Reshuffling: A Fast Algorithm for Filtering with Arbitrary Kernels

Fatih Porikli

ABSTRACT

A novel method to accelerate the application of linear filters that have multiple identical coefficients on arbitrary kernels is presented. Such filters, including Gabor filters, gray level morphological operators, volume smoothing functions, etc., are widely used in many computer vision tasks. By taking advantage of the overlapping area between the kernels of the neighboring points, the reshuffling technique prevents from the redundant multiplications when the filter response is computed. It finds a set of unique coefficients, constructs a set of relative links for each coefficient, and then sweeps through the input data by accumulating the responses at each point while applying the coefficients using their relative links. Dual solutions, single input access and single output access, that achieve 40% performance improvement are provided. In addition to computational advantage, this method keeps a minimal memory imprint, which makes it an ideal method for embedded platforms. The effects of quantization, kernel size, and symmetry on the computational savings are discussed. Results prove that the reshuffling is superior to the conventional approach.

Keywords: Linear filtering, Arbitrary kernels

1. INTRODUCTION

A kernel filter works by applying a matrix to every point in the given data. It is constituted by a kernel and a filter function defined on this kernel. The kernel specifies the region of support, which is often a rectangular area in 2D images. The matrix coefficients are multiplication factors that determine the contribution of each point within the kernel to the final response. Once all the coefficients have been multiplied by the underlying data, the response at the current point is computed by taking the sum of the products. By choosing different kernel functions, different types of filtering can be performed.

Fast realization of kernel filters has been an important research area recently. Conventional approaches try to break down the original kernel filtering into the convolution of smaller kernels, for instance, using scale-scale representations.[1] For a given image, the linear scale-space representation is a family of derived signals defined by convolution of image with the Gaussian kernel by simultaneously smoothing and subsampling a given signal. In this way, computationally efficient algorithms can be obtained. Still, significant amount of multiplications are needed to carry out all of the convolutions with the large set of small kernels.

It is also possible to design a 2D filter kernel and then decompose it into a sum of separable one dimensional filters or cascaded representations.[2–4] This can be done by using of either an eigenvalue expansion of the 2D kernel or application of Singular Value Decomposition.[5] By taking advantage of the separability of the 2D Gaussian function, Geusebroek *et al.*[6] decomposed an anisotropic (directionally dependent) Gaussian kernel into a one-dimensional Gaussian filter followed by another filter in a nonorthogonal direction. This method allows fast calculation of edge and ridge maps using convolution and recursive schemes. Shen *et al.*[7] proposed sum-box filter technique to approximately realize a given large kernel linear filter to a factor by the sum of the translated outputs of sum-box filters, requiring no multiplications by use of the analysis on the scaled Spline functions. Their method makes it possible to achieve convolution with a large filter kernel by additions only. Box technique can realize Gaussian filters with multiplications, but it cannot realize linear filters other than Gaussians. Tang *et al.*[8] proposed a fast multidimensional image filtering approach based on a fuzzy domain enhancement method, and an implementation of a recursive and separable low-pass filter. They process each pixel independently and modifies both dynamic range and local variations. Unfortunately, this method is only applicable to Gaussian functions. Fischl and Schwartz[9] proposed a computationally efficient algorithm to solve nonlinear PDE's based on an adaptively determined vector field specifying nonlocal application points. Even though their method provides speed up for image enhancement, it is not applicable to linear filters.

Fast filtering algorithms are also described for coding and decoding. The encoding process is equivalent to sampling the image with Laplacian operators of many scales. Most algorithms use orientation estimation for adaptation of local

Send correspondence to F. Porikli, E-mail: fatih@merl.com

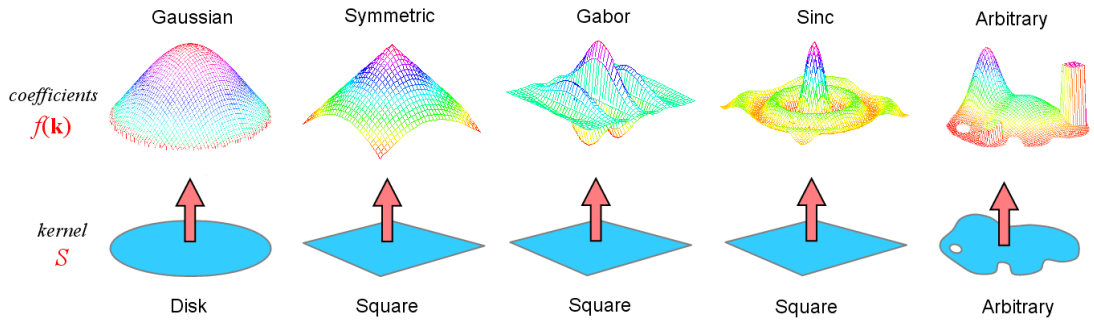


Figure 1. Several kernel filters; Gaussian, symmetric (which is also a superset of Gaussian filters), Gabor, sinc (low-pass), and an arbitrary function. Symmetrical filter functions are defined within a disk shaped kernel, however, for implementation simplicity, often rectangular kernels are used in practice. One example of arbitrary shaped kernels is template enhancing gray level morphological filter.

spatial texture. This often implies a sampling of orientations by anisotropic filtering. For a linear orientation scale-space, the anisotropic Gaussian is the best suited causal filter. Orientation analysis is often approached by a set of streerable anisotropic filters. Freeman and Adelson[10] put forward the conditions under which a filter can be tuned to a specific orientation by making a linear combination of basis filters. According to their framework, no exact basis exists for rotating an anisotropic Gaussian. Perona[11] derived a scheme for generating a finite basis which approximates an anisotropic Gaussian. However, the number of basis filters is large, and the basis filters are nonseparable, requiring high computational performance.

There are other methods that use reconfigurable hardware[12] or processor adapted software libraries such as Intel's IPL[13] to accelerate the filtering process. GPU implementations of image filtering methods became also popular, especially for the computation of area sums using pyramid representations.[14] A GPU-based filtering of images with cubic B-splines was presented by Sigg and Hadwiger[15] that exploits bilinear texture interpolations. While this technique allows for random access to a texture image, it requires considerably more texture lookups per pixel. Strengert *et al.*[16] show that modern GPUs allow implementation of pyramid methods based on bilinear texture interpolation, and present three examples: zooming with biquadratic B-spline filtering, efficient image blurring of arbitrary blur width, and smooth interpolation of scattered pixel data.

In summary, most fast approaches either strictly depend on the shape of the filter function or provide minor computational improvements. Here, we propose a fast algorithm that significantly accelerates the filtering process and is not limited to any filter shape. We take advantage of spatial arrangement of the filter coefficients and overlapping between the kernels of neighboring points to prevent from redundant multiplications. Our method finds a set of unique filter coefficients, extracts a set of relative links for each unique coefficient, and then process the input data by accumulating the responses at each point while applying the unique coefficients according to their relative links. In addition to computational advantage, this method also keeps a minimal memory imprint.

In the following section, we introduce the concept of the reshuffling. Then, we present pseudo-code for dual implementations, and give a detailed computational complexity analysis for several commonly used image filters including the ones shown in Fig 1.

2. RESHUFFLING

Suppose our kernel filter is defined in a d-dimensional real valued Cartesian space $\mathcal{R}^d$. The filter maps the data I within its kernel S centered around the point $\mathbf{p} = [x_1, ...x_d]$ in the space to a m-dimensional response vector $I(\mathbf{p}; S) \rightarrow y(\mathbf{p}) = [y_1, ..., y_m]$. Lets consider only the filters that maps to a scalar, that is $m = 1$, thus $y(\mathbf{p}) = y_1$ without loss of generality. The filter assigns a real valued coefficient $f(\mathbf{p})$ to each of the points $\mathbf{p}$ in its kernel S. Lets assume the data to be filtered is bounded within the range $N_1, ..., N_d$, i.e. $0 \leq x_i < N_i$. Note that, single channel image filters are defined in $d = 2$, color and video filters are defined in $d = 3$ as the temporal component constitutes the third dimension. Given this notation, the kernel filter response can be expressed as

$$y(\mathbf{p}) = \sum_{\mathbf{k} \in S} f(\mathbf{k}) I(\mathbf{k} + \mathbf{p}) \tag{1}$$

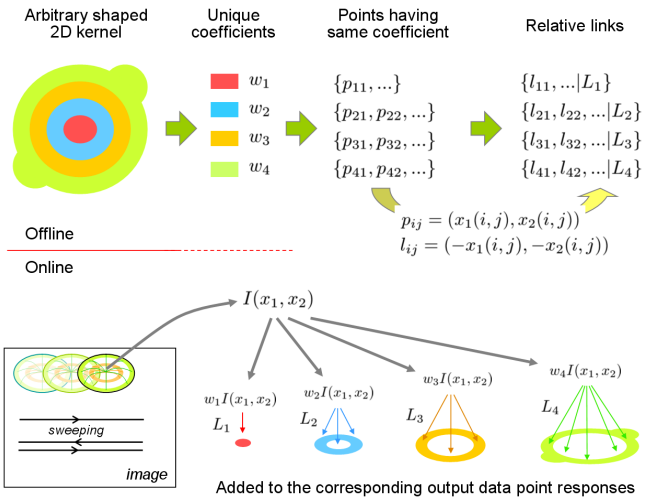


Figure 2. Instead of multiplying filter matrix with data, the reshuffling selects the unique coefficients, then multiplies the input pixel value once for each unique coefficient while adding the results to the corresponding output pixels using the relative links.

which is same as the correlation. Since the response can be computed by a bounded set of coefficients in a single pass, kernel filters are also Finite Impulse Response (FIR) filters. For traditional 2D image filtering with a square kernel bounded within $[-K_1/2, K_1/2], [-K_2/2, K_2/2]$, the response becomes

$$y(x_1, x_2) = \sum_{k_1=-K_1/2}^{K_1/2} \sum_{k_2=-K_2/2}^{K_2/2} f(k_1, k_2) I(x_1 + k_1, x_2 + k_2)$$

and it is straightforward to modify it for convolution by changing the signs. We give a pseudo-code for the conventional approach below.

Algorithm 1 Conventional 2D Kernel Filter

```
for all 0 ≤ x1 < N1 do
    for all 0 ≤ x2 < N2 do
        sum ⇐ 0
        for all −K1/2 ≤ k1 ≤ K1/2 do
            c ⇐ x1 + k1
            for all −K2/2 ≤ k2 ≤ K2/2 do
                if (k1, k2) ∈ S then
                    sum ⇐ sum + f(k1, k2) · I(c, x2 + k2)
                end if
            end for
        end for
        Y(x1, x2) ⇐ Y(x1, x2)/sum
    end for
end for
```

Even though it is tempting to implement the above formulation with no changes, we emphasize the fact that, it also causes the multiplication of the pixel value with the same coefficients over and over again in case the kernel contains same coefficients (regardless of their position within the kernel). Thus, the reshuffling first finds the unique coefficients $w_i, = i = 1, ..., U$ as in Fig. 2. Then, it constructs a linkage set L_i for each unique coefficient w_i such that the set includes the relative links of the filter positions that have the same coefficient $L_i = \{l_{i0}, ..., l_{iL_i}\}$. A relative link l_{ij} is the inverse position of the coefficient, i.e., if the filter position is (x_1, x_2) then the relative link is $(-x_2, -x_2)$ with respect to kernel

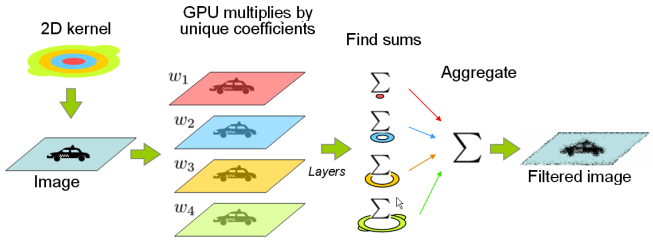


Figure 3. Dual algorithm computes the response for a single output point (single output access.

indexing. The unique coefficients and links are determined once for a kernel filter offline. This process takes almost no time.

Once we obtained the unique coefficients and relative links, then we process the data one input point at a time. We multiply the current value that kernel is centered on by each of the unique coefficient and add the result to the responses of the multiple output points as referenced by the relative links:

$$\begin{aligned} w_1 I(\mathbf{p}) &\rightarrow y(\mathbf{p}+l_{11}),...,y(\mathbf{p}+l_{1L_1}) \\ \vdots &\rightarrow \vdots \\ w_U I(\mathbf{p}) &\rightarrow y(\mathbf{p}+l_{U1}),...,y(\mathbf{p}+l_{UL_U}) \end{aligned} \tag{2}$$

In single input access, the filter response is progressively calculated, and the output is obtained when all the points in the kernel are processed as shown below.

Algorithm 2 Reshuffling - Single Input Access

(Offline) Find Unique Coefficients $w_i, i = 1,..,U$
(Offline) Extract Relative Links l_{ij}
for all $1 \leq x_1 < N_1$ **do**
  for all $1 \leq x_2 < N_2$ **do**
    $multiplier \Leftarrow I(x_1, x_2)$
    for all $1 \leq i \leq U$ **do**
      $value \Leftarrow multiplier \cdot w_i$
      for all l_{ij} **do**
        $k_1 \Leftarrow k_1 + l_{ij}, k_2 \Leftarrow k_2 + l_{ij}$
        $y(k_1, k_2) \Leftarrow y(k_1, k_2) + value$
      end for
    end for
  end for
end for

This algorithm is suitable for stream processors where the local cache size is limited, thus not to much data is transmitted over the local bus.

We also propose a dual algorithm where a kernel sized block of input is accessed every time. As above, we find the unique coefficients and the relative link sets. Dual algorithm takes the inputs within the current kernel and finds the summation of the points for each relative link set as illustrated in Fig. 3. Then, it multiplies these summations with the corresponding unique coefficients to assign the value of the output point.

$$y(\mathbf{p}) = \sum_{i}^{U} \left(w_i \sum_{j}^{L_j} I(\mathbf{p}+l_{ij}) \right) \tag{3}$$

Operator	A	B	C	D	Our
Integer addition	1	1	1	1	1
Integer multiply	4	4	24	4	1
Float addition	20	3	4.2	4	15
Float multiply	20	5	113	4	15
Logical operator	1	-	-	2	1
Array indexing	$6d-4$	-	-	$7d-3$	$2d-1$

Table 1. Column-A is the relative cost of the basic processor operators as given in.[17] Column-B is the cost of the operators executed on a P4 processor that uses streaming SIMD and Prescott arithmetic operations.[19] Column-C is the relative costs on a P4 running C++ compiler.[18] Column-D is reported on a P4 running MSVC++ compiler.[20] Our experiments estimate the costs on a Intel Core Duo as in the last column.

In this manner, the filter response for a single output point is computed. Since the image is multiplied once for each unique coefficients, this algorithm can take advantage of GPUs to further accelerate the filtering. A pseudo-code for the single output access is given below.

Algorithm 3 Reshuffling - Single Output Access

```
(Offline) Find Unique Coefficients w_i, i = 1, .., U
(Offline) Extract Relative Links l_ij
for all 1 ≤ x_1 < N_1 do
    for all 1 ≤ x_2 < N_2 do
        for all 1 ≤ i ≤ U do
            sum ⇐ 0
            for all l_ij do
                k_1 ⇐ k_1 + l_ij, k_2 ⇐ k_2 + l_ij
                sum ⇐ sum + I(k_1, k_2)
            end for
            y(k_1, k_2) ⇐ y(k_1, k_2) + w_i · sum
        end for
    end for
end for
```

3. COMPUTATIONAL IMPROVEMENTS

We performed a detailed computational complexity analysis in terms of the relative cost of processor operations, which is usually measured against the cost of an integer addition operation. In the Table 1, we present the relative costs of the basic operations reported in the literature as well as our own results. Since the cost of the array indexing becomes comparable for higher dimensional data, we also consider the cost of array indexing. For an d-dimensional array, accessing data requires d integer additions, $d-1$ integer multiplications. We observed that due to hardware integrated use of MAC's (multiplication-accumulation unit) on recent CPU's, the costs of the addition and multiplication remain equal for the same data types.

Suppose the input data has floating point values and total number of points in the kernel is A, i.e. for a rectangular kernel $A = \prod_i^d K_i$. The conventional kernel filtering algorithm requires the following tasks to compute the filter response:

- Find indices of current points: A d-dimensional array indexing and $2A$ additions,
- Check whether current point is in the kernel: d-dimensional array indexing and comparison,
- Multiply by filter coefficients: A multiplications,
- Assign sum of multiplications: $A-1$ additions and d-dimensional array indexing.

For a single data point, the conventional filtering requires $2A + (2d-1)A$ operations for computing the indices of the contributing points within the kernel in data space and accessing their values respectively. In case of using a non-rectangular kernel, there are two possible control conditions to determine whether a point belongs to the kernel or not; either using a larger fitting rectangular array and a binary mask representing the kernel membership, or computing a distance based criterion from index values. We observed from our experiments that using a mask sustains less computational load. The checking stage requires $(2d-1)A + A = 2dA$ operations.

The floating-point value of each data point within the kernel is multiplied by the corresponding filter coefficient, resulting in $15A$ operations. Computing the floating-point sum of multiplications takes $15(A-1)$ operations, while assigning the sum as the filter response takes only $2d-1$. Note that the previous computations are repeated for each of the $N_1 \times .. \times N_d$ points in the data. Then, the total number of operations needed for all candidates becomes

$$[(4d+31)A + 2d - 16] \prod_{j}^{d} N_j \tag{4}$$

The reshuffling, on the other hand, do not repeat the multiplication with the same coefficients and do not need a checking conditional for arbitrary shaped kernels. Before the application of the kernel filter, it preprocess the filter to determine the unique coefficients and constructs sets of relative links for each unique coefficient. Note that, this is done only once for a filter, that is, it is not duplicated in the filtering process. After this off-line process, the reshuffling requires these tasks:

- Get current point value: d-dimensional array indexing,
- Multiply by unique coefficients: U multiplications,
- Find indices of points: $\sum L_i$ d-dimensional array indexing and $2\sum L_i$ additions,
- Increment point values: $\sum L_i$ additions.

At each data point, the reshuffling employs $2d-1$ operations to get the value of the current data point. Then, it multiplies the floating-point value by the each of the unique coefficients, which takes $15U$ operations. Since the total number of the relative links is equal to total number of points in the kernel $\sum_i^U L_i = A$, finding indices of the points within the kernel requires $2A$ integer additions for converting relative links to the data point indices and A array indexing, which results in $(2d+1)A$ operations. As a result, the computational load of this stage is $(2d+1)A$ operations.

The corresponding multiplication results are added to the filter responses of output array data points. This involves $15A$ operations for floating-point additions and $(2d-1)A$ operations for accessing the output array. The total number of operations then becomes:

$$[(4d+15)A + 2d - 1 + 15U] \prod_{j}^{d} N_j \tag{5}$$

We define a reduction ratio that corresponds to the percentage of computational savings accomplished by the reshuffling as

$$R = \left(1 - \frac{(4d+15)A + 2d - 1 + 15U}{(4d+31)A + 2d - 16}\right) \times 100 \tag{6}$$

This number represents how many percent of the computations is prevented by using the minimalist filtering. We also compute a redundancy score as

$$\delta = (1 - U/A) \times 100 \tag{7}$$

to represent the ration of unique coefficients to all.

Effect of Quantization, Symmetry & Separability:

The success of the reshuffling comes from the smaller number of unique coefficients. In case this number is big, i.e. equal to total number of points in the kernel $U = A$, the reshuffling reduces to the conventional filtering. However, almost all of the kernel filters have very small unique coefficient numbers.

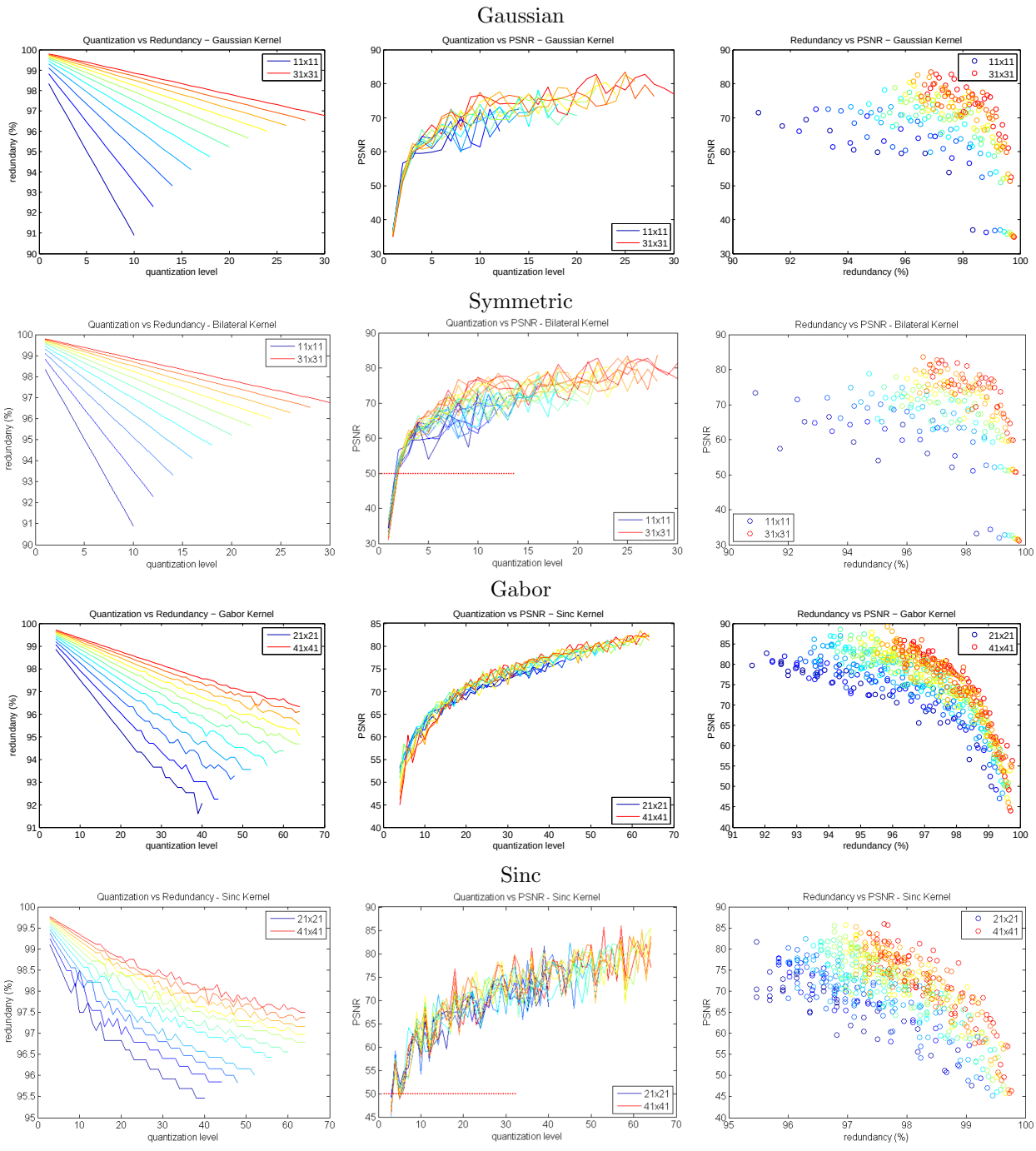


Figure 4. Detailed performance graphs for different 2D kernel sizes and different filter functions. **First** column shows how the redundancy in the number of coefficients are affected with respect to the quantization level changes. The input data has always had 256 levels. Note that, as the kernel size becomes larger, the redundancy increases proportionally. **Second** column shows the average PSNRs for quantization levels. As visible, any quantization level above 5 generates filtered results above the noticeable threshold, which is 50dB. In other words, 5-level quantization is sufficient to represent filter functions. **Last** column shows the performance of the reshuffling as the redundancy changes. As visible, more than 98% redundancy can be achieved for all filter functions while having no noticeable difference between the filtered results. These graphs prove that the reshuffling provide almost 40% without any significant loss of information. (Same color corresponds to same kernel size in above graphs).

Figure 5. An example of symmetric filtering by 0% redundancy using the conventional method (left) and 98% redundancy using the reshuffling (right).

Gaussian	Symmetric	Gabor	Sinc	Edge
40.3%	40.1%	38.2%	37.7%	41.5%

Table 2. Average computational savings at 50dB for various 2D filters on 21×21 kernels.

Quantizing is a competent way of obtaining a smaller number of unique coefficients. For the 2D kernel filters that are symmetric around the kernel origin, the minimum redundancy score is 75%. In other words, even without any quantization, the reshuffling method still significantly reduces the required computations to a quarter of the original load.

To provide a throughout analysis, we tested widely used filters; Gaussian, symmetric, Gabor, sinc, and edge. For each filter, we applied both the original filter and the corresponding reshuffling with varying levels of quantizations. We used 100 standard test images with 256 gray levels. We also assessed the PSNR that no noticeable difference is observed between both quantised and original filter reosposes. We set the threshold to 50dB, which is much higner than human visual systems perception rate 40dB. As in Fig 5, there is no visible difference at 50dB filtering by 13×13 symmetric kernel. For each image, we applied the filters.

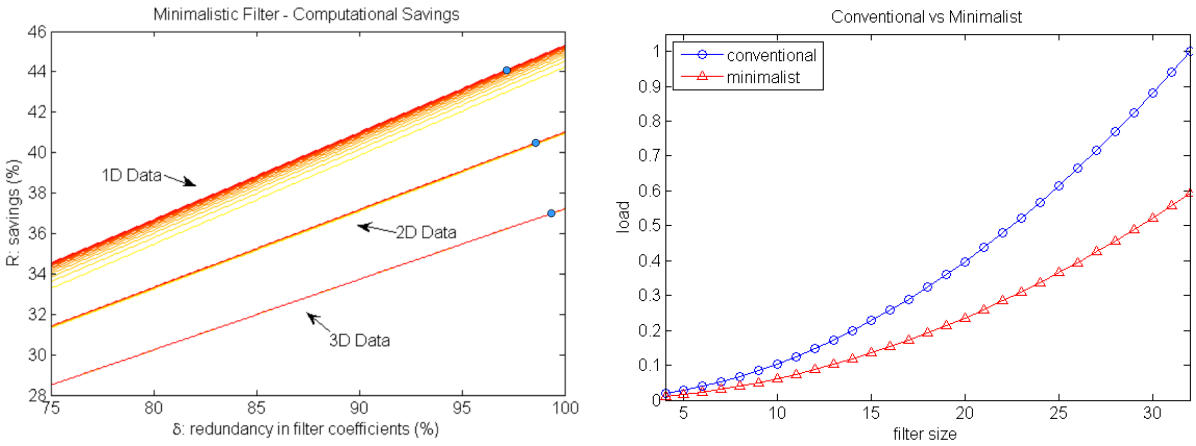
Figure 4 shows detailed performance graphs for different 2D kernel sizes. First column shows how the redundancy in the number of coefficients are affected with respect to the quantization level changes. We observed that as the kernel size becomes larger, the redundancy increases proportionally. To our surprise, any quantization level above 5 generated results above the noticeable threshold. In other words, 5-level quantization, which corresponds to around 98%-99% redundancy and 40% computational savings, is sufficient to represent filter functions. Last column proves this observation. This analysis showed that the reshuffling provide almost 40% improvement without any significant loss of information.

We also analyzed how the performance changes in different dimensional data. As given in Fig. 6-a, the savings increases as dimensionality decreases, partially a result of the array accessing costs. Note that, the savings are much higher for lower dimensional data, in other words, using lower dimensional separable kernels, the reshuffling provides even much higher gains. In addition, for higher dimensions, the savings becomes independent from filter size. This figure also shows that for separable filters, the improvements will be even higher.

Furthermore, we compared 2D kernel filtering as presented in Fig. **??**-b. We observed that as the kernel size increases, the reshuffling method becomes more advantageous. We summarized the average gains for various filters as our experiments concluded in Table 2.

4. CONCLUSION

We presented a novel method, the reshuffling, to accelerate the application of kernel filters that have multiple identical coefficients. Our contribution takes advantage of the overlap between the kernels to prevent from the redundant multiplications. Here is a summary of the advantages of the reshuffling:



(a) (b)

Figure 6. (a) Computational savings increases as the number of redundant coefficients increases. Blue dots show the average gains 43%, 40.5%, 37% without any noticable change in the outputs. Different colors indicate different kernel sizes. (b) Computational load comparison of conventional and reshuffling methods for 2D data.

- It minimizes the computational load of any kernel filter 40% on average. As opposed to the conventional methods, the computational superiority is not limited to certain filter functions. It can use any arbitrary shaped kernel with no additional cost, which is an essential property for template detection tasks such as shape matching.
- As our analysis shows, it is applicable to any dimensional data *e.g.*for volumetric smoothing or multi-spectral filtering.
- It has a small memory imprint, thus, it is a perfect candidate for embedded implementations. It can be easily implemented in a parallel or on specific hardware such as GPU's.

We emphasize that any existing fast methods based on pyramid representations and lower dimensional separable basis can benefit from the reshuffling, which is a promising future direction.

REFERENCES

[1] P. Burt, "Fast filter transforms for image processing", *Computer Vision, Graphics and Image Processing*, vol.16, 20-51, 1981.

[2] J. Crowley, R. Stern, "Fast computation of the difference of low-pass transform", *IEEE Trans. Pattern Anal. Machine Intell.*, vol.6, 212-222, 1984.

[3] P. Heckbert, "Filtering by repeated integration", *ACM SIGGRAPH Computer Graphics*, vol.20:4, 315-321, 1986.

[4] W. Wells, "Efficient synthesis of Gaussian filters by cascaded uniform filters", *IEEE Trans. Pattern Anal. Machine Intell.*, vol.8:2, 234-239, 1986.

[5] W. S. Lu, H. P. Wang, A. Antoniou, "Design of 2D FIR digital filters by using the singular value decomposition", *IEEE Trans. Circuits Syst.*, vol.37, 35-36, 1990.

[6] J. M. Geusebroek, A. Smeulders, J. Weijer, "Fast anisotropic Gauss filtering", *IEEE Transaction on Image Processing*, vol.12:8, 2003.

[7] J. Shen, W. Shen, S. Castan, T. Zhang, "Sum-box technique for fast linear filtering", *Signal Process.*, vol.82:8, 1109-1126, 2002.

[8] H. Tang, T. Zhuang, E. X. Wu, "Realizations of fast 2-D/3-D image filtering and enhancement", *IEEE Trans Medical Imaging*, vol.20:2, 32-40, 2001.

[9] B. Fischl, E. L. Schwartz, "Adaptive nonlocal filtering: A fast alternative to anisotropic diffusion for image enhancement", *IEEE Trans. Pattern Anal. Machine Intell.*, vol.21:1, 42-48, 1999.

[10] W. T. Freeman, E. H. Adelson, "The design and use of steerable filters", *IEEE Trans. Pattern Anal. Machine Intell.*, vol.13, 891-906, 1991.

[11] P. Perona, "Steerable-scalable kernels for edge detection and junction analysis", *Image Vis. Comput.*, vol.10, 663-672, 1992.

[12] J. Torresen, J. W. Bakke, L. Sekanina, "Efficient image filtering and information reduction in reconfigurable logic", *In Proc. 22nd Norchip Conference*, 2004.

[13] F. Vogt, D. Paulus, C. H. Schick, “Fast implementations of temporal color image filtering”, *In Proc. 7th Workshop Farbbildverarbeitung*, 89-98, 2001.

[14] J. Kruger, R. Westermann, “Linear algebra operators for gpu implementation of numerical algorithms”, *ACM Transactions on Graphics*, vol.22:3, 908-916, 2003.

[15] C. Sigg, M. Hadwiger, “Fast third-order texture filtering”, *In Matt Pharr, editor, GPU Gems 2: Programming Techniques for High-Performance Graphics*, 313-329, 2005.

[16] M. Strengert, M. Kraus, T. Ertl, “Pyramid Methods in GPU-Based Image Processing”, *In Proc. 11th International Fall Workshop Vision, Modeling, and Visualization*, 2006.

[17] S. Oualline, “Practical C++ programming”, *O’Reilly & Associates, ISBN: 1-56592-139-9*, 1995.

[18] J. Mathew, P. Coddington, K. Hawick, “Analysis and development of java grande benchmarks”, *In Proceedings of ACM*, 1999.

[19] R. Bryant, D. O’Hallaron, “Computer systems: a programmer’s perspective”, *Prentice Hall, ISBN 0-13-034074-1*, 2003.

[20] F. Porikli, “Integral histogram: a fast way to extract histograms in Cartesian spaces”, *In Proc. Computer Vision and Pattern Recognition Conference*, vol.1, 829-836, 2005.