially-nearby and similar-in-color can be combined into a single five-dimensional notion of similarity; pixels are mixed with other pixels that are nearby in *x*-*y*-*r*-*g*-*b*-space (this insight is due to Danny Barash [21]).

Bilateral Filter: $\vec{p}_i = \left(\frac{x_i}{\sigma_s} \; \frac{y_i}{\sigma_s} \; \frac{r_i}{\sigma_c} \; \frac{g_i}{\sigma_c} \; \frac{b_i}{\sigma_c}\right) \quad \vec{v}_i = \left(r_i \; g_i \; b_i \; 1\right)$

Figure 1.2: A bilateral filter mixes together pixels with other nearby pixels that have a similar color. This can be expressed as a Gauss transform by setting the *values* $\vec{v}_i$ to be pixel colors, and setting the *positions* $\vec{p}_i$ to five-dimensional vectors incorporating both the spatial location and the color of the pixel. The scale terms σ_s and σ_c control the size of the filter in space and color respectively.

The bilateral filter can thus be expressed by a Gauss transform in which the position vectors are five-dimensional, containing the location of the pixel and also its color. The pixel at x, y with color r, g, b has position vector $[x/\sigma_s, y/\sigma_s, r/\sigma_c, g/\sigma_c, b/\sigma_c]$, where σ_s and σ_c control the spatial and color-space extent of the filter.

A bilateral filter is a moderately effective way to denoise an image, as it smooths an image without destroying strong edges. However, it is more useful as a means of decomposing an image into multiple layers (Figure 1.3). By treating the bilateral-filtered image as a *base* layer, we can subtract it from the input to obtain a *detail* layer. The base and detail layers can be separated processed and recombined for a variety of effects.

For example, a bilateral filter with small extent can be used to isolate the fine detail in an image. Amplifying it and then adding back in the base layer will sharpen the image. This sharpening method avoids the unwanted halos around strong edges common to convolution-based sharpening, because the strong edges are represented in the unchanged base layer. By changing the size of the bilateral filter (using σ_s and σ_c) we can isolate tone and contrast at different scales, and independently manipulate

Figure 1.3: The bilateral filter can be used to decompose an image (top left) into base (top middle) and detail (top right) layers. This isolates tone and contrast at a particular scale. The layers can then be modified and recombined for a variety of effects. In the bottom left we decompose the image using a large bilateral filter, amplify the detail layer, and then recombine. The result is a amplification of coarse-scale contrast. In the bottom center we apply the same process using a smaller bilateral filter, which gives a sharpening effect. Using the a simple Gaussian blur for the decomposition instead of the bilateral filter results in unwanted effects (bottom right). Note for example the halo around the back of each dog's head.

them, making this filter a very powerful tool for photographic image processing. For a full exploration of this approach see Bae et al. [7].

A different use of a bilateral decomposition arises in tone-mapping high-dynamic-range images to be displayed on low-dynamic-range displays [22]. Here we preserve the detail layer as-is, and reduce the contrast of the base layer. This preserves local detail while compressing global contrast into a viewable range.

The bilateral filter was independently invented by Tomasi and Manduchi [45], Smith and Brady [42], and Aurich and Weule [6]. Apart from the applications above, it has seen use in video denoising [9], abstraction and stylization [47], optical flow regularization [48], smoothing photon density maps in rendering [46], and even mesh denoising [29].

1.2 The joint-bilateral filter

In some applications it is useful to smooth an image without crossing strong edges in some other reference image. This is referred to as a joint- or cross-bilateral filter. Such a filter can be expressed with a Gauss transform by deriving the value vectors $\vec{v}_i$ from the image to be smoothed, and the position vectors $\vec{p}_i$ from the reference image.

The joint-bilateral filter was invented independently by Eisemann and Durand [23] and Petschnigg et al. [39] and was used by each for combining images taken with and without flash.

One simple yet effective application of a joint-bilateral filter is to reduce color noise in a photograph (Figure 1.5). Three-dimensional color can be decomposed into one-dimensional luminance (or brightness) and two-dimensional chrominance. Most real objects have piecewise-constant chrominance, and so chrominance noise in a photograph is much more objectionable to humans than luminance noise. Furthermore, the color matrices used in digital cameras to convert from the color space of the sensor (in which the channels are highly correlated) to a standard color space often amplify chrominance noise while reducing luminance noise. We can therefore improve the appearance of a noisy digital photograph by smoothing the chrominance terms without crossing strong edges in luminance, using a joint bilateral filter.

1.3 The joint-bilateral upsample

A variant of the joint-bilateral filter can also be used to increase the spatial resolution of an image given a higher-resolution reference. This is done by interpolating the low-resolution data in a manner that does not cross strong edges in the high-resolution reference image. This technique was proposed by Kopf et al. [30]. It allows you to compute any expensive but piecewise-smooth function of the image at low-resolution, and then cheaply upsample the result. Kopf uses it for upsampling the results of tone-mapping, colorization, and depth estimation.

We can model a joint-bilateral upsample as a Gauss transform by distinguishing between the positions associated with the input values, and those associated with the

Non-local Means: $\vec{p}_i = \left(\frac{x_i}{\sigma_s} \; \frac{y_i}{\sigma_s} \; \right)$  $\left.\right)$ $\vec{v}_i = \begin{pmatrix} r_i & g_i & b_i & 1 \end{pmatrix}$

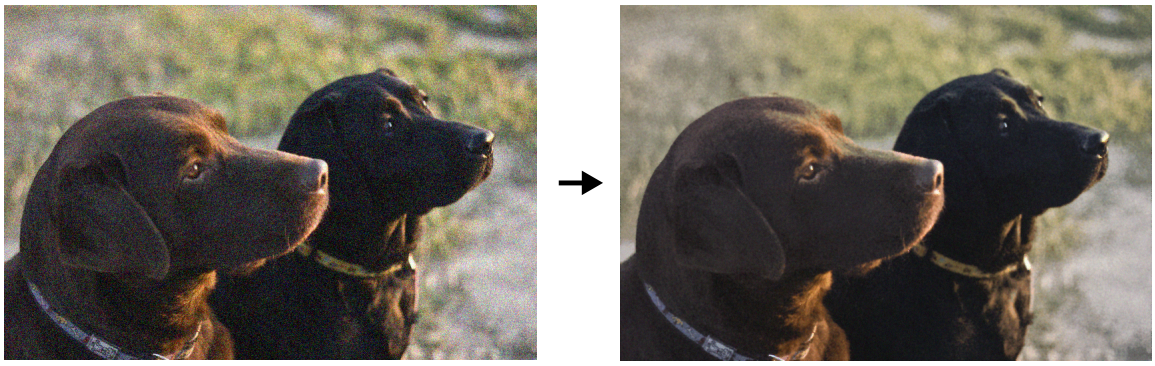

Figure 1.4: Non-local means reduces noise by mixing each pixel with other nearby pixels that have a similar local neighborhood. To express non-local means as a Gauss transform, we again set the *values* $\vec{v}_i$ to homogeneous pixel colors. The *positions* $\vec{p}_i$ include two spatial terms, and also a description of the neighborhood around pixel i. This description may simply be a list of the pixel colors in a local window, or it may be some other descriptor such as the response to a bank of filters. By mixing together *values* with similar *positions*, we mix together pixels that have similar local neighborhoods.

output values. We will denote this distinction by replacing $\vec{p}_j$ in our original Gauss transform with $\vec{\hat{p}}_j$:

$$\vec{\hat{v}}_i = \sum_j e^{-|\vec{\hat{p}}_i - \vec{p}_j|^2/2} \vec{v}_j \tag{1.2}$$

Once again, the value vectors will be homogeneous color, and the position vectors will encode color and spatial location. The input values $\vec{v}_j$ come from the low resolution image to be upsampled. The corresponding input positions $\vec{p}_j$ are derived from an appropriately downsampled version of the high-resolution reference image. The output positions $\vec{\hat{p}}_i$ come from the high-resolution reference image without downsampling. This results in filtered output values $\vec{\hat{v}}_i$ at the same resolution as the reference image.

Figure 1.5: At the top is a crop of a photograph captured with a Canon 400D at ISO 1600. This type of image poses problems for most denoising algorithms. First, the noise is not Gaussian; it includes the effect of "hot pixels", which, after demosaicing, become small brightly colored blobs. Second, the fine fur texture is difficult to algorithmically distinguish from noise. The second row shows the effect of a bilateral filter. It is somewhat effective at removing the noise, but it destroys much of the fine detail of the fur. The third row shows the effect of denoising chrominance only using a joint bilateral filter. It retains the fur texture, but removes only the color noise. The final row shows the effect of a non-local means filter, which retains most of the texture, while removing much of the noise. For this image, non-local means has likely made the largest improvement in terms of signal-to-noise ratio, but the joint-bilateral filter produces a perceptually superior result.

1.4 Non-local means

Noise is inherent to the physics of photography, and so denoising has been an active area of research. One algorithm that has proved to be effective is non-local means by Buades et al. [13]. Non-local means averages pixels together with other nearby pixels that have similar local neighborhoods. In this way, pixels along an edge will be averaged with other pixels along that edge, pixels in a flat region will be averaged with all other pixels in that region, and pixels in a highly textured area will only be averaged with pixels that exhibit the same local structure.

The attractive property of non-local means is that it does not enforce a local smoothness prior on the image (as does a Gaussian blur, or a bilateral filter). Instead it enforces self-similarity; similar patches in an image should have a central pixel with a similar color. This lets us denoise without overly smoothing the image. It also allows us to relax or even drop the notion of spatially-nearby, and search for matching patches across the entire image, or an entire burst of images, or a multi-view set of images, or *any other potentially helpful image data*. Of course with such a large set of potential patches to examine, an exhaustive search at every pixel in an image would take a prohibitive amount of time, and so accelerating non-local means is vital if it is to be useful.

We can express non-local means as a Gauss transform by once again setting our value vectors to be homogeneous pixel colors, and setting our position vectors to be some compact description of the neighborhood around each pixel, and optionally the spatial location of the pixel (Figure 1.4).

The simplest way to describe the local neighborhood of a pixel is to simply list the colors of all pixels within some window around that pixel. However, this results in a very high-dimensional position vector. For example, a 5x5 patch with three color channels would result in a 75-dimensional position vector, which increases to 77 when spatial terms are added.

A better approach is to use the output of a bank of filters. If we wish to use the same notion of patch distance as the naive method above, we can reduce the dimensionality of the space of image patches whilst best preserving distances between

patches using principal components analysis (PCA). The *eigenpatches* with large eigenvalues then form the filter bank.

This approach was explored by Tasdizen [43], who found that reducing patch-space to 6 dimensions actually improves denoising performance on typical images. The discarded eigenpatches with smaller eigenvalues tend to correspond to noise, and including them does more harm than good.

It is not clear if this notion of patch distance is the optimal one. One could also imagine using rotation- or scale-invariant local descriptors to exploit self-similarity in an image across multiple orientations or scales. As this dissertation focuses on accelerating the underlying Gauss transform, we leave such explorations to others.

1.5 This dissertation

The Gauss transform can express a rich and useful family of image processing algorithms, but its use has been hindered by its computational complexity; in the naive form every output value is a sum over all input values. We require, instead, approximate algorithms with the following key properties:

1. Scales well with respect to the number of input values, as images typically involve many millions of pixels.

2. Scales well with respect to the dimensionality of the position vectors, as in many applications we will want to use position vectors with 8 or more dimensions.

3. Scales well with respect to the size of the filter, so that we are not restricted to filters with small spatial support.

4. Generalizes across all sizes and dimensionalities of Gauss transform, and requires no particular structure to the input.

5. Produces output that is visually equivalent to that of an exact Gauss transform.

The primary contributions of this dissertation are two such accelerations of the Gauss transform based on two novel data structures. Both data structures explicitly

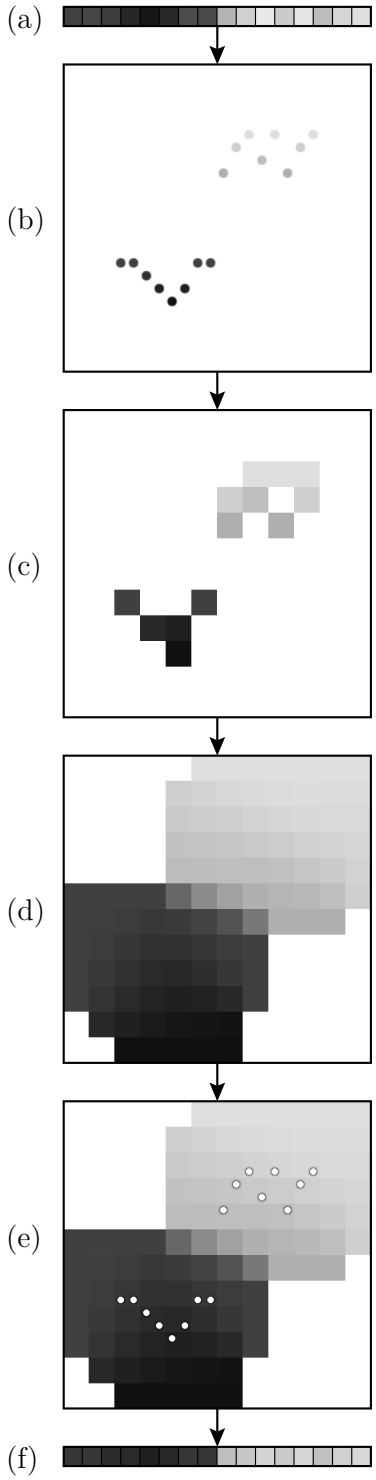

Figure 1.6: The splat-blur-slice pipeline for accelerating Gauss transforms. Gauss transforms mix together *values* that have similar *positions*. This can be accelerated with an explicit sampling of the space of position vectors.

In this example we smooth a one-dimensional grayscale signal without losing the strong edge (a). First we treat the input data as a point cloud in two dimensions (b). Each pixel in the input becomes a point with *position* given by its location in the input and its brightness, and *value* given by brightness alone.

We then *splat* these points onto our explicit sampling of position-space (c). In this case we use a regular grid to sample position space, and a nearest-neighbor splatting filter. As we're intending to blur within this space, the sampling can be much lower resolution than the input.

A good sampling of position-space makes it easy to blur within that space (d). In the case of a grid we can blur separately along each axis to construct a full Gaussian blur.

Finally, we *slice* out our output, by sampling the representation of position-space at the original positions (e). Here we use a nearest-neighbor reconstruction. This gives us a piecewise smooth version of the input, with the strong edge intact (f).

While this toy example uses nearest-neighbor filters, for real applications it is necessary to use some kind of interpolation scheme to avoid artifacts. Unfortunately this makes a grid untenable for higher-dimensional position vectors, as grid interpolation in d dimensions costs at least $O(2^d)$ time and memory.

sample the space of *position* vectors in a way that makes it easy to mix together values with similar positions. Filtering can then be implemented with a three stage pipeline. First we *splat* or resample our input data onto the chosen sampling of *position*-space. Then we *blur* across this sampling, to mix together values with similar positions. Finally we *slice*, or resample back onto the original position vectors. We can also slice onto some other set of position vectors, for example to perform a joint-bilateral upsample. Figure 1.6 illustrates this pipeline for the case of two-dimensional position vectors sampled on a regular grid.

The first data structure described herein is the permutohedral lattice (Chapter 3). It samples position-space at the vertices of the lattice A^*, which tessellates space with uniform simplices. Splatting and slicing are done with barycentric interpolation within each simplex. Blurring is done by separately blurring along each lattice axis. For n inputs with d-dimensional position vectors, filtering using the permutohedral lattice takes $O(d^2n)$ space and time. The lattice out-performs a regular grid at dimensionalities above three, making it preferable for all but the simplest Gauss transforms. This data structure was first published as Fast High-Dimensional Filtering Using the Permutohedral Lattice [1], which was joint work with Jongmin Baek and Abe Davis.

The second data structure is the Gaussian kd-tree, described in Chapter 4. The Gaussian kd-tree aggregates the initial position vectors into a kd-tree with one sample at each leaf. Splatting and slicing are then all done using randomized Gaussian queries, which find samples around a query position with a probability proportional to a Gaussian centered at that query position. The Gaussian kd-tree does not have an explicit blur stage, but instead folds blurring into splatting and slicing. Filtering using the Gaussian kd-tree has a time complexity of $O(dn \log n)$, and begins to out-perform the permutohedral lattice as the dimensionality grows beyond 12. This data structure was first published as Gaussian KD-Trees for Fast High-Dimensional Filtering [2], which was joint work with Natasha Gelfand, Jennifer Dolson, and Marc Levoy.

Importantly, neither data structure has a time or space complexity that grows with the filter size. In fact, the larger the filter size, the more coarsely we can sample the space of position vectors, which means that this class of algorithms actually runs faster as the footprint of the filter increases.

In the following chapter we will review existing accelerations of the algorithms described so far. Then, following our description of the permutohedral lattice (Chapter 3) and the Gaussian kd-tree (Chapter 4), we compare these two data structures and benchmark them against existing work (Chapter 5). In Chapter 6 we will examine some new applications of the Gauss transform made possible by these data structures, and finally in Chapter 7 we lay out some future directions of research in this area and conclude.

Chapter 2

Prior Work Accelerating Gaussian Filtering

The techniques described in the previous chapter have seen widespread use in computer vision, computational photography, and medical imaging. However, their naive evaluation is quite slow; at each pixel, all of the above techniques require searching for other pixels that have similar position vectors. Due to the rapid falloff of the Gaussian, pixels with position vectors that are dissimilar can be omitted from the summation with minimal effect. If the position vectors include spatial terms, the search can thus be constrained to a local window, but for non-trivial window sizes this is still quite expensive. For applications that use non-local means, in which the spatial terms are often weak or absent (hence *non-local*) a naive evaluation is intractable.

The computational expense of these methods limits their utility, and so a variety of attempts have been made accelerate them. Most accelerations only apply to subclasses of the applications above, but are nonetheless instructive to consider.

2.1 Accelerating the bilateral filter

Most work on accelerating the bilateral filter considers the restricted case of filtering grayscale images. In this case we average a pixel's intensity with other nearby pixels

with similar intensities. In the context of Equation 1.1 the corresponding position vectors are three-dimensional, and contain two spatial terms and one brightness term. Equation 1.1 then becomes:

$$\vec{v}_i = \sum_j G_{ij} e^{-|I_i - I_j|^2/2} \vec{v}_j$$

Where I_i is intensity at pixel i, and G_{ij} is a Gaussian in the spatial distance between pixels i and j. We use Gaussians of standard deviation one for simplicity. If we fix a particular intensity level $I_i = \alpha$, then the equation becomes:

$$\vec{v}_i = \sum_j G_{ij} \left(e^{-|\alpha - I_j|^2/2} \vec{v}_j \right)$$

None of the terms in the parentheses depend on pixel i, so this is merely a Gaussian blur of a modulated version of the image, with the modulation at pixel j given by $e^{-|\alpha - I_j|^2/2}$. A Gaussian blur is cheap to compute using a variety of methods, and this output will be accurate for pixels with intensity close to α.

One can compute intermediate filtered images for a range of different α and then interpolate between the most appropriate ones for each pixel based on its intensity. This is the approach taken by Durand and Dorsey [22]. The same fundamental approach was taken by Porikli [40]. The two approaches differ only in how they filter the intermediate images. Whereas Durand and Dorsey use a Fourier transform, Porikli uses integral images of powers of the images to allow for the rapid computation of any filter expressible as a low-order polynomial.

One can also reorder the computation slightly, and sweep through different values of α computing intermediate images. Given the two most recent intermediate images with $\alpha = \alpha_n$ and $\alpha = \alpha_{n-1}$ we can compute the output at all pixels with intensities between α_n and α_{n-1}. This requires multiple passes through the image, but only requires storing two intermediate images at a time, and so uses less memory. This approach was taken by Yang et al. [50].

Paris and Durand [35] address the issue of memory use in a different way. Given

that we blur the intermediate images, it is sufficient to construct them at low-resolution, use a small fixed-sized convolution to blur, and then linearly interpolate the result both between intermediate images and within each image. The set of intermediate images then becomes a fairly compact three-dimensional volume, with the output constructed via trilinear interpolation. This is precisely the splat-blur-slice pipeline of Figure 1.6.

Chen et al. [15] then made an interesting observation. If we consider only the last homogeneous coordinate of the value vectors, which in the input is the constant 1, our expression becomes:

$$\sum_j G_{ij} e^{-|\alpha - I_j|^2/2}$$

As α varies, this expression gives a smoothed local histogram centered at pixel i, with spatial falloff given by G_{ij}, and a Gaussian reconstruction filter across intensities. Chen calls the volume of Paris and Durand the *bilateral grid*, and treats it as a fast local histogram transform for a variety of applications.

This family of work is all built around a single key idea: we can linearize the bilateral filter by treating the input image as a 2D manifold embedded in the discretized three-dimensional space x, y, α (or intensity). In this space the filter becomes a three-dimensional Gaussian blur, which can be approximated and accelerated with a variety of methods.

This family of accelerations hints at our more general approach for accelerating Gauss transforms. The purpose of Equation 1.1 is simply to mix together values that have similar position vectors. By discretizing the space of position vectors, and embedding the input within that space by resampling (*splatting*), we can mix the values with a conventional Gaussian blur (*blurring*). The output image can then be extracted by resampling back into image space (*slicing*).

2.2 Accelerating non-local means

Accelerations of non-local means have typically used different strategies. Naive non-local means compares the $m \times m$ patch around each pixel x, y with every other such patch in the image. m is typically between 3 and 9. The pixel value is then replaced by a weighted average of every other pixel, with the weights given by a Gaussian in the Euclidean distance between the corresponding patches. This is a Gauss transform with the position vector for pixel i given by the $m \times m$ patch surrounding pixel i. For input image I of size $w \times h$, and equally-sized output image O:

$$\begin{aligned} O(x,y) &= \sum_{x'=0}^{w}\sum_{y'=0}^{h} e^{-d(x,y,x',y')/2\sigma^2} I(x',y') \\ d(x,y,x',y') &= \sum_{i=-m/2}^{m/2}\sum_{j=-m/2}^{m/2} [I(x+i,y+j) - I(x'+i,y'+j)]^2 \end{aligned}$$

The first acceleration most users of non-local means employ is to only search for similar patches within a $k \times k$ spatial window, rather than searching across the entire image:

$$O(x,y) = \sum_{\alpha=-k/2}^{k/2}\sum_{\beta=-k/2}^{k/2} e^{-d(x,y,\alpha,\beta)/2\sigma^2} I(x+\alpha,y+\beta) \quad (2.1)$$

$$d(x,y,\alpha,\beta) = \sum_{i=-m/2}^{m/2}\sum_{j=-m/2}^{m/2} [I(x+i,y+j) - I(x+\alpha+i,y+\beta+j)]^2 \quad (2.2)$$

It is debatable whether or not this restriction hurts denoising performance for non-trivial k (e.g. $k > 15$). Recall that non-local means enforces a self-similarity prior in an image; pixels with similar neighbors are likely to be similar. While the restriction to a $k \times k$ search window may cause the algorithm to miss out on globally repeated structure in an image, local self-similarity (such as the self-similarity of patches along an edge) is more common than global self-similarity.

We can further accelerate non-local means with early termination of the summation in Equation 2.2. $d(x, y, \alpha, \beta)$ computes the square Euclidean distance between the patch of pixels around (x, y) and the patch of pixels around $(x + \alpha, y + \beta)$. This sum is fed into the Gaussian expression $e^{-d(x,y,\alpha,\beta)/2\sigma^2}$, so if the sum grows larger than around $9\sigma^2$ it may as well be infinite; the patches don't match, the Gaussian will be near-zero, and this value of α, β will have minimal effect on the output at x, y and can be skipped.

Mahmoudi et al. [32] explored some more sophisticated early termination criteria. By precomputing the mean and average gradient direction of each patch, and comparing those before computing $d(x, y, \alpha, \beta)$, we can reject patches that are unlikely to be similar in the Euclidean sense.

Darbon et al. [19] take a different approach, and instead move iteration over α and β to the outer loop, computing entire slices of $d(x, y, \alpha, \beta)$ at a time. First note that if α and β are constant, then Equation 2.2 is just the convolution of the image $[I(x, y) - I(x + \alpha, y + \beta)]^2$ by a square filter of size $m \times m$. This can be computed in $O(wh)$ time using integral images (as they do), or by recursive filtering. The result is a slice of $d(x, y, \alpha, \beta)$ at a fixed (α, β), which we can then use to compute one term in the summation in Equation 2.1. By refactoring non-local means to expose this convolution, and then accelerating that convolution, they reduce the total cost from $O(whk^2m^2)$ to $O(whk^2)$.

Both of these acceleration strategies improve runtime with respect to patch size m, but do not improve runtime with respect to the search window size k. Recall, however, than m is typically small (3-9), and k is typically larger (> 15). So the benefit is limited, and we still do not have a tractable truly *non-local* means. Furthermore, these strategies are largely made moot by using a more compact description of a local neighborhood, such as Tasdizen's [43] PCA terms.

For the larger search windows we wish to use, treating every patch within that window as a potential match is computationally wasteful. Only patches that are similar have any effect on the output; for the others, the weight is too small to have any influence. Brox et al. [12] take advantage of this by clustering all patches from the input image into a patch-space-partitioning tree. The search for matching patches

is implemented with a descent of this tree. During a descent, whole subtrees can be ignored if they are known to be far from the query patch (either due to being far away in the image, or being far away in patch-space).

The algorithms described in this dissertation use an approach more similar to the accelerations of the bilateral filter, in which we discretized space and intensity and resample onto that representation. However, to apply the same technique to non-local means we'll have to discretize the space of all image patches, which is of significantly higher dimensionality.

2.3 Accelerating general Gauss transforms

Accelerating Gauss transforms in the general case has also been of interest outside of image processing, as Gauss transforms occur frequently in many other domains, including computer vision [24], artificial intelligence [37][18], physics [28], and finance [10].

A popular method for accelerating Gauss transforms is the *fast Gauss transform* of Greengard and Strain [28], which is a fast multipole algorithm of the type described earlier by Greengard [27]. The fast Gauss transform is based on the idea that the influence of values with nearby position vectors can be aggregated into a function centered at a single position. Greengard and Strain place boxes around clusters of position vectors and aggregate their influence into a single function centered at the center of the box. More formally, given a cluster of positions $\vec{p}_j$, with values $\vec{v}_j$, and bounding box centered at $\vec{p}_c$, the influence of those values at at a far-away output position $\vec{\hat{p}}$ is a sum of Gaussians:

$$\sum_j e^{|\vec{\hat{p}}-\vec{p}_j|^2/2}\vec{v}_j$$

We can pull the exponential term outside the summation, and treat it as a single Gaussian centered at $\vec{p}_c$. To account for the error thereby introduced we replace each constant $\vec{v}_j$ with a function f_j which depends on $\vec{p}_j$, $\vec{p}_c$, and $\vec{v}_j$:

$$\begin{aligned}\sum_j e^{|\vec{p}-\vec{p}_j|^2/2}\vec{v}_j &= e^{|\vec{p}-\vec{p}_c|^2/2}\sum_j f_j(\vec{p}) \\ \text{where } f_j(\vec{p}) &= \frac{\vec{v}_j e^{|\vec{p}-\vec{p}_j|^2/2}}{e^{|\vec{p}-\vec{p}_c|^2/2}}\end{aligned}$$

Greengard and Strain show that we can safely replace f_j with its Taylor expansion about $\vec{p}_c$ truncated after a small number of terms. The summation of polynomials can be simplified to a single polynomial by summing coefficients, so we can precompute a single polynomial for each cluster F_c:

$$\sum_j e^{|\vec{p}-\vec{p}_j|^2/2}\vec{v}_j \approx e^{|\vec{p}-\vec{p}_c|^2/2}F_c(\vec{p}-\vec{p}_c)$$

The coefficients of each polynomial approximation to f_j take $O(1)$ time to compute, and so for a cluster of m points, the coefficients of F_c can be computed in $O(m)$ time. Each point belongs to one such cluster, so for n total inputs, precomputing the entire set of polynomials F_c costs $O(n)$ time.

The fast Gauss transform then proceeds as follows. First, cluster the position vectors into boxes. Greengard and Strain do this by simply dividing space into a regular grid of hypercubes, each sized proportionally to the standard deviation of the Gaussian. Then compute the coefficients of F_c for each box. For n total input values this takes $O(n)$ time. Next, for each position we compute the output value by evaluating the influence of some fixed number of nearby boxes. This also takes $O(n)$ time, reducing the algorithm to linear for low-dimensional cases. Unfortunately this algorithm scales poorly as the dimensionality increases. For d dimensions, $O(2^d)$ terms are required in each polynomial F_c, and $O(2^d)$ boxes are *nearby* to each query position.

The fast Gauss transform is also far more accurate than is necessary for image processing. In fact, if we simply set $f_j = \vec{v}_j$, we obtain:

$$\sum_j e^{|\vec{p}-\vec{p}_j|^2/2}\vec{v}_j \approx e^{|\vec{p}-\vec{p}_c|^2/2}\sum_j \vec{v}_j$$

This is in fact the approximation used by the bilateral grid described earlier [35]. Both algorithms divide position-space into a regular grid of cubes, accumulate the value of the points contained within each cube (a process which we earlier termed *splatting*), and then produce the output by sampling the value of nearby cubes using a Gaussian reconstruction kernel. The bilateral grid approximates this last step with two stages – a separable Gaussian blur across the grid (*blurring*), followed by multilinear interpolation (*slicing*). Both algorithms also have the same exponential scaling with dimensionality, making them slow for dimensionalities above 3, and not useful for dimensionalities above 5.

Yang et al. [49] introduced the *improved fast Gauss transform*, which address the fast Gauss transform's scaling problems in two ways. First, they use an alternative expansion for f_j that truncates after a number of terms that grows only polynomially with dimensionality. Second, they cluster position vectors using a more general clustering algorithm: farthest-point clustering [25]. Locating nearby clusters during slicing is done by iterating over all clusters, or by storing the cluster centers in an approximate-nearest-neighbor tree [5].

Unfortunately large numbers of clusters result from the sizes of Gauss transform used in image processing, so we cannot simply iterate over clusters. Using an approximate-nearest-neighbor tree introduces an extra factor of $\log n$ and reintroduces an exponential dependence on dimension.

There is a strong parallel between these two approaches and the two data structures described in this dissertation (see Figure 2.1). The permutohedral lattice (Chapter 3) operates similarly to the fast Gauss transform and the bilateral grid, but solves the dimensionality dependence by dividing position-space into simplices instead of hypercubes. The resulting time complexity is quadratic in dimensionality rather than exponential: $O(d^2 n)$. The Gaussian kd-tree (Chapter 4) uses a tree of aggregated values much like the improved fast Gauss transform, but uses randomized tree queries to *splat* and *slice*, while incurring only a linear cost in dimension, resulting in a time complexity of $O(dn \log n)$.

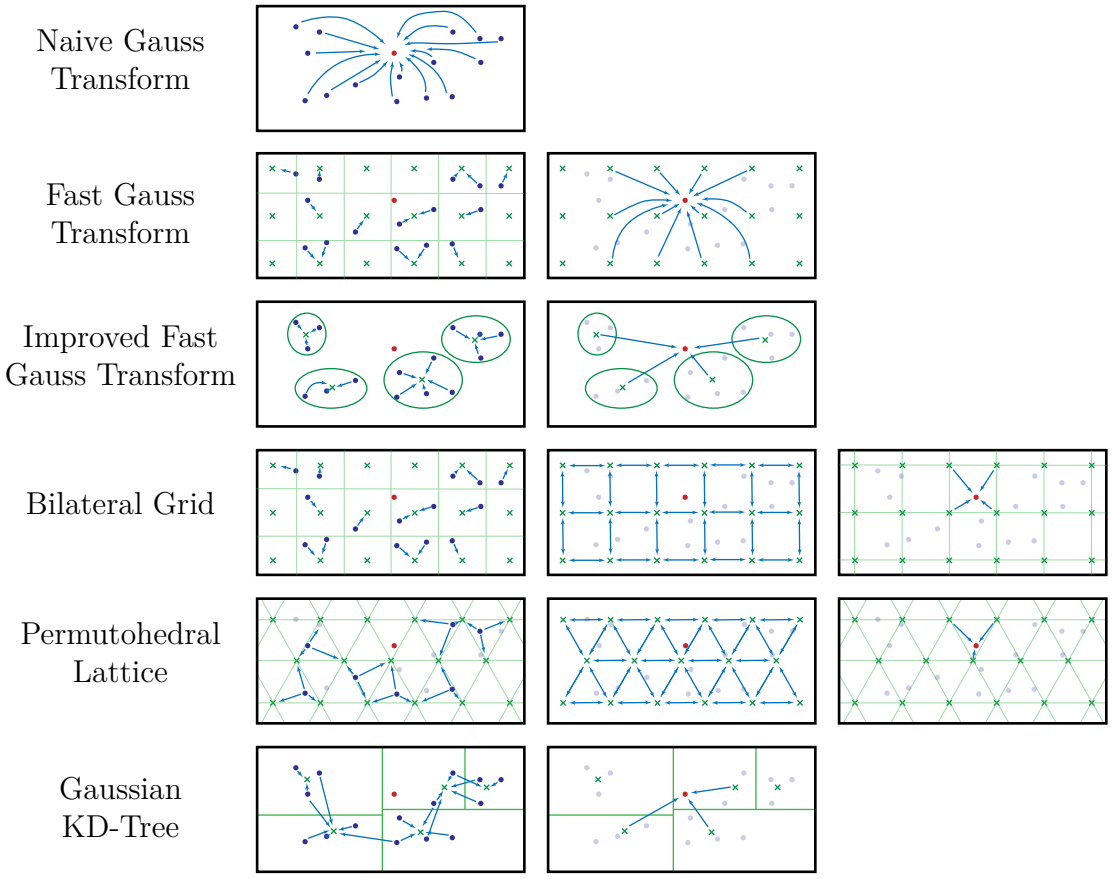

Figure 2.1: In the naive Gauss transform, each output (the red dot) gathers data from every input (the blue dots). The fast Gauss transform accelerates this with a uniform grid (the green crosses), where every input contributes to a polynomial approximation about the nearest grid point. The output then gathers from every grid point within some Gaussian-weighted window. This approach scales poorly with dimensionality, and so the improved fast Gauss transform instead clusters inputs, computing a polynomial approximation at each cluster. The output gathers from all nearby clusters using Gaussian weights. The bilateral grid is similar to the fast Gauss transform; it accumulates input values on a grid. However, it trades accuracy for speed by only accumulating constant values rather than polynomial coefficients, and factoring the Gaussian-weighted gather into a separable Gaussian blur followed by multilinear sampling. The permutohedral lattice operates similarly, but uses the lattice A^* instead of a grid. Barycentric weights within each simplex are used to resample into and out of the lattice. Finally, the Gaussian kd-tree clusters inputs using a kd-tree. Each input contributes to randomly-selected nearby cluster centers. Each output then similarly gathers from randomly-selected nearby cluster centers.

Chapter 3

The Permutohedral Lattice

The permutohedral lattice has several key properties that make it effective for fast approximate Gauss transforms using the splat-blur-slice pipeline (Figure 1.6):

1. The lattice tessellates position-space with uniform simplices, so we can use barycentric interpolation to splat the input onto the lattice vertices.

2. It is cheap to compute the vertices of the simplex enclosing any query position, including barycentric coordinates ($O(d^2)$ time). This makes the splatting and slicing stages fast.

3. A Gaussian blur on the lattice can be performed separately along each axis, and the neighbors of a lattice point are trivial to compute, so the blur stage is fast.

In this chapter we describe the lattice and its properties, state the algorithms by which we splat, blur, and slice, analyze the resulting Gauss transform, and discuss implementation details for CPU and GPU.

3.1 Definition

The d-dimensional permutohedral lattice is the projection of the scaled regular grid $(d+1)\mathbb{Z}^{d+1}$ along the vector $\vec{1} = [1, \ldots, 1]$ onto the hyperplane $H_d : \vec{x}.\vec{1} = 0$, which

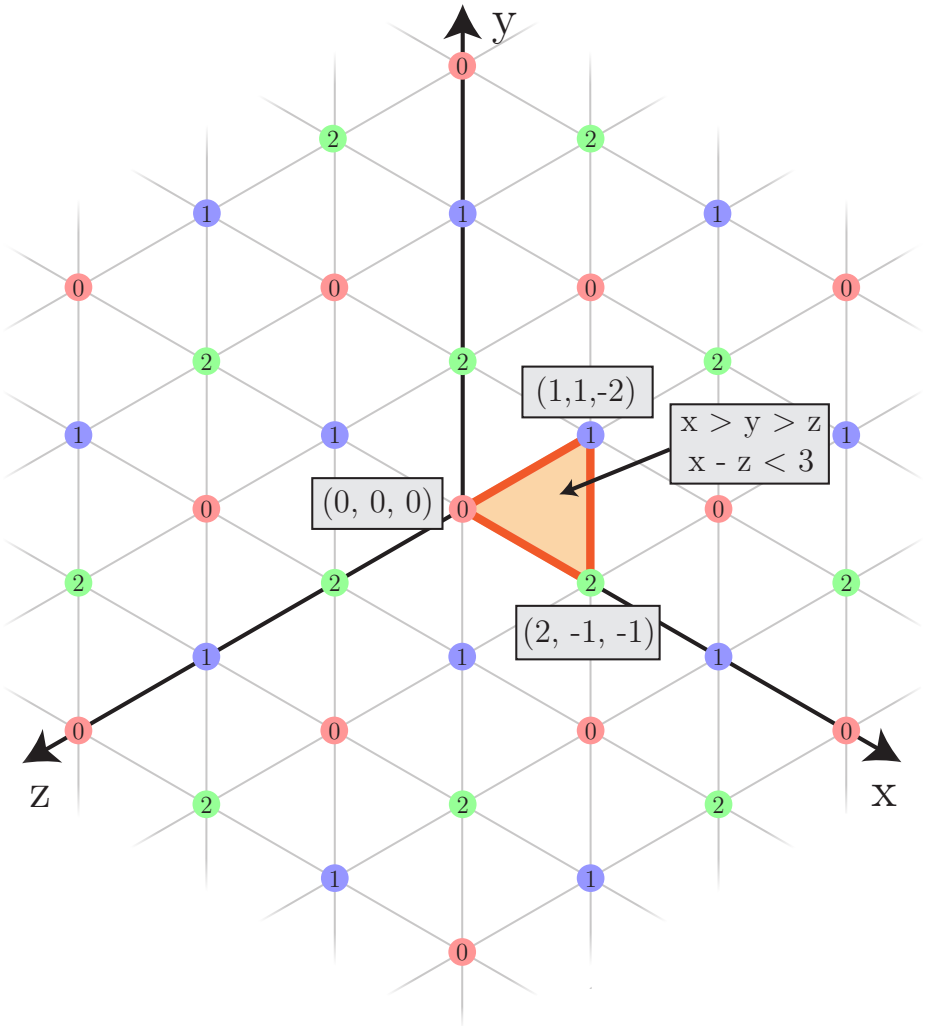

Figure 3.1: The d-dimensional permutohedral lattice is formed by projecting the scaled grid $(d+1)\mathbb{Z}^{d+1}$ onto the plane $\vec{x} \cdot \vec{1} = 0$. This forms the lattice $(d+1)A_d^*$, which we call the permutohedral lattice. Lattice points have integer coordinates with a consistent remainder modulo $d+1$. In the diagram above, which illustrates the case $d = 2$, points are labeled and colored according to their remainder. The lattice tessellates space with uniform simplices, each simplex having one vertex of each remainder. The simplices are all translations and permutations of the canonical simplex (highlighted), which is defined by the inequalities $x_0 > x_1 > \ldots > x_d$ and $x_0 - x_d < d+1$.

is the subspace of $\mathbb{R}^{d+1}$ in which coordinates sum to zero. It is hence spanned by the projection of the standard basis for $(d+1)\mathbb{Z}^{d+1}$ onto H_d:

$$B_d = \begin{pmatrix} d & -1 & \dots & -1 \\ -1 & d & \dots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \dots & d \end{pmatrix}$$

Note that each of the $(d+1)$ basis vectors (columns of B_d) has coordinates that sum to zero, and that each coordinate of each basis vector has a consistent remainder modulo $d+1$. Both of these properties are preserved when taking integer combinations, so points in the lattice are those points with integer coordinates that sum to zero and have a consistent remainder modulo $d+1$. For example when $d=3$, one lattice point is $[2,-10,6,2]$, as its integer coordinates all have the same remainder modulo $d+1$, and they sum to zero.

We describe a lattice point whose coordinates have a remainder of k as a "remainder-k" point. In Figure 3.1 we show the lattice for $d=2$, and label each lattice point by its remainder.

3.2 Key properties

3.2.1 The permutohedral lattice tessellates space with uniform simplices

Consider the d-dimensional simplex whose vertices $\vec{s_0}, \dots, \vec{s_d}$ are given by:

$$\vec{s_k} = [\underbrace{k, \dots, k}_{d+1-k}, \underbrace{k-(d+1), \dots, k-(d+1)}_{k}]$$

We call this simplex the canonical simplex. For example, when $d=4$ the vertices are the columns of:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & -1 \\ 0 & 1 & 2 & -2 & -1 \\ 0 & 1 & -3 & -2 & -1 \\ 0 & -4 & -3 & -2 & -1 \end{pmatrix}$$

Note that $\vec{s_k}$ is a lattice point of remainder k (i.e. its coordinates are all congruent to k modulo $d+1$), and the simplex includes one point of each remainder. The vertices of this simplex are the boundary cases of the inequalities $x_0 \geq x_1 \geq \ldots \geq x_d$ and $x_0 - x_d \leq d + 1$, and a point lies within the simplex if and only if it obeys these inequalities.

Now consider any permutation ρ of the coordinates of the canonical simplex. Each ρ induces a corresponding ordering of the coordinates $x_{\rho(0)} \geq x_{\rho(1)} \geq \ldots \geq x_{\rho(d)}$, and the inequality $x_{\rho(0)} - x_{\rho(d)} \leq d + 1$. Taking the union of these inequalities across all $(d + 1)!$ simplices results in the set $\{x_i | \max_i x_i - \min_i x_i \leq d + 1\}$ (the central hexagon in Figure 3.2), which is in fact the set of all points which have the origin as their closest remainder-0 point:

Proposition 3.2.1. *Given $\vec{x} \in H_d$, the following two statements are equivalent:*

1. *The closest remainder-0 point to $\vec{x}$ is the origin.*
2. $\max_k x_k - \min_k x_k \leq d + 1$.

Proof. The closest remainder-0 point has the form $(d + 1)\vec{z}$ for some $\vec{z} \in \mathbb{Z}^{d+1}$. Fix two distinct indices $i, j \in \{0, \ldots, d\}$ and define $\vec{z'}$ where

$$z'_k := \begin{cases} z_k + 1, & k = i, \\ z_k - 1, & k = j, \\ z_k, & \text{otherwise.} \end{cases}$$

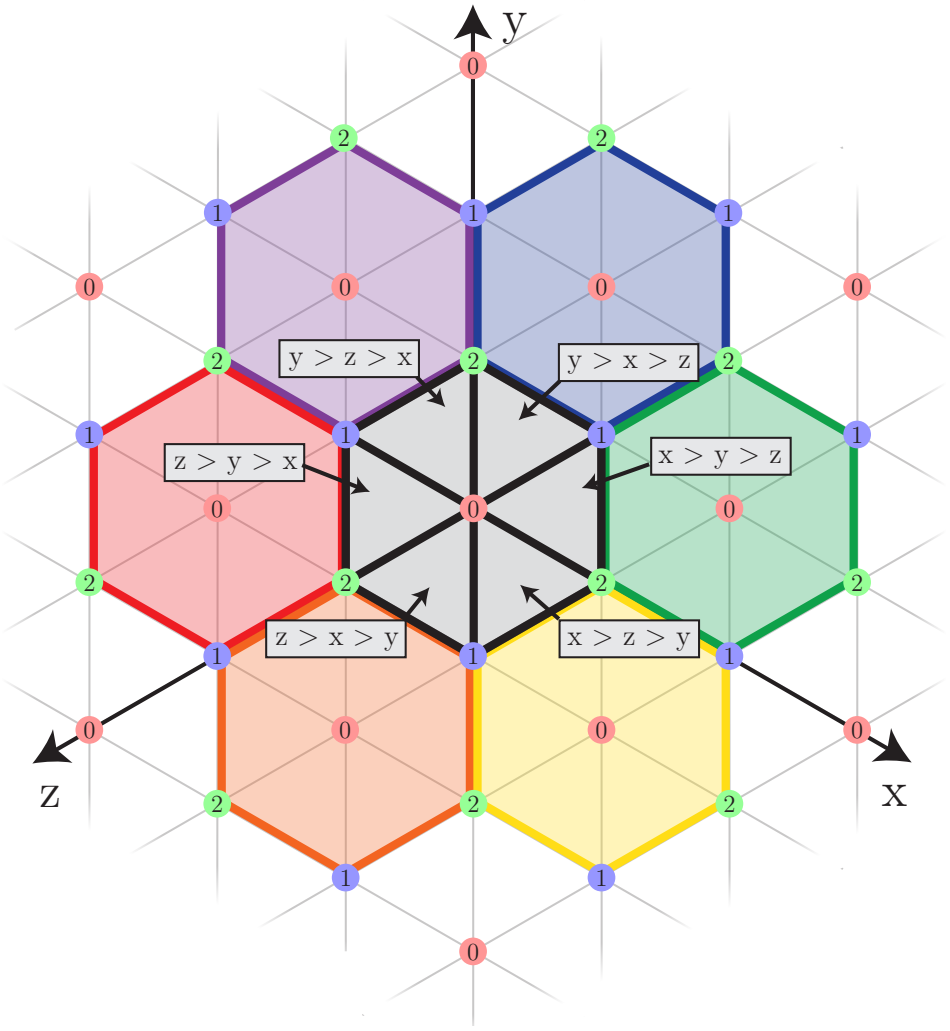

Figure 3.2: When using the permutohedral lattice to tessellate the subspace H_d, any point $\vec{x} \in H_d$ is enclosed by a simplex uniquely identified by the nearest remainder-0 lattice point $\vec{l}_0$ (the zeroes highlighted in red at the center of each hexagon) and the ordering of the coordinates of $\vec{x} - \vec{l}_0$ (the triangles within each hexagon). The nearest remainder-0 lattice point can be computed with a simple rounding algorithm, and so identifying the enclosing simplex of any point and enumerating its vertices is computationally cheap ($O(d^2)$).

By choice of $\vec{z}$, it must be that $(d+1)\vec{z}$ is closer to $\vec{x}$ than is $(d+1)\vec{z'}$. Therefore,

$$\begin{aligned}
0 &\leq \|(d+1)\vec{z'} - \vec{x}\|^2 - \|(d+1)\vec{z} - \vec{x}\|^2 \\
&= \sum_{k=0}^{d} (d+1)^2 (z_k'^2 - z_k^2) - 2(d+1)x_k(z_k' - z_k) \\
&= (d+1)\sum_{k=0}^{d} \left[(d+1)(z_k' + z_k) - 2x_k\right](z_k' - z_k) \\
&= (d+1)\left[((d+1)(z_i' + z_i) - 2x_i) - ((d+1)(z_j' + z_j) - 2x_j)\right] \\
&= (d+1)\left[((d+1)(2z_i + 1) - 2x_i) - ((d+1)(2z_j - 1) - 2x_j)\right] \\
&= 2(d+1)\left[(d+1)(1 + z_i - z_j) - (x_i - x_j)\right]
\end{aligned}$$

Dividing both sides of the last inequality by $2(d+1)$ and rearranging the terms, we obtain

$$x_i - x_j \leq (d+1)(1 + z_i - z_j). \tag{3.1}$$

(1⇒2) Condition (1) implies $\vec{z} = \vec{0}$. Then (3.1) becomes,

$$x_i - x_j \leq d + 1.$$

Since this holds for all i, j, we obtain $\max_i x_i - \min_i x_i \leq d + 1$ as desired.

(2⇒1) Condition (2) implies that for all i, j,

$$-(d+1) \leq x_i - x_j$$

Combined with (3.1), this implies

$$\begin{aligned}
&-(d+1) \leq (d+1)(1 + z_i - z_j) \\
&\Rightarrow z_j - z_i \leq 2
\end{aligned}$$

Because $(d+1)\vec{z} \in H_d$, the components of $\vec{z}$ must sum to zero, so if there are any

strictly positive components there must also be at least one strictly negative component. This combined with $z_j - z_i \leq 2$ implies each component of $\vec{z}$ is -1, 0 or 1.

Suppose nonzero components exist, i.e. $z_i = -1, z_j = 1$. For these particular values of i, j, (3.1) must hold as equality, meaning that $\vec{z'}$ and $\vec{z}$ are equidistant from $\vec{x}$. Thus we can continue adding 1 to a negative component and -1 to a positive component, until we reach the origin, without altering the distance to $\vec{x}$. So the origin must be the closest remainder-0 point to $\vec{x}$, or at least tied for the closest. □

Lattices are translation invariant, so in general the above proposition tells us that if the closest remainder-0 point to some point $\vec{x} \in H_d$ is $\vec{l}$, then $\vec{x}$ belongs to the union of the simplices that touch $\vec{l}$. Which simplex in particular is given by the ordering of the coordinates of $\vec{x} - \vec{l}$ (Figure 3.2). Thus every point belongs to a unique simplex, which is a permutation and translation of the canonical simplex, so H_d is tessellated by uniform simplices.

3.2.2 The vertices of the simplex enclosing any point can be computed in $O(d^2)$ time

The vertices of the simplex containing some point $\vec{x} \in H_d$ can be generated by first computing the closest remainder-0 point $\vec{l_0}$, and then sorting the difference $\vec{l_0} - \vec{x}$. This takes $O(d \log d)$ operations. The inverse of the resulting permutation and translation can then be applied to the canonical simplex to compute the simplex vertices in $O(d^2)$ operations. This property will be useful for the splat and slice stages.

The closest remainder-0 point can be found by first rounding each coordinate of $\vec{x}$ to the nearest multiple of $(d+1)$, and then, if the result is outside the subspace H_d, greedily walking back to H_d by identifying the coordinates that moved the farthest, and rounding them in the other direction instead. The sub-lattice formed by the remainder-0 points is called A_{d+1}, and this is the algorithm given by Conway and Sloane ([17] pp 446) for finding the closest point in that lattice.

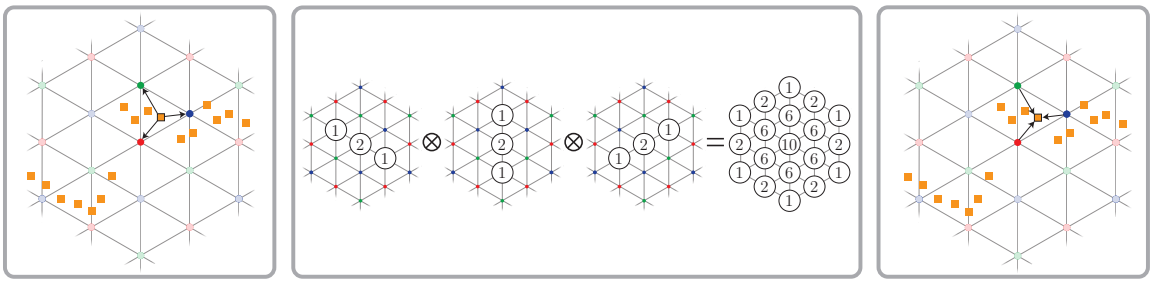

Figure 3.3: To perform a Gauss transform using the permutohedral lattice, first the position vectors $\vec{p}_i \in \mathbb{R}^d$ are embedded in the hyperplane H_d using an orthogonal basis for H_d (not pictured). Then, each input value **splats** onto the vertices of its enclosing simplex using barycentric weights. Next, lattice points **blur** their values with nearby lattice points using a separable filter. Finally, the space is **sliced** at each input position using the same barycentric weights to interpolate output values.

3.2.3 The nearest neighbors of a lattice point can be found in $O(d^2)$ time

The basis vectors given by B_d above are those of minimal length, so the nearest neighbors of a lattice point $\vec{l}_k$ are those separated by a vector of the form $\pm[-1, \ldots, -1, d, -1, \ldots, -1]$. The are $2(d+1)$ such neighbors, and each is described by a vector of length $d+1$, and so the neighbors can be fully enumerated in $O(d^2)$ time. This property will be useful during the blur stage.

3.3 Computing Gauss transforms using the lattice

There are four main stages in using the permutohedral lattice for fast Gauss transforms, illustrated in Figure 3.3. We describe each in turn.

3.3.1 Generating position vectors

First, the position vectors $\vec{p}_i$ must be generated and embedded in H_d. Generating the positions is application dependent (see Chapter 1). For a color bilateral filter,

we generate 5-D position vectors of the form $[\frac{x_i}{\sigma_p}, \frac{y_i}{\sigma_p}, \frac{r_i}{\sigma_c}, \frac{g_i}{\sigma_c}, \frac{b_i}{\sigma_c}]$, by augmenting the input image with two extra channels encoding spatial location, and then scaling each channel by the inverse of the desired standard deviation. For non-local means we would instead either extract local windows around each pixel, or compute some bank of filters around each pixel and record the responses. Typically we do the latter, using PCA to determine the optimal filter bank, as proposed by Tasdizen [43].

We must then scale the position vectors by the inverse of the standard deviation of the blur induced by the remaining steps, which totals $\sqrt{\frac{2}{3}}(d+1)$ in each dimension (derived below). Next we embed the position vectors in the subspace H_d. The basis for H_d given above is unsuitable for this task because it is not orthogonal, so we instead use the orthogonal basis:

$$E = \begin{pmatrix} 1 & 1 & \dots & 1 \\ -1 & 1 & \dots & 1 \\ 0 & -2 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -d \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \dots & 0 \\ 0 & \frac{1}{\sqrt{6}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{\sqrt{d(d+1)}} \end{pmatrix}$$

We choose this basis because it allows us to compute $E\vec{x}$ in O(d) time using the recurrence:

$$\begin{aligned} (E\vec{x})_d &= -\alpha_d x_{d-1} \\ (E\vec{x})_i &= -\alpha_i x_{i-1} + x_i/\alpha_{i+1} + (E\vec{x})_{i+1} \\ (E\vec{x})_0 &= x_0/\alpha_1 + (E\vec{x})_1 \\ where\ \alpha_i &= \sqrt{i/(i+1)} \end{aligned}$$

3.3.2 Splatting

Once each position has been embedded in the hyperplane, we must identify its enclosing simplex and compute barycentric weights. The enclosing simplex of any point

can be described by the permutation and translation that maps the simplex back to the canonical simplex, which can be computed in $O(d \log d)$ by using the rounding algorithm described earlier to find the nearest remainder-0 point, and then sorting the residual.

Therefore, to compute barycentric coordinates for a point $E\vec{p}_i$ in an arbitrary simplex, we can apply the translation and permutation to map $E\vec{p}_i$ to some $\vec{y}$ within the canonical simplex. Barycentric coordinates $\vec{b}$ for $\vec{y}$ are then given by the following:

Proposition 3.3.1. *Let $\vec{y}$ be an arbitrary point inside the canonical simplex, and let $b_0, \ldots, b_d$ be its unique barycentric coordinates in the simplex, i.e.*

$$\vec{y} = \sum_{k=0}^{d} b_k \vec{s}_k \quad \text{and} \quad \sum_{k=0}^{d} b_k = 1$$

then,

$$b_k = \begin{cases} \dfrac{y_{d-k} - y_{d+1-k}}{d+1}, & k \neq 0, \\ 1 - \dfrac{y_0 - y_d}{d+1}, & k = 0. \end{cases}$$

Proof. $\sum_{k=0}^{d} b_k = 1$ is clearly true, thus it suffices to show that $\vec{y} = \sum_{k=0}^{d} b_k \vec{s}_k$:

$$\begin{aligned}
&\forall j \left[\sum_{k=0}^{d} b_k \vec{s}_k \right]_j \\
&= \left[\sum_{k=0}^{d-j} b_k k \right] + \left[\sum_{k=d-j+1}^{d} b_k (k - (d+1)) \right] \\
&= \left[\sum_{k=0}^{d} b_k k \right] - \left[(d+1) \sum_{k=d-j+1}^{d} b_k \right] \\
&= \left[\left(\frac{y_{d-1} - y_d}{d+1} \right) + 2 \left(\frac{y_{d-2} - y_{d-1}}{d+1} \right) + \cdots + d \left(\frac{y_0 - y_1}{d+1} \right) \right] - \left[\sum_{k=d-j+1}^{d} y_{d-k} - y_{d+1-k} \right] \\
&= \frac{-y_d - y_{d-1} - \cdots - y_1 + d y_0}{d+1} - (y_0 - y_j)
\end{aligned}$$

$$= \frac{-y_d - y_{d-1} - \cdots - y_1 - y_0}{d+1} + y_j$$
$$= y_j,$$

as required. □

Barycentric interpolation is invariant to translation and commutes with permutation, and so these barycentric coordinates for $\vec{y}$ within the canonical simplex are also the (permuted) barycentric coordinates for $E\vec{p}_i$ within its simplex. Once the barycentric weights are computed, $b_k\vec{v}_i$ is added to the value stored at the remainder-k lattice point in the enclosing simplex of $\vec{p}_i$ (recall that $\vec{v}_i$ is the homogeneous value associated with position $\vec{p}_i$). The lattice point values are stored in a hash table. Lattice points that do not yet exist in the hash table are created when they are first referred to during the splat stage, and start with an initial value of zero.

There are two ways to identify each lattice point for use as a hash table key. One can apply the inverse permutation and translation to the remainder-k point of the canonical simplex to compute the lattice point's position, and use that as a key. Each key is a vector of length $d+1$, and so this results in a memory complexity of $O(dl)$ for l lattice points.

In rare cases where $l > n$, we can alternatively achieve a memory complexity of $O(dn)$ for n input values by separately storing the simplex enclosing each input position $\vec{p}_i$, as a simplex can be identified uniquely in $O(d)$ memory by its remainder-0 point and its permutation. We then identify a lattice point using its remainder and a pointer to any simplex it belongs to, for an additional $O(dn)$ memory. One lattice point belongs to many simplices, so key comparison is done by using the simplex and remainder to compute the lattice point's coordinates on the fly.

l is loosely bounded by $O(dn)$, as each input value creates at most $d+1$ new lattice points. However, filters near this bound correspond to very small filter sizes and are not very useful, as no shared lattice points means very little cross-talk between pixels, and hence very little filtering. In practice, we find that l is less than n, so we prefer the first, faster scheme, which has worst-case memory complexity of $O(d^2n)$. In either case, each hash table access costs $O(d)$ time for key comparison. Splatting each input

pixel accesses the hash table $O(d)$ times, and so the time complexity of splatting is $O(d^2 n)$.

Barycentric interpolation in the permutohedral lattice is equivalent to convolution by the projection of a uniformly-weighted $(d+1)$-dimensional hypercube of side length $d+1$ onto H_d, and the variance of the resulting kernel is $d(d+1)^2/12$, as shown by the following proposition.

Proposition 3.3.2. *The variance of the splatting kernel is* $\dfrac{d(d+1)^2}{12}$.

Proof. A lattice is translation invariant, so without loss of generality, we can compute the splatting kernel by considering the barycentric weight b_0 given to the lattice point at the origin, for a query position $\vec{x}$ whose closest remainder-0 point is the origin. Recall that first we sort the coordinates of $\vec{x}$ into decreasing order to obtain $\vec{y}$, and barycentric weights are then given by:

$$\begin{aligned} b_0 &= 1 - \frac{y_0 - y_d}{d+1} \\ \Rightarrow b_0 &= 1 - \frac{\max_i x_i - \min_i x_i}{d+1} \end{aligned}$$

Now consider taking an integral projection of the uniformly-weighted hypercube $[0, d+1]^{d+1}$ onto H_d. For each $\vec{x} \in H_d$, the points which project onto $\vec{x}$ have the form $\vec{x} + k\vec{1}$. Since $\vec{x} + k\vec{1}$ must fall in the hypercube, we have:

$$\begin{array}{rlrcrcl} & \forall i & 0 & \leq & x_i + k & \leq & d+1 \\ \Rightarrow & \forall i & -x_i & \leq & k & \leq & d+1-x_i \\ \Rightarrow & & -\min_i x_i & \leq & k & \leq & d+1-\max_i x_i \end{array}$$

This indicates that the mass of points that are projected onto $\vec{x}$ is proportional to $d+1-(\max_i x_i - \min_i x_i)$, which in turn is proportional to the splatting kernel given by b_0 above. Therefore, the variance of the splatting kernel equals that of the projected uniformly-weighted hypercube. We can compute this by integrating the second moment of the projected point over the hypercube:

$$
\begin{aligned}
\text{Variance} &= \frac{\int_{[0,d+1]^{d+1}} \left\| \vec{y} - \frac{\sum_j y_j}{d+1}\vec{1} \right\|^2 d\vec{y}}{\int_{[0,d+1]^{d+1}} d\vec{y}} \\
&= \frac{1}{(d+1)^{d+1}} \int_{[0,d+1]^{d+1}} \sum_i \left(y_i - \frac{\sum_j y_j}{d+1} \right)^2 d\vec{y} \\
&= \frac{1}{(d+1)^{d+1}} \int_{[0,d+1]^{d+1}} \sum_i \left(y_i^2 + \left(\frac{\sum_j y_j}{d+1} \right)^2 - 2\frac{\sum_j y_i y_j}{d+1} \right) d\vec{y} \\
&= \frac{1}{(d+1)^{d+1}} \int_{[0,d+1]^{d+1}} \left(\sum_i y_i^2 \right) + \frac{\left(\sum_j y_j\right)^2}{d+1} - 2\frac{\sum_{i,j} y_i y_j}{d+1} \; d\vec{y} \\
&= \frac{1}{(d+1)^{d+2}} \int_{[0,d+1]^{d+1}} (d+1) \sum_i y_i^2 - \sum_{i,j} y_i y_j \; d\vec{y} \\
&= \frac{1}{(d+1)^{d+2}} \int_{[0,d+1]^{d+1}} d \sum_i y_i^2 - \sum_{i \neq j} y_i y_j \; d\vec{y} \\
&= \frac{1}{(d+1)^{d+2}} \left[\sum_i d \int_{[0,d+1]^{d+1}} y_i^2 \; d\vec{y} \right] - \left[\sum_{i \neq j} \int_{[0,d+1]^{d+1}} y_i y_j \; d\vec{y} \right] \\
&= \frac{1}{(d+1)^{d+2}} \left[\sum_i d(d+1)^d \int_0^{d+1} y_i^2 \; dy_i \right] - \left[\sum_{i \neq j} (d+1)^{d-1} \int_0^{d+1} \int_0^{d+1} y_i y_j \; dy_i \; dy_j \right] \\
&= \frac{1}{(d+1)^{d+2}} \left[d(d+1)^{d+1} \int_0^{d+1} t^2 dt \right] - \left[d(d+1)^d \int_0^{d+1} \int_0^{d+1} st \; ds \; dt \right] \\
&= \frac{1}{(d+1)^{d+2}} \left[d(d+1)^{d+1} \frac{(d+1)^3}{3} \right] - \left[d(d+1)^d \left(\frac{(d+1)^2}{2} \right)^2 \right] \\
&= \frac{d(d+1)^2}{3} - \frac{d(d+1)^2}{4} \\
&= \frac{d(d+1)^2}{12}
\end{aligned}
$$

□

3.3.3 Blurring

Now that we have resampled our input onto the lattice, we perform the next step of the Gauss transform by blurring along the lattice. To do this we convolve by the kernel $[1,\ 2,\ 1]$ along each lattice direction of the form $\pm[1,\ldots,1,-d,1,\ldots,1]$ (Figure 3.3). Each such convolution has a variance of $d(d+1)/2$, so the combined effect of the $d+1$ convolutions is an approximate Gaussian kernel with total variance $d(d+1)^2/2$. Note that we are ignoring scale in our choice of kernel. We have this luxury because we always filter homogeneous values, which are scale-invariant.

The blur stage spreads energy from each lattice point to $O(3^d)$ neighbors. If we created hash table entries for new lattice points reached during the blur then the memory use would grow quite large. We therefore do not create new lattice points during the blur phase, which incurs some accuracy penalty relative to a naive Gauss transform, as points that may have transferred energy could instead belong to disconnected regions of the lattice.

This "error" may actually be advantageous depending on the application. For example, when bilateral filtering, the absence of these "stepping-stone" lattice points will prevent energy transfer from a white pixel to a black pixel across a hard edge, but will allow energy transfer between a black pixel and a white pixel on either side of a smooth gradient.

The blur step involves looking up $O(d)$ neighbors for each lattice point. Each lookup takes $O(d)$ time for hash table key comparison, and so the blur step has time complexity $O(d^2l)$. In the worst case, this expression is bounded by $O(d^3n)$. However, let us consider the extreme cases. If the splat positions are spaced very densely, we expect $l < n$, and so the time complexity of blurring is $O(d^2n)$.

If the positions are very sparse, then each input position creates its own simplex, far from any other. In this case, for each lattice point, all but 2 hash-table lookups fail during blurring. By inspecting the axes used to blur, we can see that for a point of remainder r, we look up $2(d+1)$ adjacent points of remainders $r+1$ and $r-1$. For a lone simplex, there is only one point of each remainder. Therefore in the sparse case each of our dn lattice points performs $O(1)$ successful hash-table lookups and $O(d)$ failed hash table lookups. A failed hash-table lookup will incur $O(1)$ failed key

comparisons before it detects failure for a well-spaced hash table. Each failed key comparison detects failure after $O(1)$ coordinates are compared, so the total time complexity is again $O(d^2n)$.

So for small filters (resulting in a very sparse cloud of position vectors) our runtime is bounded by $O(d^2n)$ because most hash-table lookups fail cheaply. For large filters (resulting in a dense cloud of position vectors) we are similarly bounded by $O(d^2n)$ because few lattice points are created. It is possible to construct pathological cases in between. If these occurred in practice we would see a runtime that was non-monotonic with respect to filter size. We have never observed this — the time taken by blurring always decreases as filter size increases and the total number of lattice points decreases.

The absence of these pathological cases in practice is due to the fact that the position vectors usually lie on some lower-dimensional manifold in position-space, (typically 2-dimensional if the input data comes from an image) and so the probing in $d+1$ different directions done by blurring reaches into empty space for all but a few of those directions. That is, the density argument applies for directions that lie along the manifold, and the sparsity argument applies for all other directions.

3.3.4 Slicing

Slicing is identical to splatting, except that it uses the barycentric weights to gather from the lattice points instead of scattering to them. It produces the same total variance of $d(d+1)^2/12$, which brings the total variance induced by the algorithm to $\frac{2}{3}d(d+1)^2$, which is equivalent to a standard deviation in each dimension of $\sqrt{\frac{2}{3}}(d+1)$. Slicing can be accelerated by storing the barycentric weights and pointers to lattice point values computed during splatting. This "slicing table" is a sparse matrix representation of slicing, which is linear in the values, and is the transpose of splatting. It can be scanned through in $O(dn)$ time to slice. The entire algorithm thus has a time complexity of $O(d^2(n+l))$.

3.4 Implementation

To recap, performing a Gauss transform using the permutohedral lattice can be broken into the following steps:

1. Splat. For each position vector $\vec{p}_i$ and value $\vec{v}_i$:
   (a) Elevate $\vec{p}_i$ into H_d using the rotation matrix E.
   (b) Compute the lattice points $\vec{l}_j$ of the simplex enclosing $E\vec{p}_i$, and the corresponding barycentric weights b_j.
   (c) For each $\vec{l}_j$:
      i. Look up its value $\vec{u}_j$ in the hash table, creating a new zero-valued entry if it does not already exist.
      ii. $\vec{u}_j \leftarrow \vec{u}_j + b_j\vec{v}_i$.
      iii. Store b_j and a reference to $\vec{u}_j$ in the slicing table at location (i, j) so we need not look them up again during slicing.
2. Blur. For each dimension j:
   (a) For each lattice point $\vec{l}_i$ with value $\vec{u}_i$:
      i. Look up the values $\vec{u}_{j-}$, $\vec{u}_{j+}$ stored at the two neighbors of $\vec{l}_i$: $\vec{l}_i \pm [\underbrace{-1, \ldots, -1}_{j}, d, \underbrace{-1, \ldots, -1}_{d-j}]$.
      ii. Compute a new value for this lattice point: $\vec{\hat{u}}_i = \vec{u}_{j-} + 2\vec{u}_i + \vec{u}_{j+}$.
   (b) Assign to each lattice point its updated value: $\vec{u}_i \leftarrow \vec{\hat{u}}_i$.
3. Slice. For each position vector $\vec{p}_i$ compute the output value $\vec{\hat{v}}_i$:
   (a) $\vec{\hat{v}}_i \leftarrow \vec{0}$.
   (b) For each dimension j:
      i. Look up the lattice value $\vec{u}$ and barycentric weight b in the slicing table at location (i, j).
      ii. $\vec{\hat{v}}_i \leftarrow \vec{\hat{v}}_i + b\vec{u}$.

3.4.1 Efficient CPU implementation

A straightforward implementation of the lattice can be found in Appendix A. There are several ways to optimize this algorithm for modern CPUs. First, the dimensionality of the value vectors is often four (red, green, blue, and a homogeneous weight). This makes four-wide SIMD floating-point units such as SSE well-suited to the arithmetic which deals with summing values during splatting, blurring, and slicing (Steps 1(c)ii, 2(a)ii, and 3(b)ii). The arithmetic which computes nearby lattice points (1b) involves sorting, branching, and some iterative algorithms, and so is much harder to parallelize in this way.

The algorithm is fairly simple to run in parallel across multiple CPU cores. Splatting and slicing are data-parallel across input positions, and blurring is data-parallel across lattice vertices. Slicing and blurring are trivial to parallelize. However, two issues arise during splatting. First, contention on hash table buckets may occur when newly-found lattice vertices are being inserted into the hash table (1(c)i). This is easily solved by attaching a lock to each bucket. Second, the additions performed while accumulating values at each lattice vertex (1(c)ii) must be atomic. The performance of the resulting code scales fairly well with available hardware resources (see Figure 3.4).

3.4.2 Efficient GPU implementation

The algorithm is also fairly straightforward to parallelize on a GPU. We constructed an implementation using NVIDIA's CUDA [14], and achieve typical speedups of 9× on a GeForce GTX 465 compared to the single-threaded CPU implementation on an Intel Core i7 950. Note that this is only 2× faster than the multi-threaded CPU implementation.

The main point of difference between the CPU and GPU versions is related to the creation of hash table entries during splatting. In the CPU version we attach locks to hash table entries and synchronize all accesses to a given entry to prevent erroneously inserting one key in multiple places. In the GPU version it is faster to break the splatting stage into three. First, we compute the slicing table, recording

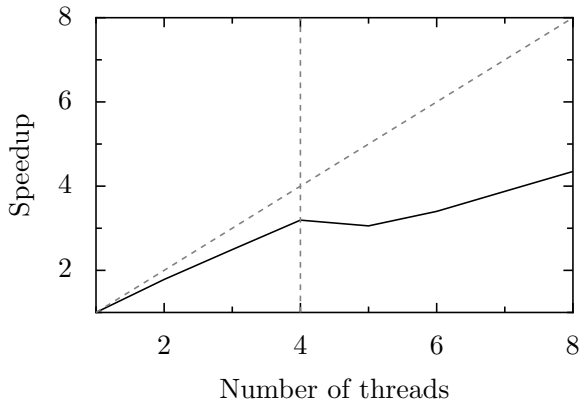

Figure 3.4: This graph shows the speedup attained by parallelizing the permutohedral lattice on an Intel Core i7 950 CPU. This CPU has four physical cores that each run two hardware threads. The splatting and slicing stages of the algorithm are data-parallel across pixels, and the blurring stage is data-parallel across lattice vertices. Splatting and blurring both rely on hash table lookups to query the values stored at lattice points. This is fairly cache-incoherent, and so the extra hardware threads are necessary to hide the memory latency and achieve the full 4x speedup.

which lattice points each input pixel splats to, and with what weights. While doing this, we insert the lattice points found into the hash table in a way which permits individual keys being inserted in multiple locations. Specifically, while we still lock each hash table entry before insertion, other simultaneous hash table insertions simply skip over locked entries while looking for a free spot rather than waiting on the lock to see if the key matches. This means we never have data dependencies involving one query reading the key that another query has written, so we can write the keys using faster non-atomic writes, and only the smaller array of locks needs to be coherent. Next, we rehash the entries of the slicing table and update it so that every reference to a lattice point refers to the unique earliest instance of that lattice point in the hash table. Finally, we use the corrected slicing table to splat, additively scattering onto lattice points as usual.

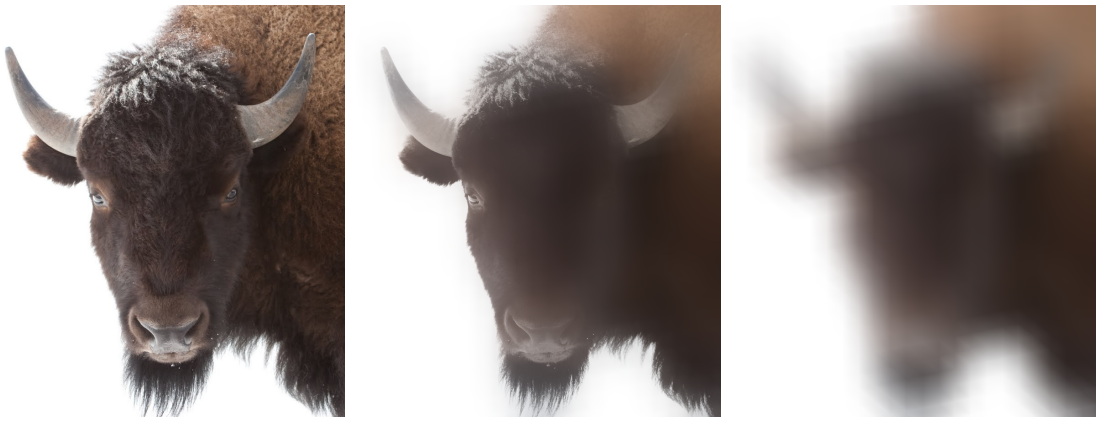

Figure 3.5: The permutohedral lattice generates output value using linear interpolation in the high-dimensional space of position vectors. This does not pose problems for typical uses of the Gauss transform, such as the aggressive color bilateral filter (center) of this input image of a bison (left), as any artifacts are obscured by the mapping back down from the space of position vectors to the output image. If we use the permutohedral lattice to perform a conventional 2D Gaussian blur (right), the effect of the linear interpolation is readily apparent, and is objectionable.

3.5 Conclusion

The permutohedral lattice offers a simple, fast method for computing approximate Gauss transforms. It works especially well for problems with dimensionalities between 5 and 8, which encompasses color bilateral filters, low-dimensional non-local means, and other related filters.

The main source of inaccuracy for the permutohedral lattice is that it computes the output using barycentric interpolation, which may produce a piecewise-linear output image. For typical image processing problems this effect is not visible in the output, as the mapping back from high-dimensional position space to output pixels obscures any resulting artifacts. It does, however, make the permutohedral lattice inappropriate for any filtering tasks that might devolve into a simple Gaussian blur for significant regions of the input; in this case the mapping from position-space to output pixels is direct, and artifacts are preserved (Figure 3.5).

Performing a Gauss transform using the permutohedral lattice constructs a simplicial scaffold around the input, using an amount of memory that scales quadratically

in the dimension of the underlying space. Performance is acceptable up to around 12 dimensions; beyond this, the runtime and memory use become objectionable. The following chapter describes the Gaussian kd-tree, which is slower than the lattice for low-dimensional cases, but scales linearly with dimensionality in time and memory.

Chapter 4

The Gaussian KD-Tree

The Gaussian kd-tree groups the input positions into clusters stored at the leaves of a kd-tree, and places one sample at the center of each cluster. We can then factor a Gauss transform into two stages: First we *splat* by scattering the input values onto the clusters nearby to each input position. Then, we *slice* by gathering values from the clusters nearby each output position.

Fixing the positions, a Gauss transform is a dense linear transform in the values. The Gaussian kd-tree acts as a low-rank factorization of this transform, with the values stored at the clusters acting as the intermediate space.

Both splatting and slicing use the same Monte-Carlo algorithm based on importance sampling. Given a query position, the algorithm returns a short list of randomly-chosen cluster centers. The probability of a cluster center belonging to the list is roughly proportional to a Gaussian centered around the query position evaluated at the cluster center. Because the proportionality is only approximate, we also return a weight with each cluster center to correct for any bias introduced.

The Gaussian kd-tree allows us to perform Gauss transforms of n input values in d dimensions with a time complexity of $O(dn \log n)$, and a memory complexity of $O(dn)$.

4.1 Constructing a Gaussian kd-tree

The Gaussian kd-tree stores a cloud of m clusters in d dimensions, one cluster per leaf. Each node of the tree η represents some d-dimensional rectangular cell, which may extend to infinity in one or more dimensions. Inner nodes subdivide this cell into two child cells, separated by an axis-aligned cut. An inner node therefore stores a dimension η_d along which it cuts, a value η_{cut} at which to cut, the bounds of the node in that dimension η_{min} and η_{max}, and pointers to its children. Leaf nodes contain only a d dimensional point, which lies somewhere within the cell it represents. The only difference between this tree and a conventional kd-tree is that we store the bounds of a cell (η_{min} and η_{max}) as well as the value at which it cuts (η_{cut}). η_{max} is computed as the minimum η_{cut} of all ancestors which cut along the same dimension and have a larger η_{cut}. η_{min} is similarly the maximum η_{cut} of all ancestors which cut along the same dimension and have a smaller η_{cut}.

The goal when building a kd-tree is usually to minimize the expected time taken by a query. In ray tracing, for example, this means it can be advantageous to have a highly unbalanced tree which carves off empty space and commonly hit areas early. However we typically slice at the same positions as we splat, which means we never sample in unpopulated areas, so how we deal with empty space is irrelevant. Furthermore, for typical data each of our leaf nodes is as likely to be reached as any other, so the tree should be balanced.

To recursively turn a list of positions $\vec{p}_i$ into a tree, we first compute their bounding box. If the bounding box has maximum side length length less than some threshold ρ we create a leaf node, and an associated cluster center at the center of the bounding box. Otherwise, we split halfway along the longest bounding box dimension, partition the input list into two over the split, and continue recursively. This scheme descends to cells that have a small maximum side-length as quickly as possible. See Figure 4.1 for an illustration of this tree building algorithm.

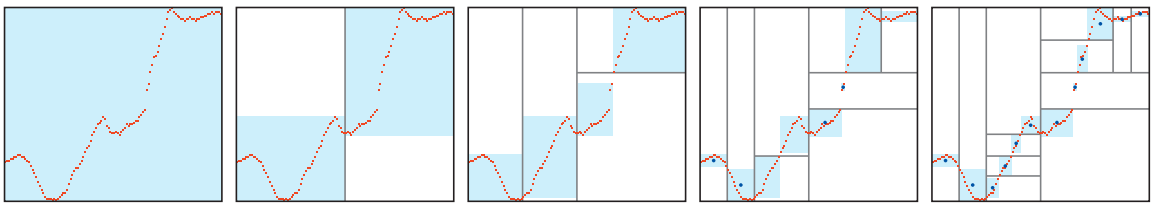

Figure 4.1: The Gaussian kd-tree groups the input data (in orange) into clusters at the leaves of a kd-tree. We build a Gaussian kd-tree by computing the bounding box of the input (in light blue), splitting it halfway along its longest edge, partitioning the input over the split, and continuing recursively on the children until the maximum edge length of the bounding box drops below some threshold ρ. In the finished tree on the right, each cluster stores a single sample at its center (the blue points).

4.2 Splatting and slicing

A query into the Gaussian kd-tree is designed to facilitate gathers from (or scatters to) values around a given query position, for the purpose of computing an importance-sampled approximation of a Gauss transform. Figure 4.2 illustrates the process. A query takes as input a query position $\vec{q}$ in the space, a standard deviation σ around that position, and a number of query samples s, and returns a list of at most s cluster centers $\vec{p}_i$ with corresponding weights w_i. If the number of query samples is set to infinity, the list returned will include all points within a fixed radius about the query, with weights proportional to a Gaussian kernel of the given standard deviation ($w_i = e^{-|\vec{q}-\vec{p}_i|^2/2\sigma^2}$). If the number of query samples is set to one, the list will contain a single cluster center, probabilistically chosen from all nearby cluster centers, such that repeatedly asking for a single sample and merging the resulting lists will produce the same result in the limit as asking for an infinite number of samples from a single query.

We can think of our query samples as a cloud of points normally distributed around the query position with the given standard deviation, although we do not explicitly represent them as such. At each inner node η we compute the expected number of query samples that lie within the left and right child by computing the area of the Gaussian, truncated to within η_{min} and η_{max}, that lies on either side of

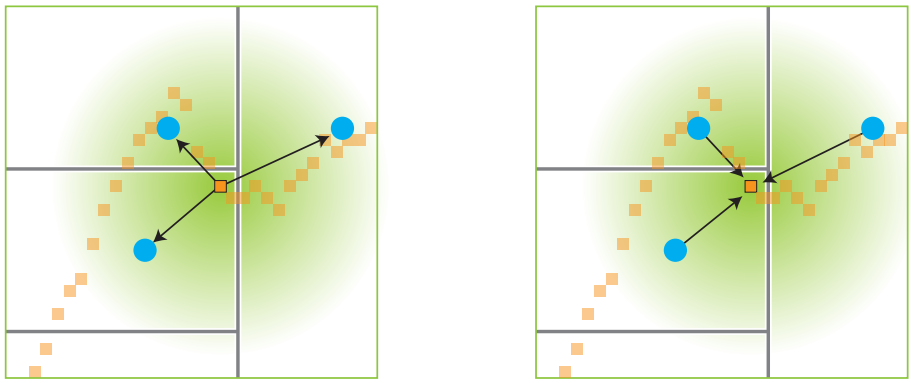

Figure 4.2: The Gaussian kd-tree groups the input data (in orange) into clusters at the leaves of a kd-tree. Each cluster stores a single sample at its center (the blue points). Querying the tree is done by simulating the descent of a Gaussian cloud of samples (in green) down to the leaves. The expected number of samples that arrive at a leaf node is proportional to the Gaussian integrated over that node, which is approximately proportional to the Gaussian evaluated at the (blue) cluster centers. Each query returns a weighted list of the leaf nodes reached, with the weight given by the number of samples that reached that node multiplied by a term that corrects for the approximation involved. The same type of query is used to first scatter data to each leaf node (splatting), and then gather data from them (slicing).

η_{cut}. The Gaussian is separable, so decisions already made by nodes that split in other dimensions are irrelevant; this is why we use a tree that makes only axis-aligned cuts. The expected number of samples that split each way are rounded down to the nearest integer, and that many samples are assigned to the left or right child respectively. The final sample omitted by the rounding, if there is one, is probabilistically assigned to either the left or the right child.

This splitting scheme is cheaper than creating an explicit cloud of query samples and individually simulating the descent of each. The runtime is sub-linear in the number of query samples, and bounded by the number of cells overlapping a query. It also stratifies the sampling, avoiding cases in which a large number of query samples descend to the same leaf node by chance. This results in less noise in the output for a given number of query samples.

We arrive at a given leaf node with a probability proportional to the integral of the Gaussian over the corresponding cell. This is not the correct probability, however, as our tree does not store values at cells; it stores values at the cluster center within each cell. To correct for this, during our descent of the tree we keep track of the probability of one sample reaching each node using our approximation. At the leaf we compute the correct probability of reaching that cluster center by evaluating the Gaussian at it. The latter divided by the former gives us a corrective weight to apply to each sample. See Algorithm 4.1 for a C++ implementation. This algorithm is essentially *weighted* importance sampling, as described by [8] in the context of radiosity. This correction allows us to use a piecewise cubic approximation to the Gaussian (given by the convolution of four identical rectangular filters) while descending the tree:

$$g(x) = \begin{cases} 0 & : \ x \leq -2 \\ (2+x)^3 & : \ -2 < x \leq -1 \\ -3x^3 - 6x^2 + 4 & : \ -1 < x \leq 2 \\ 3x^3 - 6x^2 + 4 & : \ 0 < x \leq 1 \\ (2-x)^3 & : \ 1 < x \leq 2 \\ 0 & : \ 2 < x \end{cases}$$

This function closely approximates a scaled Gaussian with variance $\frac{1}{3}$. x can be rescaled to achieve a Gaussian of any desired variance. Its (scaled) integral is significantly cheaper to evaluate than that of a true Gaussian:

$$G(x) = \begin{cases} 0 & : \ x \leq -2 \\ (2+x)^4 & : \ -2 < x \leq -1 \\ -3x^4 - 8x^3 + 16x + 12 & : \ -1 < x \leq 2 \\ 3x^3 - 8x^3 + 16x + 12 & : \ 0 < x \leq 1 \\ 24 - (2-x)^4 & : \ 1 < x \leq 2 \\ 24 & : \ 2 < x \end{cases}$$

To splat, we query around each input position with $\sigma = \frac{\sqrt{2}}{2}$ and additively scatter to the returned clusters with the returned weights. To slice, we perform an identical

Algorithm 4.1 Looking up samples in a Gaussian kd-tree.

```
// A quartic approximation to the integral of a Gaussian of standard deviation one.
float cdfApprox(float x);

// A uniform random float between zero and one.
float urand();

// The Euclidean distance between two points.
float distance(vector<float> a, vector<float> b);

class InnerNode :  public Node {
    int d;
    float min, max, cut;
    Node *left, *right;

    void Query(vector<float> q, float sigma, int samples, vector<Result> &results, float p=1) {

        float cdfMin = cdfApprox((min - q[d])/sigma);
        float cdfMax = cdfApprox((max - q[d])/sigma);
        float cdfCut = cdfApprox((cut - q[d])/sigma);
        float pLeft = (cdfCut - cdfMin)/(cdfMax - cdfMin);

        float expectedSamplesLeft = pLeft*samples;
        int samplesLeft = floor(expectedSamplesLeft);
        int samplesRight = floor(samples - expectedSamplesLeft);

        if (samplesLeft + samplesRight < samples) {
            if (urand() < expectedLeft - samplesLeft)
                samplesLeft++;
            else
                samplesRight++;
        }

        if (samplesLeft > 0)
            left->Query(q, sigma, samplesLeft, results, p*pLeft);

        if (samplesRight > 0)
            right->Query(q, sigma, samplesRight, results, p*(1-pLeft));

    }
};

class LeafNode :  public Node {
    vector<float> position, value;

    void Query(vector<float> q, float sigma, int samples, vector<Result> &results, float p) {
        float d = distance(q, position);
        float correctP = exp(-d*d/(2*sigma));
        results.push_back(Result(&value, samples*correctP/p));
    }
};
```

query but instead additively gather from the returned clusters with the returned weights. The combination of the two stages effects an approximate Gauss transform of variance one.

Note that even if we use an infinite number of samples, the Gaussian kd-tree does not compute an exact Gauss transform of variance one. The splatting step acts as one discrete Gauss transform from the input positions to the cluster positions, and the slicing step performs a second discrete Gauss transform from the cluster positions to the output positions. These two discrete transforms each have a variance of one half, but they do not compose into a single larger Gauss transform as a continuous Gauss transform does.

We could use the Gaussian kd-tree to perform a single large Gauss transform by setting ρ to zero (not clustering), and then either skipping splatting or slicing. However, we would then lose the speed benefits of the low-rank approximation that the clustering provides.

Despite this difference, the result of two successive Gauss transform has all the same desirable properties for our applications as a single larger Gauss transform (and may be superior in some cases). Therefore, the best way to evaluate parameter choices for the Gaussian kd-tree is to compare the output to that produced by a pair of naive Gauss transforms with variance one half. This gives us a clearer picture of what we sacrifice by using larger clusters or smaller sample counts, as it disambiguates the error due to sampling and clustering from the difference between one large discrete Gauss transform and the composition of two small discrete Gauss transforms.

4.3 Parameter selection

There are three parameters of the Gaussian kd-tree that trade off between performance and accuracy:

1. The number of samples used when splatting
2. The number of samples used when slicing

3. The threshold size ρ at which a cluster of input positions is treated as a single leaf node

The parameter choices that reliably give good accuracy and performance are 4 splatting samples, 64 slicing samples, and $\rho = \sqrt{2}$. We can visualize this by fixing two of the parameters and varying the third to show that these choices are reasonable (Figures 4.3, 4.4, and 4.5). These choices are surprising in two ways.

First, far more slicing samples than splatting samples are used. Recall, however, that splatting maps from a large number of input pixels to a smaller number of cluster centers. This averaging down reduces the required sample count. Furthermore, noisy values at the cluster centers may not have a visible effect on the output, or may appear as hard-to-detect low frequency variations. In contrast, the noise produced by a small number of slicing samples is visible in the output as high-frequency image noise, which is extremely apparent in the output of an algorithm designed to smooth or denoise.

Second, the choice of $\rho = \sqrt{2}$ is quite large. Recall that splatting and slicing both effect Gauss transforms of standard deviation $\frac{\sqrt{2}}{2}$. We are thus sampling once every two standard deviations of our Gaussian filters. This value was chosen because using fewer clusters makes the algorithm faster, and also because a smaller number of clusters can actually result in a *more* accurate output, as the cluster values are less noisy for a fixed number of splatting samples (see Figure 4.5).

4.4 Implementation

Performing a Gauss transform of variance 1 with the Gaussian kd-tree can be broken into the following three stages:

1. Tree building
   (a) Compute the bounding box of the input positions $\vec{p_i}$.
   (b) If the maximum side length of the bounding box is less than $\rho = \sqrt{2}$, create a leaf node and terminate.

Splatting Samples	RMS Error	Time
1	0.012	10.3 s
2	0.011	10.6 s
4	0.010	11.1 s
8	0.009	12.2 s
Exact Gauss Transform		

Figure 4.3: For three different sizes of a non-local-means Gauss transform (the columns), we show the effect of changing the number of splatting samples while holding the slicing sample count constant at 64 and the clustering threshold ρ constant at $\sqrt{2}$ (twice the standard deviation of the splatting and slicing filters). The algorithm is fairly insensitive to the number of splatting samples in terms of both accuracy and time. We conservatively choose 4 as the default.

Slicing Samples	RMS Error	Time
1	0.058	1.86 s
4	0.026	2.71 s
16	0.013	5.13 s
64	0.010	11.1 s
Exact Gauss Transform		

Figure 4.4: For three different sizes of a non-local-means Gauss transform (the columns), we show the effect of changing the number of slicing samples while holding the splatting sample count constant at 4 and the clustering threshold constant at $\sqrt{2}$. Low slicing sample counts produce a noisy result. Depending on the problem size and accuracy requirements, between 16 and 64 samples are sufficient. We conservatively use 64 as the default.

Cluster Size	RMS Error	Time
σ/2	0.018	16.3 s
σ	0.013	14.6 s
2σ	0.010	11.1 s
4σ	0.013	6.09 s

Exact Gauss Transform

Figure 4.5: For three different sizes of a non-local-means Gauss transform (the columns), we show the effect of changing the threshold ρ at which input positions are grouped into clusters, while holding the splatting and slicing sample counts constant at 4 and 64 respectively. σ is the standard deviation of the splatting and slicing filters (typically $\frac{\sqrt{2}}{2}$). $\rho = 2\sigma$ gives the best accuracy. Smaller values are slower. They also result in less averaging down during splatting, so they produce high-frequency noise. Larger values can be used for additional speed, but they begin to introduce low-frequency errors.

(c) Otherwise, partition the input positions by splitting halfway along the longest bounding box edge and repeat the tree building algorithm recursively on each partition.

2. Splat the inputs onto the leaves of the tree. For each input position $\vec{p}_i$ and corresponding value $\vec{v}_i$:

   (a) Query the tree about $\vec{p}_i$ with a standard deviation of $\frac{\sqrt{2}}{2}$ and 4 samples. This returns a list of references to leaf values $\vec{l}_j$ with weights w_j.

   (b) Update each leaf value: $\vec{l}_j \leftarrow \vec{l}_j + w_j \vec{v}_i$.

3. Slice at the output positions. For each output position $\vec{p}_i$:

   (a) Query the tree about $\vec{p}_i$ with a standard deviation of $\frac{\sqrt{2}}{2}$ and 64 samples. This returns a list of references to leaf values $\vec{l}_j$ with weights w_j.

   (b) Compute the output value at $\vec{p}_i$ as: $\vec{\hat{v}}_i = \sum_j w_j \vec{l}_j$.

4.4.1 Efficient CPU implementation

A C++ implementation of the Gaussian kd-tree can be found in Appendix B. Several optimizations can be applied to this to accelerate it on a modern CPU. They are quite similar to those performed on the permutohedral lattice. At the small scale, updating each leaf node during splatting (2b), and summing the returned leaf node values during slicing (3b) can both be vectorized across the dimensions of the value vector. Value vectors are typically four-dimensional (red, green, blue, and a homogeneous weight), which well suits the four-wide SIMD vector operations found in most current CPUs.

At the larger scale each stage can be distributed across CPU cores. Tree building (1) exhibits task parallelism after the first few partitions; each building task can operate entirely independently. It is also possible to compute bounding boxes (1a) in parallel, and to use a parallel partitioning (1c) algorithm, but there is less benefit to be had here; the number of building tasks grows exponentially and can quickly saturate any number of cores.

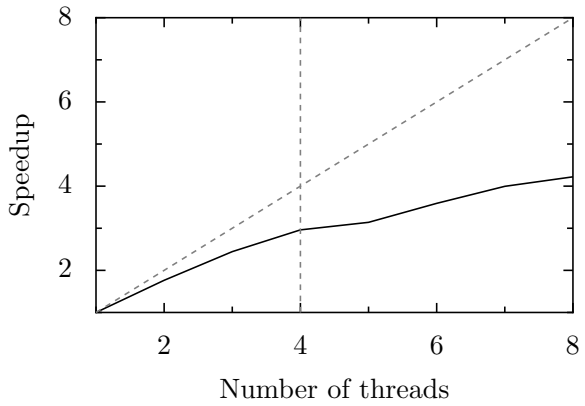

Figure 4.6: This graph shows the speedup attained by parallelizing the Gaussian kd-tree on an Intel Core i7 950 CPU. This CPU has four physical cores that each run two hardware threads. The splatting and slicing stages of the algorithm are data-parallel across pixels, and building the kd-tree is task-parallel after the first few splits, so significant speed gains are realized. However, kd-tree descent is fairly cache-incoherent, so the algorithm is highly sensitive to memory latency. Fortunately hardware threads (hyper-threading) excel at hiding memory latency; eight threads are sufficient to achieve a full 4x speedup on this four-core machine.

Splatting (2) can be parallelized across input positions. The update step (2b) must then be made atomic. Alternatively, for a small number of threads, each leaf can maintain a separate value per thread, which can be summed in an extra stage between splatting and slicing (parallelizing across leaves).

Slicing (3) is trivially parallelizable across output positions. It would also be possible to parallelize splatting and slicing using task parallelism within each query, or by parallelizing across the individual samples (giving up on our stratification scheme). However, the number of input and output positions is typically in the millions, which provides sufficient parallelism for any current CPU architecture.

4.4.2 Efficient GPU implementation

Implementing this algorithm on a modern GPU is somewhat more challenging, both due to the higher degree of parallelism, and also because building, splatting, and slicing are all recursive algorithms, which must be converted to iterative ones to run effectively on a GPU.

We implemented the algorithm in CUDA [14] and ran it on an NVIDIA GeForce GTX 465. We observed a typical speedup of 8× over the single-threaded CPU implementation running on an Intel Core i7 950 running at 3.06 GHz, and a speedup of 2× compared to the multi-threaded implementation on the same CPU.

In our GPU implementation of the Gaussian kd-tree we separate tree building into two stages. Near the root of the tree we build nodes serially on the CPU, but using parallel GPU algorithms for computing bounding boxes and partitions. Rather than using a recursive algorithm to build the tree, which effectively stores pending work in the function-call stack, we explicitly maintain a queue of pending node-building jobs. Once this queue grows to at least 1024 elements we have sufficient task parallelism available to switch to building on the GPU. Now we parallelize across the building jobs and use a serial algorithm within each thread to compute bounding boxes and partitions.

The splatting and slicing algorithms described earlier are similarly recursive, so they must be transformed for efficient implementation. The simplest way to do this is to abandon the stratification scheme and switch to individually simulating each query sample. For a single sample we can descend to a single leaf using a simple *while* loop. Accumulating values at the tree leaves while splatting must be done using atomic floating-point adds to memory, which recent versions of CUDA provide. Gathering values from leaves during slicing could similarly use atomic operations, but instead we group all the samples that gather to a particular output value into a single thread block, and use shared memory to coordinate the sum over samples using a binary reduction tree.

4.4.3 Out-of-core implementation

For some applications (such as denoising volume data), we may not be able to fit all of the positions and values in memory at once. Splatting and slicing can be done in a streaming fashion, but tree building must be modified. We load a large random subset of the position vectors into memory, and build a kd-tree using only those. We then stream through all of the position vectors, sending each to the leaf node that

contains it, and locally extending the tree if necessary. If the initial random subset selected covers the space well, we will see only a small growth of the tree.

4.5 Conclusion

The single most attractive property of the Gaussian kd-tree is that it scales linearly with the dimensionality of the position vectors. For n input positions in d dimensions, and $m < n$ leaf nodes, constructing a Gaussian kd-tree takes $O(dn \log m)$ time: At each of the $\log m$ levels of the tree, for each of the n input values, we consider each of the d dimensions once to compute a bounding box.

Querying the tree (Algorithm 4.1) using s samples takes $O(sn(\log m + d))$ time: For each input, we descend through $\log m$ inner nodes, doing a single comparison at each. Once we reach a leaf node we compute a d-dimensional Euclidean distance to compute a weight. Though samples will often be grouped together, in the worst case a query can split into s independent queries, so we have a linear dependence on s. Typically s is a small constant, and m is bounded by n, so the time complexity of performing a Gauss transform using the tree is best expressed as $O(dn \log n)$. The memory complexity of the Gaussian kd-tree is dominated by the size of the input data, and so is $O(dn)$.

4.5.1 Comparison of the Gaussian kd-tree and the permutohedral lattice

Input position vectors usually lie on a low-dimensional manifold in a d-dimensional underlying space. The Gaussian kd-tree places samples within this manifold, while the permutohedral lattice builds a simplicial scaffold around the manifold. This difference provides a key benefit to the Gaussian kd-tree.

In many higher-dimensional cases some coordinates of the position vectors may be low variance and provide little useful information. For example, we may have added a potentially useful term to each position vector which happens to be low-variance for a particular set of input data. The Gaussian kd-tree will not split on the low-variance

term, so it gracefully ignores the useless term. The permutohedral lattice will instead construct many more vertices, use more memory, and run more slowly. In general the memory use of the Gaussian kd-tree grows with the dimensionality of the manifold, while the memory use of the permutohedral lattice grows with the dimensionality of the underlying space.

4.5.2 Limitations

While it performs better than the permutohedral lattice in this regard, the Gaussian kd-tree does not conform to arbitrary input manifolds. Consider the case where many terms in the position vector are correlated. This may happen, for example, if our position vectors come from simply reading out patches of an image; neighboring pixels are highly correlated. The Gaussian kd-tree only makes axis-aligned splits, so it cannot recognize and adapt to this correlation.

This is particularly problematic in high-dimensional cases. If the maximum number of levels of the tree is similar to the dimensionality of the space ($d \approx \log n$), and all dimensions have a high-variance, we may only be able to split once in each dimension before we reach a leaf node. This means our leaf nodes are typically long and skinny, reaching out to $\pm\infty$ in many dimensions. Our assumption while querying the tree was that a Gaussian integrated over a leaf node was roughly proportional to the Gaussian evaluated at the cluster center within that leaf node. For leaf nodes with such extreme aspect ratios this is unlikely to be true, and our sampling becomes inefficient.

Performing PCA on the input positions as a preprocess can help, by decorrelating the dimensions and discarding useless ones ahead of time. However this is a global operation, and so if the manifold has varying local correlations a kd-tree cannot conform to them and may sample the manifold poorly.

In such cases it may be possible to instead use a tree based on non-axis-aligned splits (such as the random projection trees described by Dasgupta and Freund [20]). However, this would substantially complicate our query algorithm. At a split node, to compute the probabilities of descending to each child we integrate a Gaussian over

each child. If all cuts are axis-aligned we can exploit the separability of the Gaussian and consider each dimension separately. This makes the integral easy to compute and maintain during tree descent. If we introduce non-axis-aligned cuts we would have to integrate a Gaussian over a pair of arbitrary polyhedra at each inner node.

Fortunately, for the cases we find in image processing PCA serves quite well at reducing dimensionality and decorrelating position vector terms without losing any useful information. In the following chapter we will put hypotheticals aside, select a suite of typical problems, and empirically compare the Gaussian kd-tree to the permutohedral lattice and other methods of computing approximate Gauss transforms.

Chapter 5

Evaluation

In this chapter we compare the performance of the algorithms discussed so far on a suite of sample applications spanning filter sizes and dimensionalities. We are concerned mainly with runtime, but also with memory use. We compare only single-threaded CPU implementations, as those are more widely available. Most of these algorithms have comparable scaling properties when parallelized, as they are all data-parallel across pixels.

We include algorithms that meet the following criteria:

- C or C++ source code is available, so that all methods can be compiled with the same compiler with the same compiler flags on the same operating system. This is also necessary for us to be able to instrument algorithms to accurately record time and memory used.
- Can be tuned to achieve a typical RMS error under 0.01 with respect to a naive evaluation of a Gauss transform.
- Takes less than 1000 seconds to run and uses less than 4 gigabytes of memory for at least some of the test suite.

While effort has been made to include all relevant algorithms, there are many ways of accelerating our test applications. We believe that the algorithms included here are the best-in-class open-source algorithms for these applications.

5.1 Methodology

The machine used for testing is a typical high-end desktop machine (at the time of writing). It includes an Intel Core i7 950 CPU, and 6 gigabytes of RAM. Programs were compiled using gcc version 4.4.5 with all relevant optimization flags turned on. In our tests the clock starts once data has been loaded from disk and is stored in memory as an array of single-precision floating-point numbers. It stops when the output is available as a similar array. Memory use is counted as all memory allocated beyond the space required to store the input and output.

Many algorithms can be modified to be faster at the expense of accuracy, often with explicit tuning parameters. We tuned each algorithm to achieve a typical RMS error of around 0.01 with respect to a naive evaluation of the Gauss transform. This corresponds to a PSNR of 40db. This limit should give output that is visibly equivalent to that of the naive Gauss transform. There are ways to make an RMS error of 0.01 objectionable (for example by adding a high-frequency repeating pattern to an otherwise smooth image), but no instances of this occur for the algorithms tested.

5.2 Test Applications

Our test suite includes four sample applications of the Gauss transform, with a range of filter sizes for each. Filter sizes were chosen so that the low values are slightly too small to achieve a useful effect, and the high values are slightly too large. We should therefore pay most attention to the filter sizes in the middle. Our test suite includes:

- 3-dimensional grayscale bilateral filtering of a 1.5 megapixel image. This filter would be used to manipulate sharpness, tone, and contrast in a grayscale image, or to manipulate the same in the luminance channel of a color image. The spatial standard deviations of these filters are powers of two ranging from 1 to 64 pixels. This corresponds to filter footprints that range from 7×7 pixels to 385×385 pixels. The corresponding intensity standard deviations are also powers of two, ranging from $\frac{1}{32}$ to 2, where intensity is scaled to between zero and one.

Figure 5.1: The grayscale image of the tree on the left is used for testing grayscale bilateral filter algorithms, the colorful canyon in the center is used for testing 5-dimensional color bilateral filters, and the noisy desert scene on the right is used for testing 8- and 16-dimensional denoising.

- 5-dimensional color bilateral filtering of a 1.5 megapixel image. This would typically be used to manipulate sharpness, tone, and contrast in color images. The spatial and color-space standard deviations used here match those used for the 3-dimensional case.

- 8-dimensional non-local means of a 1.5 megapixel image. In this application an image is denoised using 7×7 patches weighted with a Gaussian mask of standard deviation 1. Patches are reduced to 6 dimensions using PCA, and two spatial terms are added to make up the 8 dimensions. The spatial standard deviations range from 8 to 512, which correspond to filter footprints from about 50×50 pixels to about 3000×3000 pixels (i.e. a truly *non-local* means). The patch-space standard deviations range from $\frac{1}{32}$ to 2.

- 16-dimensional non-local means of the same 1.5 megapixel image. This application is similar to the above, except the patches used are 9×9 weighted with a Gaussian of standard deviation 1.4. The patches are reduced to 14 dimensions using PCA, to which the two spatial dimensions are added. The standard deviations used are identical to the 8-dimensional case.

The input images for each application are shown in Figure 5.1. Crops of the input and ideal outputs for each application are shown in Figures 5.2, 5.3, 5.4, and 5.5. These were produced with a naive evaluation of the Gauss transform.

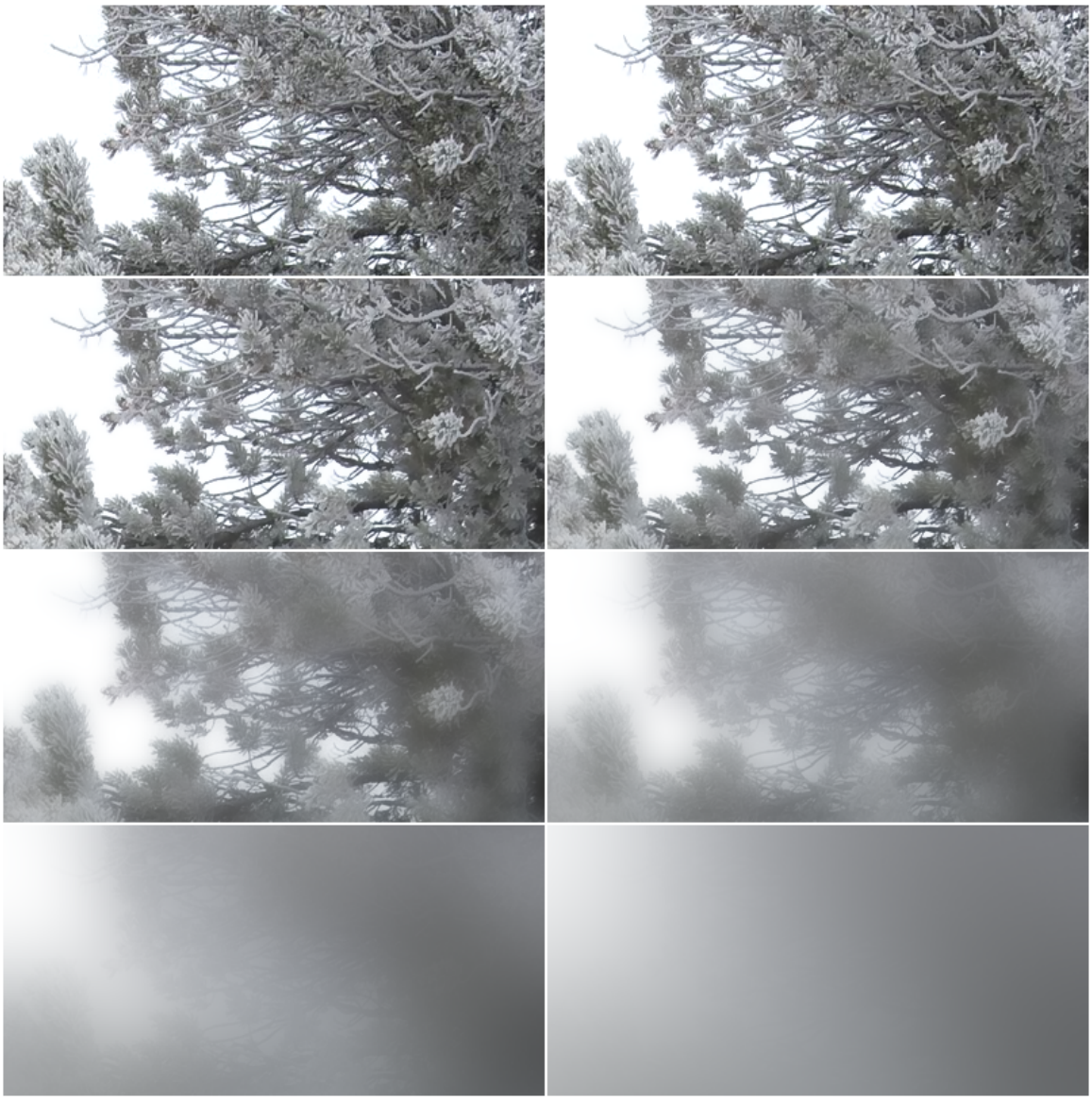

Figure 5.2: These crops of the tree in Figure 5.1 show the effects of a grayscale bilateral filter. Such a filter would be used to isolate tone and contrast at different scales. The input is at the top left, and the standard deviations used to filter increase to the right and downwards.

Figure 5.3: These crops of the canyon in Figure 5.1 show the effects of a color bilateral filter. The input is at the top left, and the standard deviations used to filter increase to the right and downwards.

Figure 5.4: In these crops of the desert scene in Figure 5.1, we see the effects of 8-dimensional non-local means. The input is at the top left, and the standard deviations used to filter increase to the right and downwards.

Figure 5.5: In these crops of the desert scene in Figure 5.1, we see the effects of 16-dimensional non-local means. The input is at the top left, and the standard deviations used to filter increase as we move right and down. The results are quite similar to those of 8-dimensional non-local means (Figure 5.4). In fact, eight dimensions are usually enough for non-local means, as demonstrated by Tasdizen [43]. We include this case nonetheless to test the scaling performance of the algorithms as dimensionality increases.

5.3 The algorithms

Naive: This algorithm computes a naive Gauss transform, considering all pixels within three spatial standard deviations of each input pixel. This is a standard naive implementation of the bilateral filter or non-local means. While this algorithm scales linearly with dimensionality, it scales quadratically in filter size. It is a plausible choice only for very small filter sizes.

Bilateral grid: This is the bilateral grid of Paris et al. [36], using multi-linear splatting and slicing, and a separable blur kernel of [1, 2, 1] in each dimension. It scales exponentially with dimensionality, but is simple to implement, easy to parallelize, and very fast for low-dimensional cases.

Permutohedral lattice: This is the permutohedral lattice described in Chapter 3. It scales quadratically with dimensionality, but performs more arithmetic than the bilateral grid for lower dimensional cases. We should expect it to perform well at moderate dimensionalities.

Sparse grid: The two major differences between the permutohedral lattice and the bilateral grid are the choice of lattice, and also the fact that the permutohedral lattice implementation stores values sparsely in a hash table. In order to disambiguate these two effects, we also benchmark a sparse bilateral grid algorithm that uses the same hash table implementation. However, the time and memory complexity still grow exponentially with dimensionality.

Gaussian kd-tree: This is the Gaussian kd-tree described in Chapter 4. It scales linearly with dimensionality, but with a fairly high constant, so we should expect to see it perform well in the higher-dimensional cases. We use 4 splatting samples and 64 slicing samples, which are the conservative values we recommended earlier.

Improved fast Gauss transform: This is the fast method of evaluating the Gauss Transform of C. Yang et al. [49]. It is a fully general method capable of extremely

high accuracy, but even when tuned for speed, it is not particularly fast compared to the more approximate methods used in image filtering. The implementation used is the open-source figtree library [33] described by Morariu et al. [34]. We set the sole accuracy parameter to 0.1, which reliably provides an RMS error below 0.01 in our tests. The figtree library automatically selects all other parameters.

Its main competitor in the artificial intelligence literature is the dual-tree method of Lee, Gray, and Moore [31]. Using an implementation published by the authors as part of mlpack [26], the dual-tree method was found to be extremely slow for the large filter sizes used in image processing; even with parameters tuned for maximum speed (given our accuracy requirement), in no cases did the algorithm terminate within 1000 seconds, so it was disqualified from this comparison.

Real-time $O(1)$ bilateral filtering: This is the method of Yang et al. [50]. It targets only the three-dimensional case, for which it sweeps through the intensity levels, computing intermediate filtered images for each level, and filling in output pixels as their intensities are reached. We modified the implementation provided by the authors to work on arbitrary floating-point input, as all the other methods benchmarked here do.

5.4 Results

Figures 5.6 and 5.7 show the results of runtime, memory, and error tests for our algorithms on each of the four tasks. Our analysis is broken up by problem.

5.4.1 Grayscale bilateral filtering (d = 3)

Several algorithms are well-suited to grayscale bilateral filtering. The real-time bilateral filter of Yang et al. [50] uses a consistently low amount of memory and is the fastest. However it filters using only linear interpolation in the intensity dimension, whereas the bilateral grid performs an explicit blur as well. For very large intensity standard deviations linear interpolation begins to cause inaccuracies in the output.

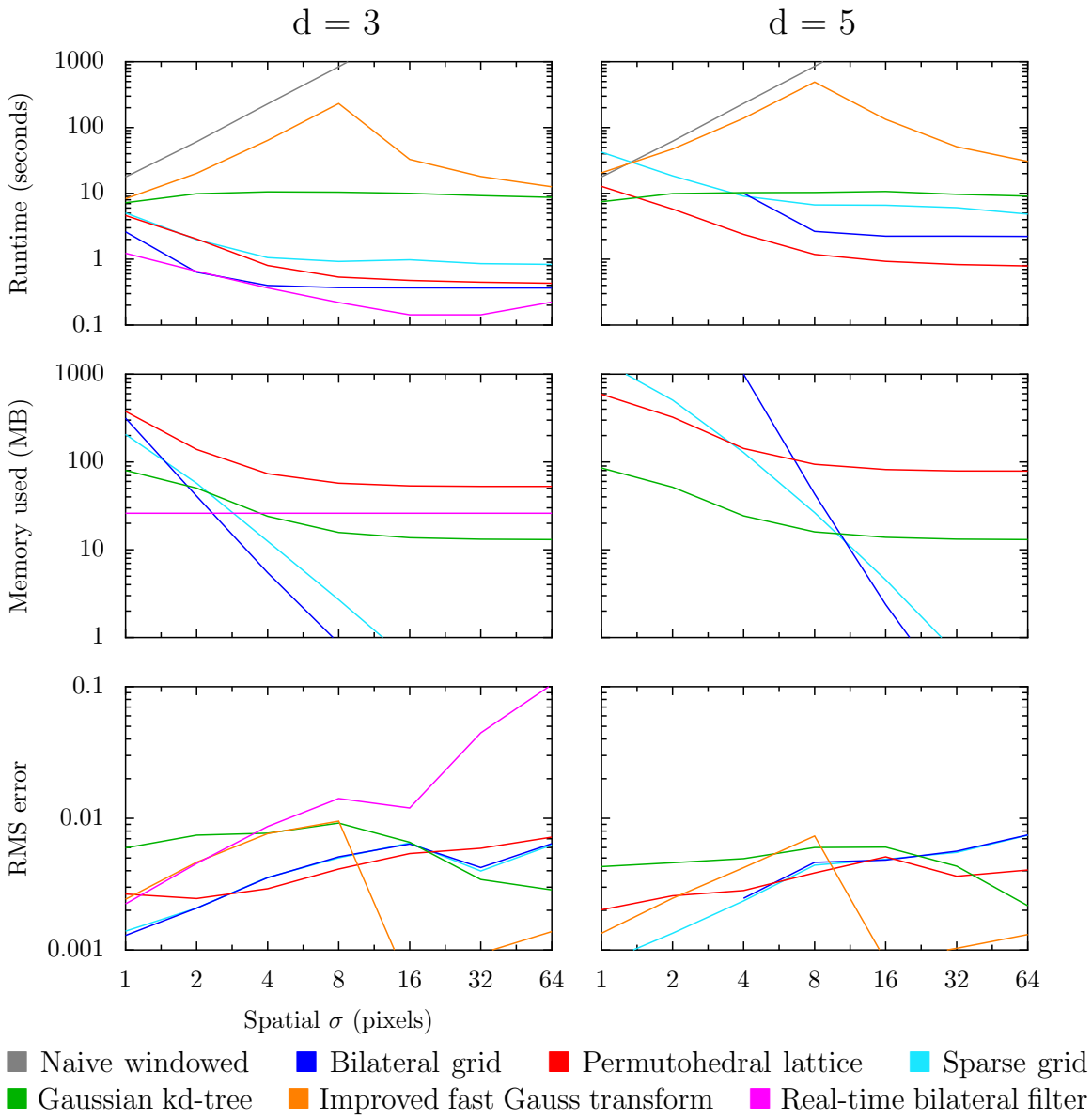

Figure 5.6: This graph shows the runtime, peak memory use, and RMS error of various algorithms when performing grayscale (left column), and color (right column) bilateral filters, each as a function of the spatial size of the filter. The real-time O(1) bilateral filter [50] (pink) and the bilateral grid [36] (blue) are both fine choices for grayscale filtering. For color bilateral filters, the fastest method is the permutohedral lattice (red). However, if memory use is a concern, either the Gaussian kd-tree (green) or the bilateral grid (blue) is preferable, depending on filter size.

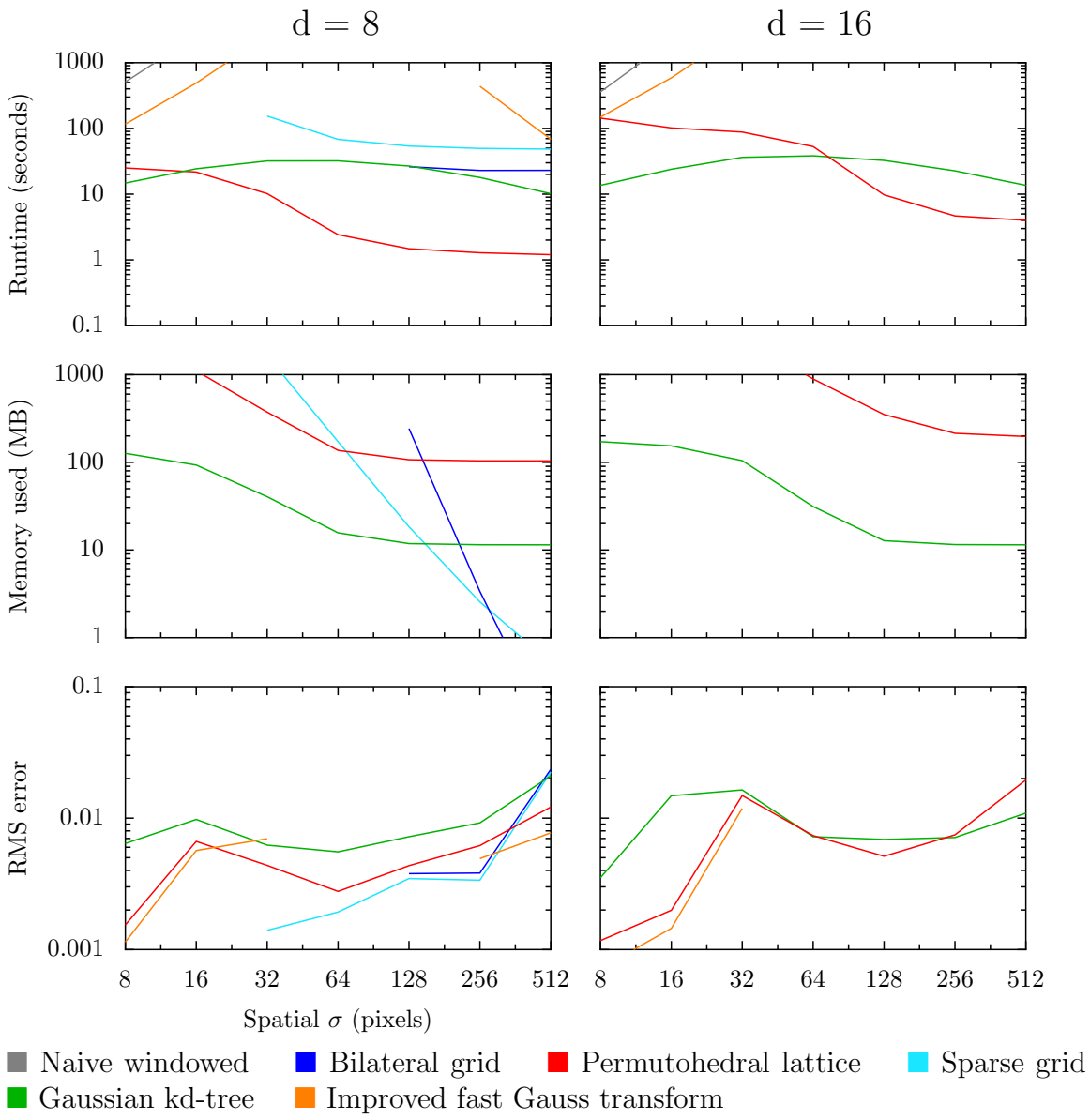

Figure 5.7: This graph shows the runtime, peak memory use, and RMS error of various algorithms when performing 8-dimensional (left column), and 16-dimensional (right column) non-local means. At 8 dimensions, the permutohedral lattice is fastest. At 16 dimensions the Gaussian kd-tree may be faster depending on filter size. However, note that in both cases the Gaussian kd-tree consistently uses ten times less memory.

These inaccuracies do not render the output useless; merely different to that of the naive Gauss transform. Furthermore such large filters are rarely used in practice, so this is unlikely to cause problems for most applications. If it is a concern one could switch to the bilateral grid for larger filter sizes, or modify Yang's method to store more than two intensity slices at once and perform higher-order interpolation.

5.4.2 Color bilateral filtering (d = 5)

For color bilateral filtering the permutohedral lattice is the fastest. It does however use a lot of memory for small to moderate filter sizes (over 100 MB for our 18 MB input). If memory use is a concern, the Gaussian kd-tree may be a better choice. A five-dimensional bilateral grid is also feasible for larger filter sizes, though for smaller filters it requires too much memory to be practical (and too much memory to be benchmarked; hence the missing data for the bilateral grid for smaller filter sizes).

5.4.3 8-D non-local means (d = 8)

At 8 dimensions the permutohedral lattice is still roughly 10 times faster than the Gaussian kd-tree, but uses 10 times as much memory. The other methods either use too much time or too much memory to be benchmarked for all cases. For typical denoising applications then, the choice of algorithm should come down to whether there is enough memory available to use the permutohedral lattice. If there is not, the Gaussian kd-tree is the best alternative.

5.4.4 16-D non-local means (d = 16)

At 16 dimensions, the permutohedral lattice is still competitive in terms of speed, but its memory use has become exorbitant. For this 18 MB input the permutohedral lattice is consuming over a gigabyte of memory for small to moderate filter sizes. Recall that the permutohedral lattice constructs a simplicial scaffold around the input position vectors. The number of lattice points thus constructed grows linearly with the dimensionality, d. As each lattice point stores its location as a vector of length d

(to use as a hash table key), memory use grows quadratically with d. The Gaussian kd-tree is the superior algorithm in this regime.

5.4.5 Which algorithms shouldn't be used at all?

Some algorithms were not competitive for any test. First, even for the smallest spatial standard deviation tested, $\sigma = 1$, which corresponds to a filter footprint of 7×7 pixels, the naive windowed implementation of these filters is slower than using one of our acceleration structures.

Second, the sparse bilateral grid, which uses the same hash table as our permutohedral lattice but tessellates space with hypercubes instead of simplices, is always inferior to the dense bilateral grid. This demonstrates that it is the choice of lattice, rather than the sparsity, which makes the permutohedral lattice faster than the bilateral grid. The reason is simple. For d dimensions, during splatting and slicing the permutohedral lattice touches $d + 1$ vertices. For each vertex one floating-point multiply and accumulate must be done per color channel. A grid touches 2^d lattice points during splatting and slicing. At $d = 5$, for example, this means a grid requires about 5 times as many floating-point operations to splat and slice. This factor dominates the overhead involved in computing lattice vertices and looking them up in the hash table.

Finally, the improved fast Gauss transform fared quite poorly in our comparisons. This algorithm is ill-suited to these applications two reasons. First, the improved fast Gauss transform is better equipped to handle tasks requiring higher accuracy, and incurs a lot of overhead in order to achieve this. Second, operating with homogeneous coordinates covers many sins. The bilateral grid, the permutohedral lattice, and the Gaussian kd-tree are all fairly inaccurate for non-homogeneous Gauss transforms, which is the intended application domain of the improved fast Gauss transform. We will return to this issue in Section 7.3.

5.4.6 Which algorithm should I use?

The tests above give us an indication of which algorithm to use for a few specific dimensionalities. We can expand this picture to the full range of dimensionalities we might encounter by modifying our denoising task to preserve between 1 and 18 dimensions after PCA.

Figure 5.8 shows which algorithm is fastest as a function of dimensionality and filter size for this task. A reasonable conclusion to draw is that we should use the permutohedral lattice for dimensionalities below 12, and the Gaussian kd-tree after that. This threshold can be moved up or down based on the memory available. If three-dimensional filters are of particular interest, it is also worth switching to the bilateral grid or the real-time bilateral filter for those cases.

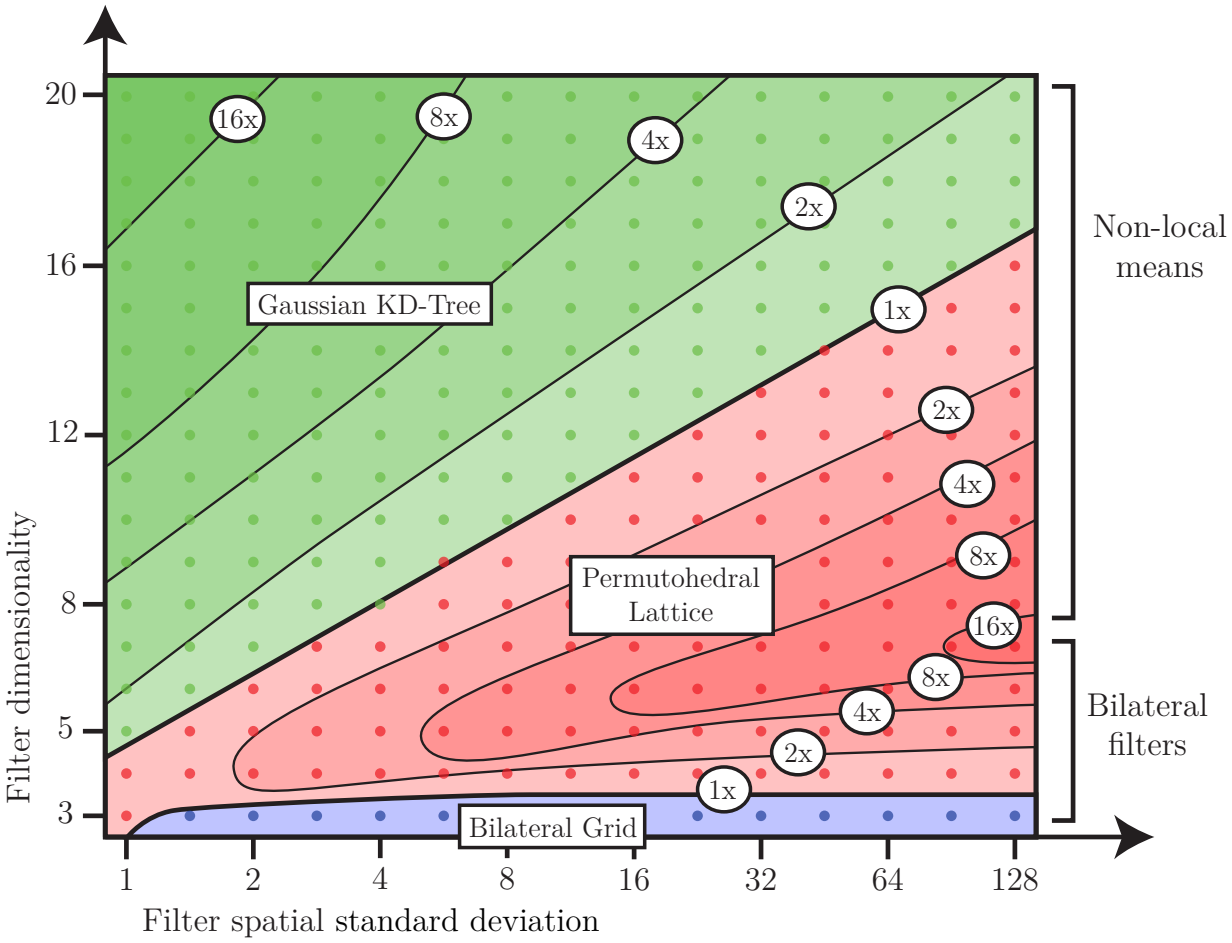

Figure 5.8: This contour plot shows the fastest method for each dimensionality and spatial filter size, and how many times faster it is than the second fastest method. The bilateral grid [15] is best for three-dimensional filters. The Gaussian KD-Tree [2] is best for high dimensionalities and small filter sizes. The permutohedral lattice is the fastest method for dimensionalities from 4 up to around 12, depending on the filter size. Runtimes were sampled at the colored dots and interpolated. Only methods capable of arbitrary-dimensional filters were compared.

Chapter 6

Applications

In Chapter 1 we discussed basic applications of Gauss transforms in photography. In this chapter we examine two more applications in photography, and one application of the Gauss transform in the related field of medical imaging.

6.1 Burst denoising

Lack of light is usually the most significant limiting factor in photography. Photographs taken with insufficient light require a high analog gain, which amplifies noise. There are two conventional methods for acquiring more light: lengthening the exposure time, or adding additional light with a flash. Either method creates artifacts. Lengthy exposure times create motion blur, and a camera flash destroys desirable ambient illumination, introduces hard shadows, over-brightens nearby objects, and causes a "red-eye" effect.

A third method for acquiring more light has recently become popular: take many noisy photographs and combine them. For a static scene, a noisy burst of photographs need merely be aligned and averaged to substantially reduce the noise (Telleen at al. [44]). Averaging n photographs reduces noise by a factor of $\sqrt{n}$. Noise scales roughly linearly with analog gain, and so a burst of 256 photographs at an ISO of 1600 can be averaged to produce an effective ISO of 100. However, averaging large numbers of photographs is not a panacea. The storage and processing costs incurred

by large bursts are considerable, and most interesting scenes (e.g. humans) contain some internal motion.

Bennett and McMillan [9] address a similar problem: denoising video. They average pixels either across space or time using a bilateral filter. Their temporal "bilateral" filter computes distance using small local patches, and so it is in fact what we would call non-local means along the temporal axis.

However, the algorithm of Bennett and McMillan cannot denoise areas in motion. For still photography, we can deal with motion using optical flow to warp all input images to some single reference frame. Modern optical flow algorithms are moderately robust to noise, and even if the flow vectors are incorrect in some frames, the resulting artifacts tend to average away in the output (see Figure 6.1). If we also apply a denoising method to the burst before averaging, we can substantially reduce the number of frames required to produce a noise-free photograph.

In this application, we compute a single noise-free output photograph from a noisy burst using the following pipeline:

1. Perform a joint bilateral filter of chrominance with respect to luminance to ameliorate the effects of hot pixels. These manifest as brightly colored dots in the input. This is a three-dimensional Gauss transform, and so we use a bilateral grid. We use a spatial standard deviation of 4 pixels, and a luminance standard deviation of 0.25.

2. Select one image from the burst to be the reference. This is usually the one with the best composition.

3. Compute the flow field from the reference to each input image. We use the algorithm of Brox et al. [11].

4. Denoise the entire burst using non-local means. Each patch will search for matching patches across the burst about the flow vectors. We can implement this by simply subtracting the optical flow vectors from the spatial component of our position vectors. Note that this results in position vectors that no longer lie on a uniform grid, so we must use a method that can tolerate this. This

is an advantage over other methods shared by the permutohedral lattice and the Gaussian kd-tree. We use 13×13 patches weighted with a Gaussian of standard deviation 2, reduced to 10 dimensions using PCA. We search about the flow vectors using a spatial standard deviation of 4 pixels, and a patch-space standard deviation of 0.2. The input burst of photographs consumes a large amount of memory, so we use the Gaussian kd-tree to perform the Gauss transform to minimize the memory overhead.

5. Warp all the images to match the reference using the optical flow vectors, and then average the sequence across time.

6. Add simulated photon-shot noise. Averaging large numbers of frames tends to over-smooth flat areas of the image. We expect to see some amount of noise in any photograph due to the physical properties of light. Adding a small amount of noise also masks any loss of sharpness caused by non-local means and by averaging along imperfect flow vectors.

This pipeline (along with several alternatives) is illustrated for two example scenes in Figures 6.3 and 6.2. Both scenes were captured using a Casio EX-F1 operating at ISO 1600 with an exposure time of 1/60s. The captured images were averaged down by a factor of two in each direction, and then further digitally amplified. In Figure 6.3, pixel values were doubled; in Figure 6.2 they were tripled.

Combining multiple frames in this way is a promising direction for casual photography. Small cameras have small apertures that let in very little light. However, during aiming and focusing the image sensor is running continuously. These images are currently discarded, and so their light is wasted. It would be better to save them, and use them to denoise the image acquired when the shutter button is pressed. In this way the shutter button merely marks a moment in time. Images acquired during the surrounding 10 to 20 seconds can be combined to create the actual photograph. The major barrier to adopting this technique is the amount of computation required. This dissertation makes non-local means computationally cheaper, but high-quality optical flow is still very expensive. In both of the cases illustrated here, computing optical flow takes about 90% of the total runtime.

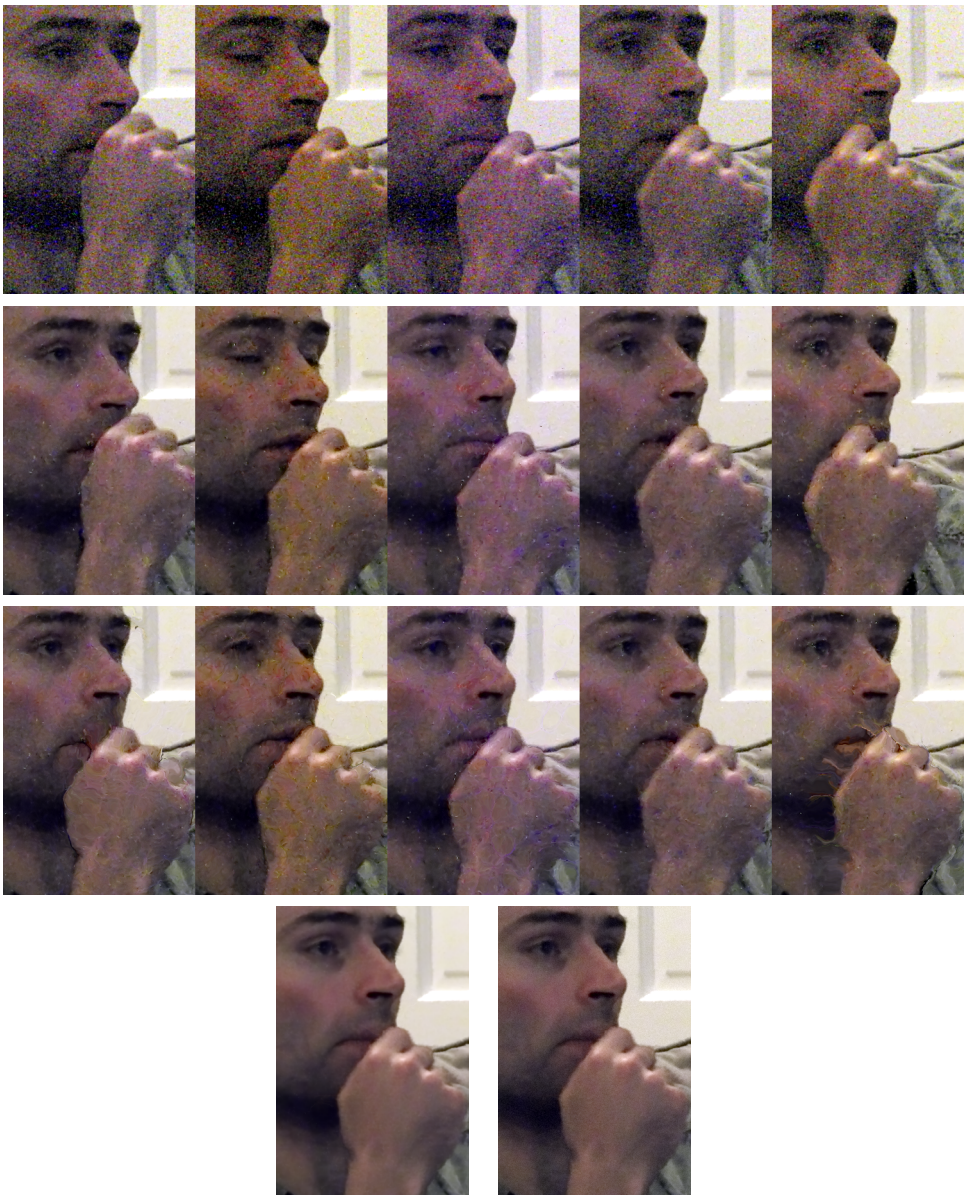

Figure 6.1: The top row shows crops of 5 frames from a 27 frame burst. To combine these photographs into a single noise free image, we first perform optical flow, and denoise using non-local means along the flow vectors (second row). We then use the flow vectors again to warp each frame to align to the frame with the desired composition (third row). Note that the warping has introduced artifacts where there were occlusions (the blinking eyes and the moving hand). By averaging these frames we remove the residual noise and hide the optical flow artifacts (bottom left). Finally, we add simulated photon shot noise to correct for the over-smoothed look of the averaged frame (bottom right).

Figure 6.2: In this example, we combine 27 noisy photographs into a single output. The top left shows the noisy input frame with the desired composition. We first perform a joint bilateral filter of chrominance with respect to luminance to remove some chrominance noise (top right). If we globally align and average the result across time we introduce blur due to the movement of the subjects (middle left). Warping the frames using optical flow and then average produces better results (middle right), but does not remove all the noise. We instead perform non-local means along the flow vectors before averaging (bottom left), and then add some simulated photon shot noise (bottom right). We can thus produce a passable photograph from these extremely noisy inputs.

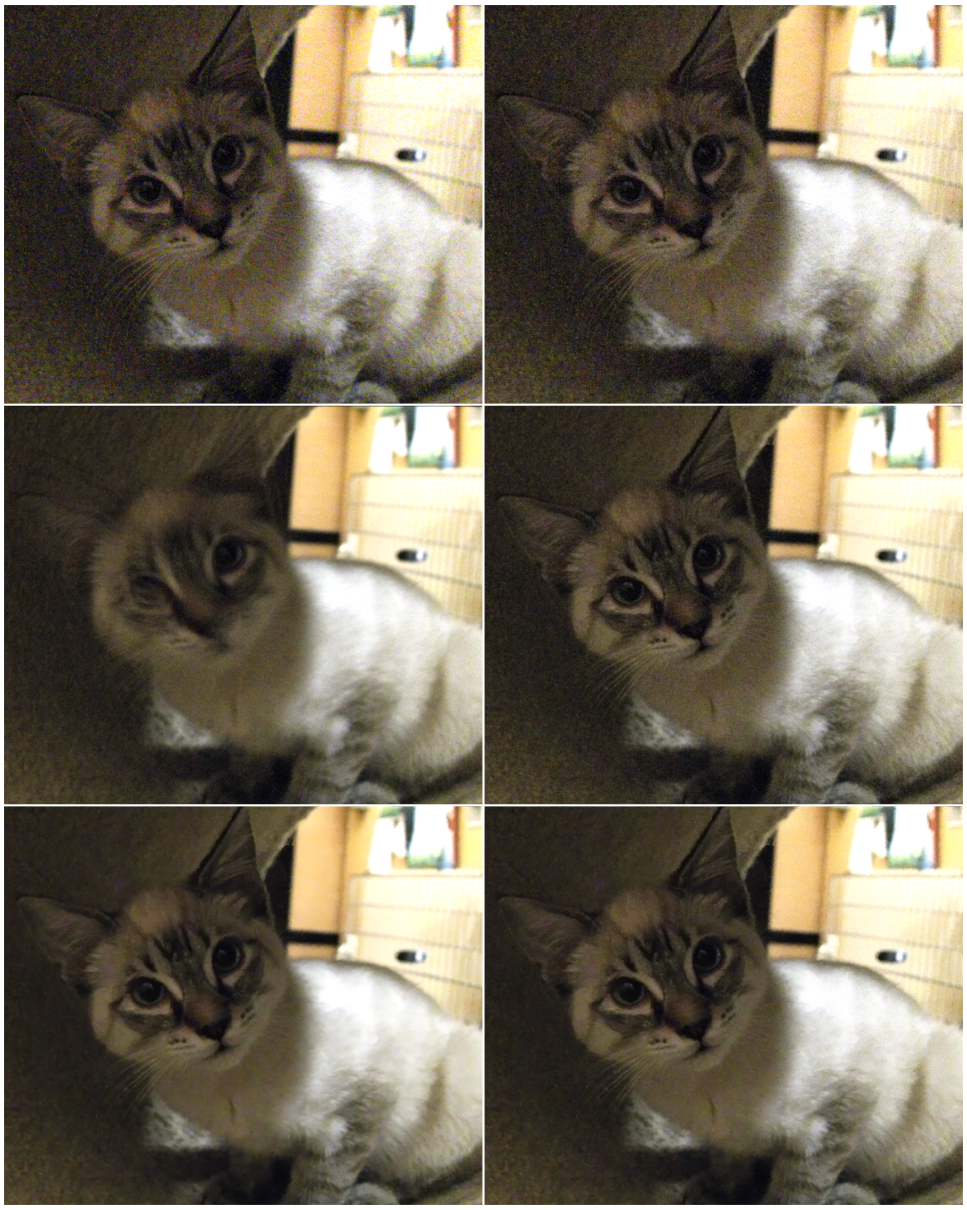

Figure 6.3: This second example follows the same pipeline as in Figure 6.2, but with only four less-noisy inputs. The input photograph with the desired composition is in the top left. First we remove some chrominance noise (top right). Globally aligning and averaging the frames produces blur, as the kitten is rotating its head (middle left). Warping the frames using optical flow and then averaging is better (middle right), but not as good as denoising the volume before warping and averaging (bottom left), and then adding simulated photon shot noise (bottom right).

6.2 Automatically-propagating local edits

Post-production of photographs usually includes both global edits like color correction, and also local edits such as dodging and burning. Local changes to sharpness, brightness, contrast, saturation, and other properties are commonplace. These manipulations are usually painted on using a mask, and are often the most time-consuming tasks in post-production, because they require careful application to avoid artifacts.

For example, if we wish to darken a bright sky in a landscape image, the horizon must be dealt with very carefully to avoid either creating a dark halo below it (where the brush has strayed too far into the landscape) or a bright halo above it (where the brush has erred on the side of caution).

In this application we use a joint non-local means filter to make local adjustments to an image in a way that *automatically* propagates across similar colors and textures. This operates in a similar manner to the propagating edits of An and Pellacini [4], but is an order of magnitude faster, allowing for interactive-rate manipulations.

The user applies approximate edits with a few strokes. Each stroke paints values on a mask in those locations. The mask is then filtered with respect to position vectors derived from the input image. We use six PCA terms and two spatial terms to capture changes in brightness, color, and texture. By choosing appropriate position dimensions, local edits can be made to respect boundaries with respect to any set of local descriptors. The filtered mask serves as an influence map for how the edit should be applied.

To adjust brightness, for example, the user paints dark or bright values into the mask. We then filter the mask, and use it to modulate the input image. Filtering is done at interactive rates (10 frames per second at 800×600), so the user can see the fully propagated edit as they paint it on the canvas. To achieve this speed we use the GPU implementation of the permutohedral lattice, and omit the blur stage. For an example session, see Figure 6.4.

Figure 6.4: We can use the GPU implementation of the permutohedral lattice to make local manipulations to a photograph automatically propagate to similarly-textured regions. The user paints strokes (bottom left) on the input image (top left). These strokes are filtered in real time with respect to 8-dimensional position vectors derived from the input image. The filtered mask (bottom right) is then used to modulate the input to produce the output (top right), which is what the user sees while working. In this case we have darkened the far hill and near grass, and brightened the girl's hair, the post, the far sky, and the flowers. This session took about 20 seconds.

6.3 Volume denoising

Non-local means is quite popular in medical imaging, as it easily extends to volume data. Instead of two-dimensional patches we simply use small sub-volumes. In Figure 6.5 we show the result of non-local means applied to a volume data set of a bacterium. This data set (courtesy of Amat et al. [3]) was acquired using cryo-electron tomography. Such volumes are typically very noisy, because bombarding specimens with large numbers of electrons tends to alter them, meaning few electrons must be used, limiting the signal-to-noise ratio obtainable.

The extension to volume data is straightforward. However, there are two more subtle aspects to this particular application of non-local means that make it interesting. First, we found it advantageous to omit a spatial term and filter only with respect to patch distance. The volume is homogeneous; any given local structure recurs widely spread throughout much of the volume. Spatially remote patches may be quite close in patch-space, and may have a significant influence on each other. A conventional non-local means implementation which searches within some fixed radius would either miss these matches or devolve into an intractable all-pairs $O(n^2)$ search. In contrast, the permutohedral lattice and the Gaussian kd-tree both speed up as the spatial search window expands.

Second, the noise in this volume is not uniform white noise. Rather, it is "pink" noise, which is dominated by lower frequencies, and so neighboring pixels have highly correlated noise. This means that spatially nearby patches are similar by virtue of having a similar noise pattern, regardless of the similarity of the underlying structure. A Gauss transform therefore blurs the volume in addition to denoising. We can counteract this by spatially blurring the volume and extrapolating from our output away from the blurred volume. Note that spatially blurring the volume is a Gauss transform in its own right. The implication is this: a linear combination of Gauss transforms may be superior to a single Gauss transform, which is equivalent to smoothing in patch-space using a non-Gaussian filter. For any given application this raises an intriguing question: which combination of Gaussians is optimal for this task? We leave an attempt to address this question in the general case as an open problem.

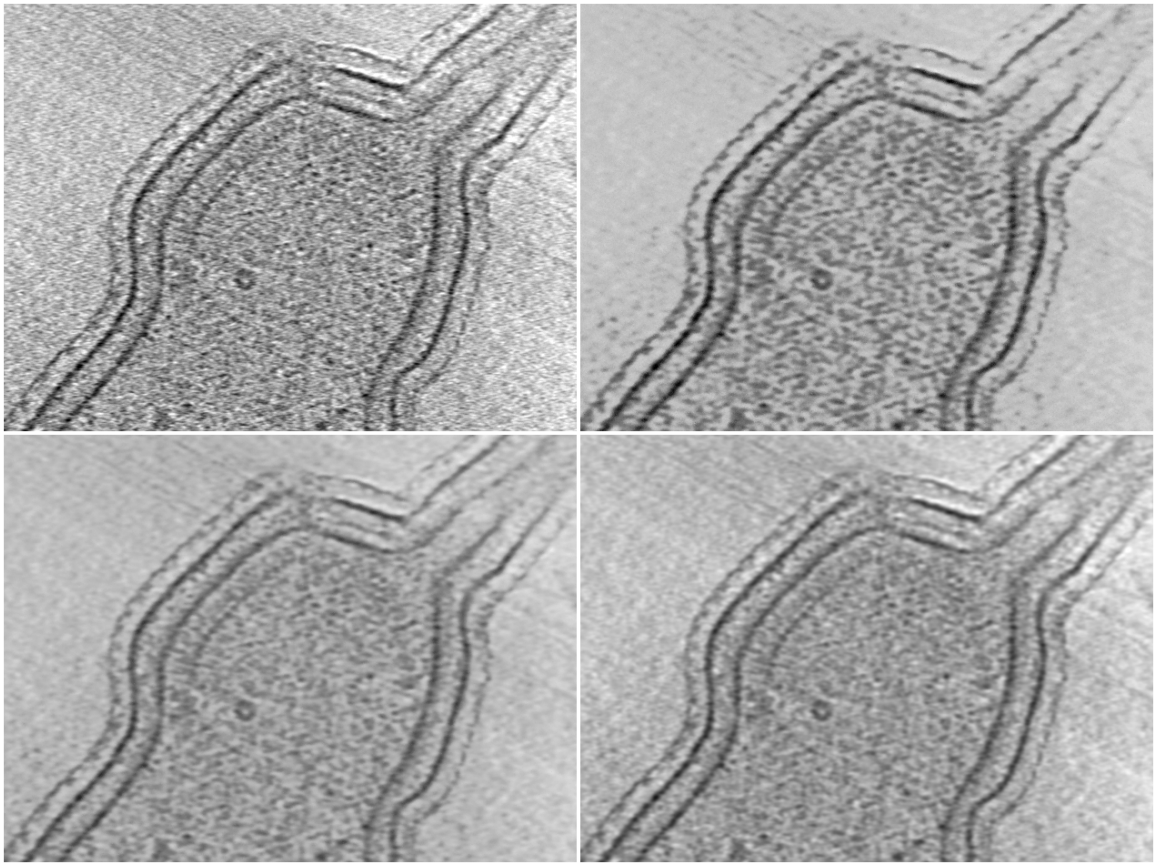

Figure 6.5: The top left shows a slice of a noisy $640 \times 480 \times 240$ volume data set of a *Caulobacter crescentus* acquired using cryo-electron tomography. The volume is denoised using non-local means with $9 \times 9 \times 9$ volumetric patches weighted with a 3-dimensional Gaussian of standard deviation 1.4. The patches are reduced to 16 dimensions using PCA, and non-local means is performed using the Gaussian kd-tree. The patch space standard deviation is 0.75. We do *not* include a spatial term, so this computation would be intractable with conventional methods. The output of non-local means is in the bottom left. It appears similar to simply blurring the volume (bottom right). If we linearly exaggerate the difference between the two we obtain the result in the top right. Structures such as the cell walls are preserved, while the noise is almost eliminated.

Chapter 7

Conclusion

We have now explored the use of Gauss transforms in photographic image processing, using the two novel data structures described in this dissertation: the Gaussian kd-tree and the permutohedral lattice. The key insights of these two techniques can be summarized in a few sentences:

We can express many useful operations in photographic image processing as a Gauss transform, which can be accelerated by resampling and blurring in a high-dimensional space. For moderate dimensionalities, we can most efficiently resample using a lattice which tessellates space with simplices. For higher dimensionalities, we can compute our Gaussian-weighted integrals using importance-sampled kd-tree queries.

It is our hope that the applications described in the previous chapter are merely the tip of the iceberg. Fast Gauss transforms using these data structures should enable new applications in computational photography, and help us answer the central questions of the field.

Discrete Gauss transforms are widely used in many other fields, such as artificial intelligence and computer vision. The data structures described in this dissertation may also unlock new applications in these fields, and the application of techniques like these to the diverse requirements of other fields will undoubtedly be an interesting research topic in its own right.

In this final chapter we lay out future work to be done in this area; first in the

form of limitations of the data structures described, and second in the form of new applications to be considered.

7.1 Limitations of the permutohedral lattice

The permutohedral lattice has three outstanding issues:

The output is produced by high-dimensional barycentric interpolation, which sometimes produces visible piecewise-linear artifacts in the output. This problem can be surmounted by using higher-order interpolation. While one may be tempted to expand the slicing and splatting kernels to touch more lattice vertices, this quickly introduces an exponential dependence on dimension. Instead, we could store a linear model at each lattice vertex as opposed to a simple constant. Each lattice vertex would describe a linear function of best fit for the input positions that fall within its influence. By interpolating between these linear models during slicing we can produce a piecewise-quadratic output. To save time and memory we could also constrain the linear model to vary only with the spatial dimensions of the position vector, which would be enough to remove the piecewise-linear artifacts in the output image (illustrated in Figure 3.5).

The implementation requires storing lattice points in a hash table. This data structure is awkward to parallelize on wide data-parallel processors like GPUs. This is a widely-discussed issue common to GPU implementations of algorithms like this one. We expect these architectures to evolve to better support these kinds of sparse data structures.

The permutohedral lattice creates a simplicial scaffold about the input manifold. This representation uses a very large amount of memory as dimensionality increases. This issue is fundamental to the lattice and difficult to overcome. It may be possible to finesse this by only probabilistically creating lattice points based on the expected weight that arrives at them. One could also try

a nearest-neighbor splatting procedure, coupled with creating higher-order models at the lattice points to regain the lost accuracy. This quickly becomes equivalent to modifying the fast Gauss transform [28] to use simplices instead of hypercubes, which may be a worthwhile route.

7.2 Limitations of the Gaussian kd-tree

The Gaussian kd-tree also has three outstanding issues worthy of discussion:

It is hard to choose good sample counts for splatting and slicing. In Chapter 4 we conservatively chose 4 samples for splatting and 64 samples for slicing. However for many applications fewer samples produce acceptable results. Also, it may be possible to save time by spatially varying the sample counts. For example, when performing a bilateral filter, large flat regions can be accurately filtered using very few samples. Finally, it may be worthwhile examining the applicability of the variance-reduction techniques used in ray tracing.

Building a kd-tree is hard to parallelize. The Gaussian kd-tree algorithm exhibits mixed parallelism. Splatting and slicing are simple and data-parallel, and so are well supported by GPU architectures. Tree building is more complex. Initially it uses large reductions (computing a bounding box) and sorting-like steps (partitioning the data over a split node). These are difficult to implement efficiently on a GPU. Once the building process warms up, it begins to exhibit task-parallelism, which is also difficult to exploit on a GPU.

At higher dimensionalities, our weighted importance sampling becomes inefficient as each split node insufficiently constrains its children. During a query of the Gaussian kd-tree, the number of samples sent to each leaf node is proportional to a Gaussian integrated over that leaf. This is an approximation to the ideal probability, which is the Gaussian evaluated at the sample stored in each leaf. We corrected for this approximation using a weight attached to each returned

leaf node. However, for n inputs the depth of the tree is $O(\log n)$, which may be comparable to the dimensionality of the space for higher dimensionalities. This means that each leaf node probably extends to $\pm$ infinity in one or more dimensions. As the dimensionality increases our approximation becomes increasingly poor, and the sampling may grow inefficient; it may return many leaf nodes with a weight near zero. A tree that more tightly constrains the extent of its leaves may be superior for such high-dimensional cases. For example, a tree based on a hierarchy of bounding spheres would make our approximation very accurate, but would require more work at each split node to determine how many samples to send to each child.

7.3 The non-homogeneous case

All of the applications described in this dissertation used Gauss transforms of homogeneous values. That is, instead of computing a weighted sum of Gaussians, we compute a weighted average. We did this by augmenting our input value vectors with an extra coordinate with value 1. On the output side, we divide through by this last coordinate to convert the weighted sum into a weighted average. All of our tests for accuracy were done on typical image processing tasks which used this homogeneous representation.

A homogeneous representation can hide certain types of error. First, imagine if the output were off by some global scale factor. This would be corrected in the division and would not affect the final image. To perform an accurate non-homogeneous Gauss transform we would need to compute and correct for this global scale factor.

This issue is not particularly daunting; even if our algorithms cannot be modified to remove any global scale factor, most applications involve other constraints which can be used to solve for it. For example, one common use of non-homogeneous Gauss transforms is in kernel density estimation, in which the output function is treated as a probability distribution, and is known to have a sum of 1.

A second type of error is far more challenging to deal with. Consider the case where each output value is individually scaled by some unknown factor. The homogeneous case is unaffected by this type of error, as it is cancelled in the final division. This

second type of error does indeed occur with both the permutohedral lattice, the Gaussian kd-tree, and other methods based on resampling.

Let us take the case of a uniformly-weighted Gauss transform (all value vectors are the constant 1):

$$\hat{\vec{v}}_i = \sum_j e^{-|\vec{p}_i - \vec{p}_j|^2/2}$$

Now consider an outlier – a lone position vector far from any other. With the permutohedral lattice, this value may be placed arbitrarily with respect to the lattice. In the two extreme cases, it may lie exactly on a lattice point, or it may lie at the centroid of a simplex. If we ignore the blurring stage, the output value at the same position is 1 in the first case, and $\frac{1}{d+1}$ in the second. Thus the relative error can be quite large, and is spatially varying. The blur stage complicates the picture slightly, but does not alter the underlying issue.

In this particular case, the Gaussian kd-tree will create one cluster at the outlying position, and send all splatting and slicing samples to that cluster for that query position. Thus it will correctly produce an output value of 1 (assuming we have corrected for the global scale induced by our sample counts).

The Gaussian kd-tree, however, is still based on resampling, and produces the same type of error in a slightly more complex case. Consider a position vector that lies midway between two cluster centers. During splatting it contributes $\frac{1}{2}$ to each cluster center. During slicing it weighs these two values equally, and its contribution to its own output value will be $\frac{1}{2}^2 + \frac{1}{2}^2 = \frac{1}{2}$ where it should be 1.

These effects are significant. In a simulated application with 64 equally-weighted Gaussians of standard deviation 1, uniformly randomly located within a 10×10 square, the Gaussian kd-tree produces a typical RMS error of around 0.15 relative to an exact Gauss transform, and the permutohedral lattice produces an RMS error of 0.3. Not performing any Gauss transform and simply setting all the output values to 1 gives an RMS error of 0.4. These numbers are a far cry from the typical RMS errors under 0.01 that we see in the homogeneous case.

Resampling approaches like these produce good far-field estimates; two points

several standard deviations apart will contribute roughly the right amount of energy to each other. The near-field estimates, however, can be almost arbitrarily bad. It may be possible to augment one of these data structures to behave differently for these near-field interactions. Alternatively, we could borrow the method of the fast Gauss transform and use the higher-order models described above; instead of splatting constant values onto the permutohedral lattice, each input position could contribute to a polynomial expansion about the vertices of the enclosing simplex. Slicing could then be done by interpolating the responses of the same polynomials.

7.4 Further applications

The applications described in Chapters 1 and 6 are by no means an exhaustive list. Here are a few more possible directions to explore.

7.4.1 Least-squares smoothing

Gaussian filtering can be interpreted as enforcing a smoothness prior on the data: small changes in position vector should correspond to small changes in value. It enforces this via explicit smoothing in the space of position vectors. However, data structures like the bilateral grid and permutohedral lattice already encode a notion of smoothness; a coarse sampling of a space (combined with a reasonable reconstruction filter) can only ever produce smooth functions. We might therefore enforce a smoothness prior less aggressively by asking: what are the best values to store at each lattice vertex, such that after slicing we most closely match the input (in the least squares sense)? This would be the smooth function that most closely resembles the input, which may denoise without the loss of tone and contrast seen with non-local means (See for example Figures 5.4 and 5.5).

7.4.2 Filtering gradients or PCA terms

The bilateral filter is fairly poor at smoothing gradients, as the footprint of the filter is truncated to a thin band of similar intensities. This property of the filter is usually

helpful, as it prevents smoothing across step edges, but it also serves to accentuate any irregularities in a smoothly varying region. The trilateral filter of Choudhury and Tumblin [16] addresses this by modifying the bilateral filter to tilt its footprint to match local gradients. The effect is that the trilateral filter favors piecewise-linear output, whereas the bilateral filter favors piecewise-constant output.

An alternative way to achieve the same effect is to perform a bilateral filter on image gradients, and then re-integrate using a Poisson solver (as in [38]), as piecewise-constant gradients entail piecewise-linear output. A gradient field of a color image is naturally six-dimensional, so this may be an excellent application for the permutohedral lattice.

More generally one could imagine filtering not just gradients but arbitrary local descriptors, and then solving for the image that best matches the descriptors produced. For example, consider non-local means with PCA-reduced patches, which uses the output of a filter bank for position vectors and homogeneous pixel color for value vectors. Instead let us use the filter bank responses for both positions and values. The problem of producing an output image from the filtered responses is an over-constrained deconvolution, which can be easily solved with division in Fourier-space.

7.4.3 Scaling up non-local means

Non-local means doesn't scale as elegantly as it could. As you throw more and more data into the mix, the number of plausible but incorrect matches to each patch grows, resulting in a loss of fine detail. Each pixel is also attracted towards the average color of all the data used, resulting in a loss of overall tone. These effects are illustrated in Figure 7.1. A more discriminating non-local means could potentially be used to denoise an image using the entire corpus of photographs available on the internet. Here is a sketch of such an algorithm:

1. Using metadata, high-level scene understanding algorithms, or simple image descriptors, acquire a large candidate set of images from the web that are likely to contain patches useful for denoising some noisy input image. For landscape

Figure 7.1: As we include more and more image data as reference, non-local means performs increasingly poorly. In the top left is a crop from the noisy image used earlier in Chapter 5. In the center is the same crop of non-local means applied with no spatial term; each patch is denoised using every other patch in the image. Much of the noise is removed. On the right we try to improve the result by including 40 other images from the same photo album as reference patches. Some of these images are shown across the bottom. Two types of artifact result. First, there is a loss of overall tone and contrast. Note for example the rich red hues present in the midground in the input that are merely brown in the output. Second, much of the fine detail is over-smoothed, such as the bush in the foreground.

photographs a GPS tag and time of day is sufficient to find other photographs of the same location in similar lighting. For portraits, a face-recognition algorithm run against your existing photo collection would provide useful patches. For our example noisy landscape used in Figure 7.1, an image search for "monument valley at dawn" produces a wealth of similar images.

2. Splat all of these photographs into a high-dimensional data structure such as the Gaussian kd-tree or permutohedral lattice.

3. Slice out a denoised photograph using position vectors derived from your noisy input image.

To attempt such an algorithm, we first need to address non-local means' tendency to pull all colors towards the average. There are several viable approaches. The simplest is to separate the image into base and detail layers (with a bilateral filter), and only perform non-local means on the detail layer. The unchanged base layer

would preserve global tone and contrast. There are some interesting questions to answer here, such as whether it is better to derive position vectors from the input or from the detail layer alone.

We next need to address the loss of detail. This is substantially more difficult. One approach could be to use non-Gaussian high-dimensional filters, as we did in Section 6.3. A Gaussian is always positive. If we instead used a filter with negative lobes we could make good matches attract and merely-plausible matches actively repel. Such a filter could be accelerated with the techniques in this dissertation by expressing it as a difference of Gaussians. Selecting the best Gaussians to use is an interesting problem.

7.5 Closing remarks

The rise of digital photography has empowered photographers with an unprecedented amount creative control by moving much of the imaging process from the realm of physics and chemistry into software. Instead of merely reproducing the light that strikes the sensor, we treat this data as one of many raw ingredients to be used to make a photograph. Computational photography provides a toolbox of algorithms that manipulate raw photographic data in myriad ways to produce photographs that could never have been produced with a traditional film camera.

This dissertation adds two tools to the box: the permutohedral lattice and the Gaussian kd-tree. They can be used to quickly filter, decompose, or denoise images – essential steps in the digital darkroom. We hope that these algorithms are also of interest to imaging hardware or software architects, looking for algorithms with mixed types of parallelism. We hope that they are of interest to aficionados of the Gauss transform in other fields. Most of all, we hope that these tools are of use to fellow computational photographers, as our community continues to redefine what it means to make a photograph.

Appendix A

Permutohedral lattice source code

This appendix contains C++ source code for the permutohedral lattice. It assumes the input values and positions are available as arrays of single-precision floating-point numbers. The interesting functions are `PermutohedralLattice::filter`, which enacts the splat-blur-slice pipeline, and `PermutohedralLattice::splat`, which identifies the vertices of the simplex enclosing a query position.

```
#include <math.h>
#include <vector>
#include <memory>

using std::vector;

// Hash table implementation for permutohedral lattice.
//
// The lattice points are stored sparsely using a hash table.
// The key for each point is its spatial location in the (d+1)-
// dimensional space.
class HashTablePermutohedral {
public:
    // Hash table constructor
    //  kd : the dimensionality of the position vectors
    //  vd : the dimensionality of the value vectors
    HashTablePermutohedral(int kd, int vd) : kd(kd), vd(vd) {
        filled = 0;
        entries.resize(1 << 15);
```

```
        keys.resize(kd*entries.size()/2);
        values.resize(vd*entries.size()/2, 0.0f);
    }

    // Returns the number of vectors stored.
    int size() { return filled; }

    // Returns a pointer to the keys array.
    vector<short> &getKeys() { return keys; }

    // Returns a pointer to the values array.
    vector<float> &getValues() { return values; }

    // Looks up the value vector associated with a given key. May or
    // may not create a new entry if that key doesn't exist.
    float *lookup(const vector<short> &key, bool create = true) {
        // Double hash table size if necessary
        if (create && filled >= entries.size()/2) { grow(); }

        // Hash the key
        size_t h = hash(&key[0]) % entries.size();

        // Find the entry with the given key
        while (1) {
            Entry e = entries[h];

            // Check if the cell is empty
            if (e.keyIdx == -1) {
                if (!create) return NULL;  // Not found

                // Need to create an entry. Store the given key.
                for (int i = 0; i < kd; i++) {
                    keys[filled*kd+i] = key[i];
                }
                e.keyIdx = filled*kd;
                e.valueIdx = filled*vd;
                entries[h] = e;
                filled++;
                return &values[e.valueIdx];
            }

            // check if the cell has a matching key
            bool match = true;
```

```
            for (int i = 0; i < kd && match; i++) {
                match = keys[e.keyIdx+i] == key[i];
            }
            if (match) {
                return &values[e.valueIdx];
            }

            // increment the bucket with wraparound
            h++;
            if (h == entries.size()) { h = 0; }
        }
    }

    // Hash function used in this implementation. A simple base conversion.
    size_t hash(const short *key) {
        size_t h = 0;
        for (int i = 0; i < kd; i++) {
            h += key[i];
            h *= 2531011;
        }
        return h;
    }

private:
    // Grows the hash table when it runs out of space
    void grow() {
        // Grow the arrays
        values.resize(vd*entries.size(), 0.0f);
        keys.resize(kd*entries.size());
        vector<Entry> newEntries(entries.size()*2);

        // Rehash all the entries
        for (size_t i = 0; i < entries.size(); i++) {
            if (entries[i].keyIdx == -1) { continue; }
            size_t h = hash(&keys[entries[i].keyIdx]) % newEntries.size();
            while (newEntries[h].keyIdx != -1) {
                h++;
                if (h == newEntries.size()) { h = 0; }
            }
            newEntries[h] = entries[i];
        }

        entries.swap(newEntries);
```

```
    }

    // Private struct for the hash table entries.
    struct Entry {
        Entry() : keyIdx(-1), valueIdx(-1) {}
        int keyIdx;
        int valueIdx;
    };

    vector<short> keys;
    vector<float> values;
    vector<Entry> entries;
    size_t filled;
    int kd, vd;
};

// The algorithm class that performs the filter
//
// PermutohedralLattice::filter(...) does all the work.
//
class PermutohedralLattice {
public:

    // Performs a Gauss transform
    //  pos : position vectors
    //  pd  : position dimensions
    //  val : value vectors
    //  vd  : value dimensions
    //  n   : number of items to filter
    //  out : place to store the output
    static void filter(const float *pos, int pd,
                       const float *val, int vd,
                       int n, float *out) {

        // Create lattice
        PermutohedralLattice lattice(pd, vd, n);

        // Splat
        for (int i = 0; i < n; i++) {
            lattice.splat(pos + i*pd, val + i*vd);
        }

        // Blur
```

```
        lattice.blur();

        // Slice
        lattice.beginSlice();
        for (int i = 0; i < n; i++) {
            lattice.slice(out + i*vd);
        }
    }

    // Permutohedral lattice constructor
    //  pd : dimensionality of position vectors
    //  vd : dimensionality of value vectors
    //  n  : number of points in the input
    PermutohedralLattice(int pd, int vd, int n) :
        d(pd), vd(vd), n(n), hashTable(pd, vd) {

        // Allocate storage for various arrays
        elevated.resize(d+1);
        scaleFactor.resize(d);
        greedy.resize(d+1);
        rank.resize(d+1);
        barycentric.resize(d+2);
        canonical.resize((d+1)*(d+1));
        key.resize(d+1);
        replay.resize(n*(d+1));
        nReplay = 0;

        // compute the coordinates of the canonical simplex, in which
        // the difference between a contained point and the zero
        // remainder vertex is always in ascending order.
        for (int i = 0; i <= d; i++) {
            for (int j = 0; j <= d-i; j++) {
                canonical[i*(d+1)+j] = i;
            }
            for (int j = d-i+1; j <= d; j++) {
                canonical[i*(d+1)+j] = i - (d+1);
            }
        }

        // Compute part of the rotation matrix E that elevates a
        // position vector into the hyperplane
        for (int i = 0; i < d; i++) {
            // the diagonal entries for normalization
```

```
            scaleFactor[i] = 1.0f/(sqrtf((float)(i+1)*(i+2)));

            // We presume that the user would like to do a Gaussian
            // blur of standard deviation 1 in each dimension (or a
            // total variance of d, summed over dimensions.)  Because
            // the total variance of the blur performed by this
            // algorithm is not d, we must scale the space to offset
            // this.
            //
            // The total variance of the algorithm is:
            //  [variance of splatting] +
            //  [variance of blurring] +
            //  [variance of splatting]
            //   = d(d+1)(d+1)/12 + d(d+1)(d+1)/2 + d(d+1)(d+1)/12
            //   = 2d(d+1)(d+1)/3.
            //
            // So we need to scale the space by (d+1)sqrt(2/3).

            scaleFactor[i] *= (d+1)*sqrtf(2.0/3);
        }
    }

    // Performs splatting with given position and value vectors
    void splat(const float *position, const float *value) {

        // First elevate position into the (d+1)-dimensional hyperplane
        elevated[d] = -d*position[d-1]*scaleFactor[d-1];
        for (int i = d-1; i > 0; i--)
            elevated[i] = (elevated[i+1] -
                           i*position[i-1]*scaleFactor[i-1] +
                           (i+2)*position[i]*scaleFactor[i]);
        elevated[0] = elevated[1] + 2*position[0]*scaleFactor[0];

        // Prepare to find the closest lattice points
        float scale = 1.0f/(d+1);

        // Greedily search for the closest remainder-zero lattice point
        int sum = 0;
        for (int i = 0; i <= d; i++) {
            float v = elevated[i]*scale;
            float up = ceilf(v)*(d+1);
            float down = floorf(v)*(d+1);
```

```
        if (up - elevated[i] < elevated[i] - down) {
            greedy[i] = (short)up;
        } else {
            greedy[i] = (short)down;
        }

        sum += greedy[i];
    }
    sum /= d+1;

    // Rank differential to find the permutation between this
    // simplex and the canonical one.
    for (int i = 0; i < d+1; i++) rank[i] = 0;
    for (int i = 0; i < d; i++) {
        for (int j = i+1; j <= d; j++) {
            if (elevated[i] - greedy[i] < elevated[j] - greedy[j]) {
                rank[i]++;
            } else {
                rank[j]++;
            }
        }
    }

    if (sum > 0) {
        // Sum too large - the point is off the hyperplane. We
        // need to bring down the ones with the smallest
        // differential
        for (int i = 0; i <= d; i++) {
            if (rank[i] >= d + 1 - sum) {
                greedy[i] -= d+1;
                rank[i] += sum - (d+1);
            } else {
                rank[i] += sum;
            }
        }
    } else if (sum < 0) {
        // Sum too small - the point is off the hyperplane. We
        // need to bring up the ones with largest differential
        for (int i = 0; i <= d; i++) {
            if (rank[i] < -sum) {
                greedy[i] += d+1;
                rank[i] += (d+1) + sum;
            } else {
```

```
                rank[i] += sum;
            }
        }
    }

    // Compute barycentric coordinates
    for (int i = 0; i < d+2; i++) { barycentric[i] = 0.0f; }
    for (int i = 0; i <= d; i++) {
        barycentric[d-rank[i]] += (elevated[i] - greedy[i]) * scale;
        barycentric[d+1-rank[i]] -= (elevated[i] - greedy[i]) * scale;
    }
    barycentric[0] += 1.0f + barycentric[d+1];

    // Splat the value into each vertex of the simplex, with
    // barycentric weights
    for (int remainder = 0; remainder <= d; remainder++) {
        // Compute the location of the lattice point explicitly
        // (all but the last coordinate - it's redundant because
        // they sum to zero)
        for (int i = 0; i < d; i++) {
            key[i] = greedy[i] + canonical[remainder*(d+1) + rank[i]];
        }

        // Retrieve pointer to the value at this vertex
        float *val = hashTable.lookup(key, true);

        // Accumulate values with barycentric weight
        for (int i = 0; i < vd; i++) {
            val[i] += barycentric[remainder]*value[i];
        }

        // Record this interaction to use later when slicing
        replay[nReplay].offset = val - &hashTable.getValues()[0];
        replay[nReplay].weight = barycentric[remainder];
        nReplay++;
    }
}

// Prepare for slicing
void beginSlice() {
    nReplay = 0;
}
```

```
// Performs slicing out of position vectors. The barycentric
// weights and the simplex containing each position vector were
// calculated and stored in the splatting step.
void slice(float *col) {
    const vector<float> &vals = hashTable.getValues();
    for (int j = 0; j < vd; j++) { col[j] = 0; }
    for (int i = 0; i <= d; i++) {
        ReplayEntry r = replay[nReplay++];
        for (int j = 0; j < vd; j++) {
            col[j] += r.weight*vals[r.offset + j];
        }
    }
}

// Performs a Gaussian blur along each projected axis in the hyperplane.
void blur() {

    // Prepare temporary arrays
    vector<short> neighbor1(d+1), neighbor2(d+1);
    vector<float> zero(vd, 0.0f);
    vector<float> newValue(vd*hashTable.size());
    vector<float> &oldValue = hashTable.getValues();

    // For each of d+1 axes,
    for (int j = 0; j <= d; j++) {
        // For each vertex in the lattice,
        for (int i = 0; i < hashTable.size(); i++) {
            // Blur point i in dimension j

            short *key = &(hashTable.getKeys()[i*d]);
            for (int k = 0; k < d; k++) {
                neighbor1[k] = key[k] + 1;
                neighbor2[k] = key[k] - 1;
            }
            neighbor1[j] = key[j] - d;
            neighbor2[j] = key[j] + d;

            float *oldVal = &oldValue[i*vd];
            float *newVal = &newValue[i*vd];

            float *v1 = hashTable.lookup(neighbor1, false);
            float *v2 = hashTable.lookup(neighbor2, false);
            if (!v1) v1 = &zero[0];
```

```
                if (!v2) v2 = &zero[0];

                // Mix values of the three vertices
                for (int k = 0; k < vd; k++) {
                    newVal[k] = (v1[k] + 2*oldVal[k] + v2[k]);
                }
            }

            newValue.swap(oldValue);
        }
    }

private:

    int d, vd, n;
    vector<float> elevated, scaleFactor, barycentric;
    vector<short> canonical, key, greedy;
    vector<char> rank;

    struct ReplayEntry {
        int offset;
        float weight;
    };
    vector<ReplayEntry> replay;
    int nReplay;

    HashTablePermutohedral hashTable;
};
```

Appendix B

Gaussian kd-tree source code

This appendix contains C++ source code for the Gaussian kd-tree. Like the code above, it assumes the input values and positions are available as arrays of single-precision floating-point numbers. The interesting functions are `GKDTree::filter`, which enacts building, splatting, and slicing, and `Split::lookup`, which describes how to send a query down the tree.

```
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <vector>
#include <limits>
#include <algorithm>

using std::vector;
using std::swap;

const float INF = std::numeric_limits<float>::infinity();

// Random floating-point number between zero and one
float randFloat() {
    return rand()/(RAND_MAX+1.0f);
}
```

```
// A quartic approximation to the integral of a Gaussian of variance 1/2
inline float gCDF(float x) {
    x *= 0.81649658092772592f;
    if (x < -2) {
        return 0;
    } else if (x < -1) {
        x += 2;
        x *= x;
        x *= x;
        return x;
    } else if (x < 0) {
        return 12 + x*(16 - x*x*(8 + 3*x));
    } else if (x < 1) {
        return 12 + x*(16 - x*x*(8 - 3*x));
    } else if (x < 2) {
        x -= 2;
        x *= x;
        x *= x;
        return 24 - x;
    } else {
        return 24;
    }
}

// The algorithm class that performs the filter
//
// GKDTreeLattice::filter(...) does all the work.
//
class GKDTree {
  public:

    // Performs a Gauss transform
    //  pos : position vectors
    //  pd  : position dimensions
    //  val : value vectors
    //  vd  : value dimensions
    //  n   : number of items to filter
    //  out : place to store the output
    static void filter(const float *pos, int pd,
                       const float *val, int vd,
                       int n, float *out) {
        // Make an array of pointer to each position vector. We'll
        // reshuffle this array while building the tree.
```

```
vector<const float *> points(n);
for (int i = 0; i < n; i++) {
    points[i] = pos + i*pd;
}

// Build a tree. The last argument is the maximum side length
// of a cell. We set it to twice the standard deviation of
// splatting and slicing.
GKDTree tree(pd, &points[0], points.size(), sqrtf(2.0f));

// Arrays to use while splatting and slicing
vector<int> indices(64);
vector<float> weights(64);

// The values stored at the leaves
vector<float> leafValues(tree.nLeaves()*vd, 0.0f);

// Splatting: For each position vector ...
for (int i = 0; i < n; i++) {
    // find up to 4 leaves nearby ...
    int results = tree.lookup(pos + i*pd, &indices[0],
                              &weights[0], 4);
    // and scatter to them.
    for (int j = 0; j < results; j++) {
        for (int k = 0; k < vd; k++) {
            leafValues[indices[j]*vd + k] += val[i*vd + k]*weights[j];
        }
    }
}

// Clear the output array
memset(out, 0, sizeof(float)*n*vd);

// Slicing: For each position vector ...
for (int i = 0; i < n; i++) {
    // find up to 64 leaves nearby ...
    int results = tree.lookup(pos + i*pd, &indices[0],
                              &weights[0], 64);
    // and gather from them.
    for (int j = 0; j < results; j++) {
        for (int k = 0; k < vd; k++) {
            out[i*vd + k] += leafValues[indices[j]*vd + k]*weights[j];
        }
```

```
            }
        }
    }

    // GKDTree constructor
    //  kd     : the dimensionality of the position vectors
    //  pos    : an array of pointers to the position vectors
    //  n      : the number of position vectors
    //  sBound : the maximum allowable side length of a leaf node
    GKDTree(int kd, const float **pos, int n, float sBound) :
        dimensions(kd), sizeBound(sBound), leaves(0) {

        // Recursively build the tree
        root = build(pos, n);

        // Recursively compute the bounds of each node
        vector<float> kdtreeMins(dimensions, -INF);
        vector<float> kdtreeMaxs(dimensions, +INF);
        root->computeBounds(&kdtreeMins[0], &kdtreeMaxs[0]);
    }

    // Destructor. Recursively deletes the tree.
    ~GKDTree() {
        delete root;
    }

    int nLeaves() {
        return leaves;
    }

    // Query the kdtree. Returns the number of leaf nodes found.
    //  query    : the position around which to search
    //  ids      : the ids of the leaf nodes found
    //  weights  : the weight for each leaf node found
    //  nSamples : how many query samples to use
    int lookup(const float *query, int *ids, float *weights, int nSamples) {
        return root->lookup(query, ids, weights, nSamples, 1);
    }

private:

    // The interface for nodes
    class Node {
```

```
  public:
    virtual ~Node() {}

    // Query the kdtree. Same interface as above, but also tracks
    // the probability of reaching this node using the last
    // argument.
    virtual int lookup(const float *query, int *ids,
                       float *weights, int nSamples, float p) = 0;

    // Compute the bounds of the node along the cut dimension
    virtual void computeBounds(float *mins,
                               float *maxs) {}
};

// An internal split node.
class Split : public Node {
  public:
    virtual ~Split() {
        delete left;
        delete right;
    }

    // For a Gaussian centered at the given value, truncated to
    // within this leaf, what is the fraction of the Gaussian on
    // the left of this cut value. This gives the probability of
    // splitting left at this node.
    inline float pLeft(float value) {
        float val = gCDF(cutVal - value);
        float minBound = gCDF(minVal - value);
        float maxBound = gCDF(maxVal - value);
        return (val - minBound) / (maxBound - minBound);
    }

    int lookup(const float *query, int *ids, float *weights,
               int nSamples, float p) {
        // Compute the probability of a sample splitting left
        float val = pLeft(query[cutDim]);

        // Common-case optimization for a single sample
        if (nSamples == 1) {
            if (randFloat() < val) {
                return left->lookup(query, ids, weights, 1, p*val);
            } else {
```

```
            return right->lookup(query, ids, weights, 1, p*(1-val));
        }
    }

    // Send some samples to the left of the split
    int leftSamples = (int)(floorf(val*nSamples));

    // Send some samples to the right of the split
    int rightSamples = (int)(floorf((1-val)*nSamples));

    // There's probably one sample left over by the
    // rounding. Probabilistically assign it to the left or
    // right.
    if (leftSamples + rightSamples != nSamples) {
        float fval = val*nSamples - leftSamples;
        if (randFloat() < fval) leftSamples++;
        else rightSamples++;
    }

    int samplesFound = 0;

    // Descend the left subtree.
    if (leftSamples > 0) {
        samplesFound += left->lookup(query, ids, weights,
                                     leftSamples, p*val);
    }

    // Descend the right subtree
    if (rightSamples > 0) {
        samplesFound += right->lookup(query,
                                      ids + samplesFound,
                                      weights + samplesFound,
                                      rightSamples, p*(1-val));
    }

    return samplesFound;
}

// Recursively compute the bounds of each cell in the
// dimension that that cell cuts along.
void computeBounds(float *mins, float *maxs) {
    minVal = mins[cutDim];
    maxVal = maxs[cutDim];
```

```
            maxs[cutDim] = cutVal;
            left->computeBounds(mins, maxs);
            maxs[cutDim] = maxVal;

            mins[cutDim] = cutVal;
            right->computeBounds(mins, maxs);
            mins[cutDim] = minVal;
        }

        // The dimension along which this cell cuts
        int cutDim;

        // The cut value and bounds in that dimension
        float cutVal, minVal, maxVal;

        // The children of this node
        Node *left, *right;
    };

    // A leaf node. Has an id and a position.
    class Leaf : public Node {
    public:

        int lookup(const float *query, int *ids,
                   float *weights, int nSamples, float p) {
            // p is the probability with which one sample arrived
            // here.  Calculate the correct probability, q, by
            // evaluating the Gaussian.

            float q = 0;
            for (size_t i = 0; i < position.size(); i++) {
                float delta = query[i] - position[i];
                q += delta*delta;
            }
            q = expf(-q);

            // Weight each sample by the ratio of the correct
            // probability to the actual probability.
            *weights = nSamples * q / p;
            *ids = id;

            return 1;
```

```
    }

    int id;
    vector<float> position;
};

// Construct a kd-tree node from an array of position vectors
Node *build(const float **pos, int n) {
    // Compute a bounding box
    vector<float> mins(dimensions, +INF), maxs(dimensions, -INF);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < dimensions; j++) {
            if (pos[i][j] < mins[j]) mins[j] = pos[i][j];
            if (pos[i][j] > maxs[j]) maxs[j] = pos[i][j];
        }
    }

    // Find the longest dimension
    int longest = 0;
    for (int i = 1; i < dimensions; i++) {
        if (maxs[i] - mins[i] > maxs[longest] - mins[longest]) {
            longest = i;
        }
    }

    if (maxs[longest] - mins[longest] > sizeBound) {
        // If it's large enough, cut in that dimension and make a
        // split node
        Split *node = new Split;
        node->cutDim = longest;
        node->cutVal = (maxs[longest] + mins[longest])/2;

        // resort the input over the split
        int pivot = 0;
        for (int i = 0; i < n; i++) {
            // The next value is larger than the pivot
            if (pos[i][longest] >= node->cutVal) continue;

            // We haven't seen anything larger than the pivot yet
            if (i == pivot) {
                pivot++;
                continue;
            }
```

```
                // The current value is smaller than the pivot
                swap(pos[i], pos[pivot]);
                pivot++;
            }

            // Build the two subtrees
            node->left = build(pos, pivot);
            node->right = build(pos+pivot, n-pivot);
            return node;
        } else {
            // Make a leaf node with a sample in the center of the
            // bounding box
            Leaf *node = new Leaf;
            node->id = leaves++;
            node->position.resize(dimensions);
            for (int i = 0; i < dimensions; i++) {
                node->position[i] = (mins[i] + maxs[i])/2;
            }
            return node;
        }
    }

    Node *root;
    int dimensions;
    float sizeBound;
    int leaves;
};
```

Appendix C

Bilateral filter source code

This appendix demonstrates how to use the permutohedral lattice to perform a color bilateral filter. We assume an image type is available that provides a reference to the pixel at x, y in color channel c via `im(x,y,c)`. This implementation is intended to be as simple as possible. Numerous optimizations could be applied. One significant optimization would be to generate position and value vectors on the fly while splatting and slicing rather than pregenerating them as this code does.

```
// A bilateral filter of a color image with the given spatial standard
// deviation and color-space standard deviation
void bilateral(Image im, float spatialSigma, float colorSigma) {

    // Construct the five-dimensional position vectors and
    // four-dimensional value vectors
    vector<float> positions(im.width*im.height*5);
    vector<float> values(im.width*im.height*4);

    int idx = 0;
    for (int y = 0; y < im.height; y++) {
        for (int x = 0; x < im.width; x++) {
            positions[idx*5+0] = x/spatialSigma;
            positions[idx*5+1] = y/spatialSigma;
            positions[idx*5+2] = im(x,y,0)/colorSigma;
            positions[idx*5+3] = im(x,y,1)/colorSigma;
            positions[idx*5+4] = im(x,y,2)/colorSigma;
            values[idx*4+0] = im(x,y,0);
```

```
            values[idx*4+1] = im(x,y,1);
            values[idx*4+2] = im(x,y,2);
            values[idx*4+3] = 1.0f;
            idx++;
        }
    }

    // Perform the Gauss transform. For the five-dimensional case the
    // Permutohedral Lattice is appropriate.
    PermutohedralLattice::filter(&positions[0], 5,
                                 &values[0], 4,
                                 im.width*im.height,
                                 &values[0]);

    // Divide through by the homogeneous coordinate and store the
    // result back to the image
    idx = 0;
    for (int y = 0; y < im.height; y++) {
        for (int x = 0; x < im.width; x++) {
            float w = values[idx*4+3];
            im(x,y,0) = values[idx*4+0]/w;
            im(x,y,1) = values[idx*4+1]/w;
            im(x,y,2) = values[idx*4+2]/w;
            idx++;
        }
    }
}
```

Appendix D

Non-local means source code

This appendix demonstrates how to use the permutohedral lattice or Gaussian kd-tree to perform non-local means denoising. As in the previous appendix, we assume an image type is available that provides a reference to the pixel at x, y in color channel c via `im(x,y,c)`. We use the TNT and Jama linear algebra libraries [41] to compute eigenpatches for dimensionality reduction.

```
#include <tnt/tnt.h>
#include <jama/jama_eig.h>

// A random integer in the range [min, max)
int randomInt(int min, int max) {
    return (rand() % (max - min)) + min;
}

// Denoise an image using non-local means, including dimensionality
// reduction of patch-space with PCA.
void nlmeans(Image im, float spatialSigma, float patchSigma,
             int patchDimensions, float patchMaskSigma) {

    // Our patches are weighted with a Gaussian mask of standard
    // deviation patchMaskSigma. Their footprint is three times this
    // in each direction, in order to extend to three standard
    // deviations.
    int patchRadius = (int)roundf(patchMaskSigma*3);
    int patchSize = (patchRadius*2+1)*(patchRadius*2+1);
```

```
// Compute the Gaussian mask
vector<float> mask(patchSize);
int idx = 0;
for (int y = -patchRadius; y <= patchRadius; y++) {
    for (int x = -patchRadius; x <= patchRadius; x++) {
        float distance = x*x + y*y;
        mask[idx] = expf(-distance/(2*patchMaskSigma*patchMaskSigma));
        idx++;
    }
}

// Storage for a patch with three color channels
vector<float> patch(patchSize*3);

// To perform dimensionality-reduction with PCA, first we need to
// compute the covariance matrix of the patches in this image. We
// use the TNT and JAMA linear algebra libraries for this
// (http://math.nist.gov/tnt/documentation.html).
TNT::Array2D<double> covariance(patch.size(), patch.size(), 0.0);
TNT::Array1D<double> mean(patch.size(), 0.0);

// Randomly sample 10000 patches from the input image in order to
// gather covariance statistics.
const int patchSamples = 10000;
for (int iter = 0; iter < patchSamples; iter++) {

    // Don't consider patches near the boundary, to avoid dealing
    // with boundary conditions.
    int px = randomInt(patchRadius, im.width-patchRadius);
    int py = randomInt(patchRadius, im.height-patchRadius);

    // Extract a Gaussian-weighted patch
    idx = 0;
    for (int y = py-patchRadius; y <= py+patchRadius; y++) {
        for (int x = px-patchRadius; x <= px+patchRadius; x++) {
            patch[idx*3+0] = mask[idx]*im(x,y,0);
            patch[idx*3+1] = mask[idx]*im(x,y,1);
            patch[idx*3+2] = mask[idx]*im(x,y,2);
            idx++;
        }
    }
```

```
        // Add its statistics to the running totals
        for (int i = 0; i < (int)patch.size(); i++) {
            mean[i] += patch[i];
            for (int j = 0; j < (int)patch.size(); j++) {
                covariance[i][j] += patch[i]*patch[j];
            }
        }
    }

    // Subtract the mean and normalize the covariance matrix
    for (int i = 0; i < (int)patch.size(); i++) {
        for (int j = 0; j < (int)patch.size(); j++) {
            covariance[i][j] -= mean[i]*mean[j]/patchSamples;
            covariance[i][j] /= patchSamples;
        }
    }

    // Now compute the eigenvectors of the patch covariance matrix
    JAMA::Eigenvalue<double> eig(covariance);
    TNT::Array2D<double> eigenvectors(patch.size(), patch.size());
    eig.getV(eigenvectors);

    // Allocate storage for the position vectors. They have
    // patchDimensions patch-similarity terms, and 2 spatial terms.
    vector<float> positions(im.width*im.height*(patchDimensions+2));

    // We can now compute the position vectors by projecting each
    // Gaussian-weighted patch onto the basis provided by the
    // eigenvectors
    int posIdx = 0;
    for (int y = 0; y < im.height; y++) {
        for (int x = 0; x < im.width; x++) {

            // Add the spatial terms first
            positions[posIdx*(patchDimensions+2)+0] = x/spatialSigma;
            positions[posIdx*(patchDimensions+2)+1] = y/spatialSigma;

            // Initialize the patch terms to zero
            for (int i = 0; i < patchDimensions; i++) {
                positions[posIdx*(patchDimensions+2)+2+i] = 0.0f;
            }

            // Gather the Gaussian-weighted patch at this location and
```

```
            // construct the patch terms
            int patchIdx = 0;
            for (int dy = -patchRadius; dy <= patchRadius; dy++) {
                for (int dx = -patchRadius; dx <= patchRadius; dx++) {

                    // Use clamping for boundaries
                    int imx = x+dx, imy = y+dy;
                    if (imx < 0) imx = 0;
                    if (imx > im.width-1) imx = im.width-1;
                    if (imy < 0) imy = 0;
                    if (imy > im.height-1) imy = im.height-1;

                    float r = im(imx, imy, 0);
                    float g = im(imx, imy, 1);
                    float b = im(imx, imy, 2);

                    for (int i = 0; i < patchDimensions; i++) {
                        positions[posIdx*(patchDimensions+2)+2+i] +=
                            eigenvectors[patchIdx*3+0][i]*r +
                            eigenvectors[patchIdx*3+1][i]*g +
                            eigenvectors[patchIdx*3+2][i]*b;
                    }
                    patchIdx++;
                }
            }

            // Scale the patch terms by the patch standard deviation
            for (int i = 0; i < patchDimensions; i++) {
                positions[posIdx*(patchDimensions+2)+2+i] /= patchSigma;
            }

            posIdx++;
        }
    }

    // We now have our position vectors. Next, construct the value
    // vectors.
    vector<float> values(im.width*im.height*4);
    idx = 0;
    for (int y = 0; y < im.height; y++) {
        for (int x = 0; x < im.width; x++) {
            values[idx*4+0] = im(x,y,0);
            values[idx*4+1] = im(x,y,1);
```

```
            values[idx*4+2] = im(x,y,2);
            values[idx*4+3] = 1.0f;
            idx++;
        }
    }

    // Now we perform the Gauss transform. We choose an algorithm
    // based on dimensionality.
    if (patchDimensions+2 >= 12) {
        GKDTree::filter(&positions[0], patchDimensions+2,
                        &values[0], 4,
                        im.width*im.height,
                        &values[0]);
    } else {
        PermutohedralLattice::filter(&positions[0], patchDimensions+2,
                                     &values[0], 4,
                                     im.width*im.height,
                                     &values[0]);
    }

    // Finally, divide through by the homogeneous coordinate and store
    // the output.
    idx = 0;
    for (int y = 0; y < im.height; y++) {
        for (int x = 0; x < im.width; x++) {
            float w = values[idx*4+3];
            im(x,y,0) = values[idx*4+0]/w;
            im(x,y,1) = values[idx*4+1]/w;
            im(x,y,2) = values[idx*4+2]/w;
            idx++;
        }
    }
}
```

Bibliography

[1] A. Adams, J. Baek, and M. A. Davis. Fast high-dimensional filtering using the permutohedral lattice. In *Computer Graphics Forum (Proc. Eurographics)*, 2010.

[2] A. Adams, N. Gelfand, J. Dolson, and M. Levoy. Gaussian kd-trees for fast high-dimensional filtering. In *ACM Transactions on Graphics (Proc. SIGGRAPH 09)*, pages 1–12, 2009.

[3] F. Amat, F. Moussavi, L. Comolli, G. Elidan, K. Downing, and M. Horowitz. Markov random field based automatic image alignment for electron tomography. *Journal of Structural Biology*, pages 260–275, March 2008.

[4] X. An and F. Pellacini. AppProp: all-pairs appearance-space edit propagation. In *ACM Transactions on Graphics (Proc. SIGGRAPH 08)*, pages 1–9, 2008.

[5] S. Arya, D. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45(6), 1998.

[6] V. Aurich and J. Weule. Non-linear Gaussian filters performing edge preserving diffusion. In *Mustererkennung 1995, 17. DAGM-Symposium*, pages 538–545, 1995.

[7] S. Bae, S. Paris, and F. Durand. Two-scale tone management for photographic look. In *ACM Transactions on Graphics (Proc. SIGGRAPH 06)*, pages 637–645, 2006.

[8] P. Bekaert, M. Sbert, and Y. D. Willems. Weighted importance sampling techniques for monte carlo radiosity. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 35–46, London, UK, 2000. Springer-Verlag.

[9] E. P. Bennett and L. McMillan. Video enhancement using per-pixel virtual exposures. In *ACM Transactions on Graphics (Proc. SIGGRAPH 05)*, pages 845–852, 2005.

[10] M. Broadie and Y. Yamamoto. Application of the fast Gauss transform to option pricing. In *Management Science*, page 1071, 2003.

[11] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert. High accuracy optical flow estimation based on a theory for warping. In *Proc. ECCV*, pages 25–36, 2004.

[12] T. Brox, O. Kleinschmidt, and D. Cremers. Efficient nonlocal means for denoising of textural patterns. *Image Processing, IEEE Transactions on*, 17(7):1083–1092, July 2008.

[13] A. Buades, B. Coll, and J.-M. Morel. A non-local algorithm for image denoising. In *Proc. CVPR*, pages 60–65 vol. 2, 2005.

[14] I. Buck. GPU computing: Programming a massively parallel processor. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, page 17, Washington, DC, USA, 2007. IEEE Computer Society.

[15] J. Chen, S. Paris, and F. Durand. Real-time edge-aware image processing with the bilateral grid. In *ACM Transactions on Graphics (Proc. SIGGRAPH 07)*, page 103, 2007.

[16] P. Choudhury and J. Tumblin. The trilateral filter for high contrast images and meshes. In *Eurographics Symposium on Rendering*, pages 186–196, 2003.

[17] J. Conway and N. Sloane. *Sphere Packings, Lattice and Groups*. Springer-Verlag, 3rd edition, 1999.

[18] C. Cortes and V. Capnik. Support-vector networks. In *Machine Learning*, volume 20, pages 273–297, 1995.

[19] J. Darbon, A. Cunha, T. F. Chan, S. Osher, and G. J. Jensen. Fast nonlocal filtering applied to electron cryomicroscopy. In *Proc. ISBI*, pages 1331–1334, 2008.

[20] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *ACM Symposium on Theory of Computing*, 2008.

[21] D.Barash. A fundamental relationship between bilateral filtering, adaptive smoothing and the nonlinear diffusion equation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(6):844–847, 2002.

[22] F. Durand and J. Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. In *Proc. SIGGRAPH 02*, pages 257–266, 2002.

[23] E. Eisemann and F. Durand. Flash photography enhancement via intrinsic relighting. In *ACM Transactions on Graphics (Proc. SIGGRAPH 04)*, pages 673–678, 2004.

[24] A. Elgammal, R. Duraiswami, and L. S. Davis. Efficient non-parametric adaptive color modeling using fast Gauss transform. In *Proc. CVPR*, 2001.

[25] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. In *Theoretical Computer Science*, volume 38, pages 293–306.

[26] A. Gray, G. Boyer, R. Riegel, N. Vasiloglou, D. Lee, L. Poorman, C. Mappus, N. Mehta, H. Ouyang, P. Ram, L. Tran, and W. C. Wong. The mlpack library version 0.2, 2009. http://mloss.org/software/view/152/.

[27] L. Greengard. *The rapid evaluation of potential fields in particle systems*. 1988.

[28] L. Greengard. Fast algorithms for classical physics. In *Science*, pages 909–914, 1994.

[29] T. Jones, F. Durand, and M. Desbrun. Non-itertive, feature-preserving mesh smoothing. In *ACM Transactions on Graphics (Proc. SIGGRAPH 03)*, volume 24, 2003.

[30] J. Kopf, M. F. Cohen, D. Lischinski, and M. Uyttendaele. Joint bilateral upsampling. In *ACM Transactions on Graphics (Proc. SIGGRAPH 07)*, page 96, 2007.

[31] D. Lee, A. Gray, and A. Moore. Dual-tree fast gauss transforms. In *Proc. NIPS*, pages 747–754, 2005.

[32] M. Mahmoudi and G. Sapiro. Fast image and video denoising via nonlocal means of similar neighborhoods. *IEEE Signal Processing Letters*, pages 839–842, 2005.

[33] V. I. Morariu. The figtree library version 0.9.3, 2010. http://www.umiacs.umd.edu/~morariu/figtree/.

[34] V. I. Morariu, B. V. Srinivasan, V. C. Raykar, R. Duraiswami, and L. S. Davis. Automatic online tuning for fast gaussian summation. In *Advances in Neural Information Processing Systems (NIPS)*, 2008.

[35] S. Paris and F. Durand. A fast approximation of the bilateral filter using a signal processing approach. In *Proc. ECCV*, pages 568–580, 2006.

[36] S. Paris and F. Durand. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision*, 81:24–52, 2009.

[37] J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. In *Neural Computation*, volume 3, page 246, 1991.

[38] P. Pérez, M. Gangnet, and A. Blake. Poisson image editing. In *ACM Transactions on Graphics (Proc. SIGGRAPH 03)*, pages 313–318, 2003.

[39] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama. Digital photography with flash and no-flash image pairs. In *ACM Transactions on Graphics (Proc. SIGGRAPH 04)*, pages 664–672, 2004.

[40] F. Porikli. Constant time O(1) bilateral filtering. In *Proc. CVPR*, pages 1–8, 2008.

[41] R. Pozo. The template numerical toolkit and jama linear algebra package version 3.0.11, 2008. http://math.nist.gov/tnt/download.html.

[42] S. Smith and J. M. Brady. SUSAN: A new approach to low level image processing. *International Journal of Computer Vision*, 23:45–78, 1997.

[43] T. Tasdizen. Principal components for non-local means denoising. In *Proc. ICIP*, pages 1728–1731, 2008.

[44] J. Telleen, A. Sullivan, J. Yee, O. Wang, P. Gunawardane, I. Collins, and J. Davis. Synthetic shutter speed imaging. In *Computer Graphics Forum (Proc. Eurographics)*, 2007.

[45] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proc. ICCV*, page 839, 1998.

[46] M. Weber, M. Milch, K. Myszkowski, K. Dmitriev, P. Rokita, and H.-P. Seidel. Spatio-temporal photon density estimation using bilateral filtering. In *Computer Graphics International (CGI 2004)*, pages 120–127. IEEE, 2004.

[47] H. Winnemöller, S. C. Olsen, and B. Gooch. Real-time video abstraction. In *ACM Transactions on Graphics (Proc. SIGGRAPH 06)*, pages 1221–1226, 2006.

[48] J. Xiao, H. Cheng, H. Sawhney, C. Rao, and M. Isnardi. Bilateral filtering-based optical flow estimation with occlusion detection. In *Proc. ECCV*, pages 211–224, 2006.

[49] C. Yang, R. Duraiswami, N. A. Gumerov, and L. Davis. Improved fast Gauss transform and efficient kernel density estimation. In *Proc. ICCV*, pages 664–671 vol.1, 2003.

[50] Q. Yang, K.-H. Tan, and N. Ahuja. Real-time O(1) bilateral filtering. In *Proc. CVPR*, pages 557–564, 2009.