Compressed Domain Processing of JPEG-encoded images[1]

Brian C. Smith, Cornell University
Lawrence A. Rowe, University of California at Berkeley

Abstract

This paper addresses the problem of processing motion-JPEG video data in the compressed domain. The operations covered are those where a pixel in the output image is an arbitrary linear combination of pixels in the input image, which includes convolution, scaling, rotation, translation, morphing, de-interlacing, image composition, and transcoding. This paper further develops an approximation technique called *condensation* to improve performance and evaluates condensations in terms of processing speed and image quality. Using condensation, motion-JPEG video can be processed at near real-time rates on current generation workstations.

1 Introduction

Processing video data is problematic due to the high data rates involved. Television quality video requires approximately 100 GBytes for each hour, or about 27 MBytes for each second. Such data sizes and rates severely stress storage systems and networks and make even the most trivial real-time processing impossible without special purpose hardware. Consequently, most video data is stored in a compressed format.

While compression decreases storage and network costs, it increases processing cost because the data must be decompressed first. The overhead of decompression is enormous: today's sophisticated compression algorithms, such as JPEG or MPEG, require between 150 and 300 instructions per pixel for decompression [1]. This corresponds to a rate of 2.7 billion instructions for each second of NTSC quality video processed. Furthermore, the data must often be compressed after processing, incurring additional overhead.

One way to circumvent these problems is to process the video data in compressed form. This technique reduces the amount of data that must be processed and avoids complex compression and decompression. Decreasing data volume has the side effect of increasing data locality and more effectively using the processor cache, improving performance further.

1.This research was supported by the National Science Foundation under grants DCR-85-07256 and MIP-90-14940.

In a previous paper [2], we showed how to perform scalar and pixel-wise addition or multiplication directly on motion-JPEG video. In this paper, we extend this work to show how a wider class of operations, where each pixel in the output image is a linear combination of several pixels in the input image, can be computed in the compressed domain. Doing so is challenging because such operations often cross JPEG block boundaries. We address this problem by writing the operations as tensors to capture the block structure of the compressed image data. We then show how to express JPEG compression and decompression as tensors, allowing the compressed domain equivalent of the image operator to be computed.

Unfortunately, the resulting operation turns out to be no faster than spatial domain processing. Consequently, we develop an approximation technique called *condensation* the introduces a dead-zone in the compressed domain operator. Condensation dramatically reduces the cost of computing an image, but degrades its quality. This speed/ quality trade-off is studied in section 5. A prototype implementation shows that this technique runs at rates approaches real-time on current generation computers. For example, a smoothing filter can be applied to a 320 by 240 JPEG-encoded image in about 75 milliseconds on a DEC alpha workstation, which is approximately 12 frames per second.

The rest of this paper is organized as follows. Section 2 shows how to express images, JPEG compression, JPEG decompression, and image operations as tensors. In section 3 we combine these tensors to construct a single linear operator that can be applied to compressed video images. Section 4 describes an approximation technique, called *thresholding condensation*, that allows the operators to be efficiently computed. In section 5, we report the results of an experimental implementation of these techniques, in section 6 we discuss applications of this technique, and in section 7 we compare our work with related work and suggest directions for future research.

2 Images, Image Processing, and JPEG as Tensors

Image Representation and Manipulation

A gray scale image, **f**, is conventionally represented as an matrix of pixels $f_{\alpha\beta}$, where α and β specify the row and column position of the pixel. Many operations on images can be expressed using linear combinations of these

pixels. For instance, one way to express a smoothing operation is this: a pixel in the output image **g** is half the value of the corresponding pixel in the input image **f** plus one-eighth the sum of the four neighboring pixels:

$$\mathbf{g}_{\alpha\beta} = \frac{\mathbf{f}_{\alpha-1,\beta} + \mathbf{f}_{\alpha,\beta-1} + \mathbf{f}_{\alpha+1,\beta} + \mathbf{f}_{\alpha,\beta+1}}{8} + \frac{\mathbf{f}_{\alpha,\beta}}{2} \qquad \text{(EQN 1)}$$

or, in the operation of shrinking an image by a factor of two, each pixel in **g** is the average of four pixels in **f**:

$$\mathbf{g}_{\alpha\beta} = \frac{\mathbf{f}_{2\alpha,2\beta} + \mathbf{f}_{2\alpha+1,2\beta} + \mathbf{f}_{2\alpha,2\beta+1} + \mathbf{f}_{2\alpha+1,2\beta+1}}{4} \qquad \text{(EQN 2)}$$

If the coefficients in these operations are gathered in a four dimensional matrix, **T**, the output image **g** is the product of **T** and the input image **f**:

$$\mathbf{g}_{\gamma\delta} = \sum_{\alpha\beta} \mathbf{T}_{\alpha\beta\gamma\delta}\mathbf{f}_{\alpha\beta}$$

For example, in equation 1, **T** is

$$\mathbf{T}_{\alpha\beta\gamma\delta} = \begin{cases} 1/2 & \text{if } \alpha = \gamma \text{ and } \beta = \delta \\ 1/8 & \text{if } \alpha = \gamma \text{ and } \beta = \delta \pm 1 \\ 1/8 & \text{if } \alpha = \gamma \pm 1 \text{ and } \beta = \delta \\ 0 & \text{otherwise} \end{cases}$$

and in equation 2, **T** is

$$\mathbf{T}_{\alpha\beta\gamma\delta} = \begin{cases} 1/4 & \text{if } \gamma = \lfloor \alpha/2 \rfloor \text{ and } \delta = \lfloor \beta/2 \rfloor \\ 0 & \text{otherwise} \end{cases}$$

T is a fourth rank tensor (a four dimensional matrix) that maps a second rank tensor (a two dimensional matrix -- the input image **f**) into another second rank tensor (the output image **f**). We make no assumptions about the structure of **T**, although in practice, **T** is very sparse, since in most image processing operations an output pixel depends on few input pixels. The tensor representation is quite flexible. Since each output pixel is a distinct linear combination of the input pixels, it can capture image processing operations not easily expressed in other formulations. For example, **T** can represent an operator that blurs one part of an image and sharpens another, or an operator that performs different affine transformations on different parts of the image, as in morphing.

Suppose we want to apply **T** directly on a JPEG-encoded image (JPEG is described in the next section). One problem that arises is that **T** may cross block boundaries. To capture this block structure, we represent **f** as a fourth rank tensor f_{xyij}, with $\alpha = 8x+i$ and $\beta = 8y+j$. The first pair of indices, *x* and *y*, specify the block address, and the second pair, *i* and *j*, specify the pixel offset within the block, as shown in figure 1. Images that represented this way are called *block-oriented* images.

To capture block structure in tensor operators, we convert the fourth rank tensor $T_{\alpha\beta\gamma\delta}$ into an eighth rank tensor $T_{xyijwzuv}$, with the correspondence $\alpha = 8x+i$, $\beta = 8y+j$, $\gamma = 8w+u$, and $\delta = 8z+v$. This new tensor maps one block-oriented image to another

$$\mathbf{g_{wzuv}} = \sum_{xyij} \mathbf{T_{xyijwzuv} f_{xyij}}$$

which we can abbreviate **g** = **Tf**.

T is an eighth rank tensor that maps a fourth rank tensor (the input image **f**, in block representation) into another fourth rank tensor (the output image **f**). To understand **T**, the idea of a block transform (BT) is introduced. A BT maps one 8x8 block of pixels to another. To compute a block in an output image **g**, BTs are applied to the each input block in **f** and the transformed blocks are summed pixel-wise. **T** is four dimensional array of BTs, with two indices specifying the output block (w,z), and two specifying the input block (x,y).

For example, consider scaling a 32x16 pixel image **f** to a 16x8 image **g** (a *shrink-by-2* operation), as shown in figure 2. To compute the left block in **g**, 8 BTs are applied to the blocks in **f**. The resulting blocks, shown in the top of the figure, are added pixel-wise to produce the output block. This strategy is repeated to the right block in **g**.

JPEG compression as a tensor

Having shown how to express images and their operators as tensors, we now turn to the task of expressing JPEG compression and decompression as a tensor. To do so, we have to rearrange the steps of the baseline JPEG algorithm slightly. Briefly, JPEG divides an image into 8x8 blocks and applies six steps to each block. Step one applies the discrete cosine transform (DCT) to the block. Step two orders the 64 DCT coefficients into a 64 element vector using a zig-zag scan, a heuristic to place the low frequency coefficients early in the vector. Step three scales

the result by dividing each coefficient by a constant. A different constant is used for each coefficient. These constants are usually arranged in a table, called the *quantization table*. Step four rounds the result to the nearest integer. Step five run length encodes the vector, and step six computes the difference between the DC value of this block and the DC value of the previous block (DPCM) and applies an entropy coding technique (either Huffman or arithmetic) to the result.

In most formulations, steps three and four are taken together and called *quantization*. We split them apart so that the first three steps can be combined into a linear operator, **J**, since the DCT, zig-zag scanning, and scaling are all linear operations.

With the first three steps combined, JPEG compression is the four step process as shown in figure 3. The first step applies the linear transformation **J** to each 8 x 8 pixel block **f** in the input image. The output is a 64 element vector **F**. The second step rounds each element of **F** to the nearest integer. The third step produces a sparse vector representation of **F** (called the *semi-compressed*, or *SC*, vector) using run length encoding, and the final step applies DPCM to the DC component and entropy encodes the SC-vector.

Decompression of a block, also depicted in figure 3, entropy decodes the JPEG bitstream, inverts the DPCM to recover the SC-vector, and applies the linear transformation $\mathbf{J}^{-1}$ to recover an approximation of the original pixel values.

The *JPEG operator* **J** is the composition of three linear operations: 1) a discrete cosine transform (DCT), 2) zig-zag scanning, and 3) scaling. If we write these steps as the tensors **D**, **Z**, and **S**, the **J** is given by

$$\mathbf{J_{ijl}} = \sum_{\mathbf{u,\,v,\,k}} \mathbf{S_{kl} Z_{uvk} D_{ijuv}} \qquad \text{(EQN 3)}$$

D is a fourth rank tensor whose elements are

$$\mathbf{D_{ijuv}} = \frac{1}{4}\mathbf{A(u)A(v)}\cos\frac{(2\mathbf{i}+1)\mathbf{u}\pi}{16}\cos\frac{(2\mathbf{j}+1)\mathbf{b}\pi}{16} \qquad \text{(EQN 4)}$$

with

$$\mathbf{A}(\alpha) = \begin{cases} 1/\sqrt{2} & \text{if } \alpha=0 \\ 0 & \text{otherwise} \end{cases} \quad \text{(EQN 5)}$$

Z is a third rank tensor whose elements are all 0 or 1. It is similar in spirit to a permutation matrix, since its function is to rearrange data. It's elements are

$$\mathbf{Z_{uvk}} = \begin{cases} 1 & \text{if zigzag[u,v]= k} \\ 0 & \text{otherwise} \end{cases} \quad \text{(EQN 6)}$$

and **S** is a diagonal, second rank tensor

$$\mathbf{S_{kl}} = \begin{cases} 1/\mathbf{q}[\mathbf{k}] & \text{if } \mathbf{l} = \mathbf{k} \\ 0 & \text{otherwise} \end{cases} \quad \text{(EQN 7)}$$

where the vector **q** is derived from the JPEG quantization table.

$\mathbf{J^{-1}}$ is the inverse of **J**, and is similarly defined.

$$\mathbf{J^{-1}_{ijl}} = \sum_{\mathbf{u,v,k}} \mathbf{D^{-1}_{uvij}} \mathbf{Z^{-1}_{uvk}} \mathbf{S^{-1}_{kl}} \quad \text{(EQN 8)}$$

where $\mathbf{D^{-1}}$ is the IDCT

$$\mathbf{D^{-1}_{uvij}} = \frac{1}{4}\mathbf{A}(\mathbf{u})\mathbf{A}(\mathbf{v})\cos\frac{(2\mathbf{i}+1)\mathbf{u}\pi}{16}\cos\frac{(2\mathbf{j}+1)\mathbf{v}\pi}{16} \quad \text{(EQN 9)}$$

$\mathbf{S^{-1}}$ is a diagonal operator:

$$\mathbf{S^{-1}_{kl}} = \begin{cases} \mathbf{q}[\mathbf{k}] & \text{if } \mathbf{l} = \mathbf{k} \\ 0 & \text{otherwise} \end{cases} \quad \text{(EQN 10)}$$

and $\mathbf{Z^{-1}}$ is the inverse zig-zag operator:

$$\mathbf{Z^{-1}_{uvk}} = \begin{cases} 1 & \text{if zigzag[u,v]= k} \\ 0 & \text{otherwise} \end{cases} \quad \text{(EQN 11)}$$

J is a third rank tensor that maps a second rank tensor (an 8x8 pixel block) into a first rank tensor (a 64 element vector), as shown in figure 3. This mapping is expressed in the equation $\mathbf{F_l} = \sum_{\mathbf{i,\,i}} \mathbf{J_{ijl}} \mathbf{f_{ij}}$. Similarly, $\mathbf{J^{-1}}$ is a third rank tensor that reverse this mapping, using $\mathbf{f_{ij}} = \sum_{\mathbf{l}} \mathbf{J^{-1}_{ijl}} \mathbf{F_l}$.

The special structure of **Z** and **S** (and their inverses) allow use to compute **J** and $\mathbf{J^{-1}}$ efficiently. The C code in figure 4 shows a function, `InitOperators`, which computes **J** and $\mathbf{J^{-1}}$ and stores the result in two three dimensional lookup tables, one for each operator. The code uses three externally defined arrays, `zz`u, zzv, and `qt`, which encode the permutation specified by zig-zag ordering and the quantization tables, respectively.

3 Compressed Domain Processing

Having formulated images, processing, JPEG compression, and JPEG decompression as tensors, these steps are easily combined. Consider the process of processing a JPEG compressed grayscale image. With the processing specified by the tensor **T**, the following steps, illustrated in figure 5, are needed:

1. Decompress the input bitstream to form the *SC image*. The SC image is a two dimensional array of first rank tensors called *SC-vectors.* The SC-vectors are denoted $\mathbf{F_{xy}}$ or $\mathbf{H_{wz}}$, for the input or output image, respectively, and may be sparsely encoded. Each SC-vector corresponds to an 8x8 pixel block in the decompressed image. The subscripts specify the block offset.
2. Convert each SC-vector to an 8x8 block of pixels by applying $\mathbf{J}^{-1}$: $\mathbf{f_{xy}} = \mathbf{J^{-1}F_{xy}}$.
3. Compute each 8x8 pixel block in the output image using the block transforms $\mathbf{T}_{\text{wzxy}}$: $\mathbf{h_{xy}} = \sum_{\mathbf{xy}} \mathbf{T_{wzxy}} \mathbf{f_{xy}}$
4. Convert the output image to the SC representation using $\mathbf{H}_{\text{wz}} = \mathbf{Jh_{wz}}$.
5. Round, run-length, DPCM, and entropy encode $\mathbf{H}_{\text{wz}}$.

Steps 2, 3, and 4 can be combined:

$$\mathbf{H_{xy}} = \mathbf{J}\left(\sum_{\mathbf{xy}} \mathbf{T_{wzxy}}(\mathbf{J^{-1}F_{xy}})\right) = \sum_{\mathbf{xy}} (\mathbf{JT_{wzxy}J^{-1}})\mathbf{F_{xy}} \qquad \text{(EQN 12)}$$

The term in parentheses is the compressed domain equivalent of the block transform $\mathbf{T}_{\text{wzxy}}$:

$$\tau_{wzxy} = \mathbf{J}\mathbf{T}_{wzxy}\mathbf{J}^{-1} \qquad \text{(EQN 13)}$$

τ_{wzxy} is a block transform that computes an SC-vector in the output directly from the SC-vectors in the input. Using τ_{wzxy}, images can be processed in the compressed domain, as shown in figure 6. The steps are:

1. Decompress the input bitstream to form the SC image.
2. Compute each output SC-vector using the block transform τ_{wzxy}: $\mathbf{H}_{\mathbf{xy}} = \sum_{\mathbf{xy}} \tau_{\mathbf{wzxy}} \mathbf{F}_{\mathbf{xy}}$.
3. Round, run-length, DPCM, and entropy encode $\mathbf{H}_{wz}$.

Each SC-vector in the output ($\mathbf{H}_{wz}$) is computed by multiplying each SC-vector in the input ($\mathbf{F}_{xy}$) by it corresponding block transform (τ_{wzxy}) and accumulating. Since SC-vectors are first rank tensors (i.e., vectors, or one dimensional arrays), and the block transforms τ_{wzxy} are second rank tensors (i.e., matrices, or two dimensional arrays), the structure of the calculation is a sum of matrix/vector multiples.

Returning to the example of shrinking a 32x16 image (4x2 blocks) to a 16x8 image (2x1 blocks), to compute the left SC-vector, $\mathbf{H}_{00}$ (figure 7), the SC-vectors $\mathbf{F}_{00}$, $\mathbf{F}_{01}$, $\mathbf{F}_{02}$, $\mathbf{F}_{03}$, $\mathbf{F}_{10}$, $\mathbf{F}_{11}$, $\mathbf{F}_{12}$, and $\mathbf{F}_{13}$ are multiplied by the block transforms τ_{0000}, τ_{0001}, τ_{0002}, τ_{0003}, τ_{0010}, τ_{0011}, τ_{0012}, and τ_{0013}, respectively:

$$\mathbf{H}_{00} = \tau_{0000}*\mathbf{F}_{00} + \tau_{0001}*\mathbf{F}_{01} + \tau_{0002}*\mathbf{F}_{02} + \tau_{0003}*\mathbf{F}_{03} + \tau_{0010}*\mathbf{F}_{10} + \tau_{0011}*\mathbf{F}_{11} + \tau_{0012}*\mathbf{F}_{12} + \tau_{0013}*\mathbf{F}_{13}$$

Since, in shrink by 2, the right most four blocks in $\mathbf{F}$ do not affect $\mathbf{H}_{00}$, the block transforms τ_{0002}, τ_{0003}, τ_{0012}, and τ_{0013} are zero and can be ignored for efficiency. The other four block transforms, τ_{0000}, τ_{0001}, τ_{0010}, and τ_{0011}, are 64 by 64 matrices. To give you some intuition for what these matrices look like, the first sixteen rows and columns of τ_{0000} are shown in the top of figure 4. A scatter plot showing the positions of the non-zero elements of τ_{0000} is shown in the bottom of figure 4.

This example illustrates an important property of τ: most of the τ_{wzxy} are zero. This property allows the computation of output images to be performed efficiently. It holds for many compressed domain image operators, since in most image processing a pixel in the output image is a function of only a few pixels in the input.

Despite the spareness of τ, the compressed domain operation is still slow. To see why, recall that the output SC-vector is the sum of a sequence of matrix/SC-vector multiplies. Each SC-vector has 64 elements, and each of the multiplying matrices are 64x64, so 4K multiply/add operations are required for each matrix/SC-vector multiply. With several such terms, the operation count gets large. For example, shrink-by-2 requires four matrix/SC-vector multiplies per output block, so 16K multiply/add operations are required for each output block. Thus, an average of 16K/64 = 256 multiplies is required per pixel (since each SC-vector represents 64 pixels), which is more expensive than the spatial domain operation.

Since the SC-vectors are stored in a run length encoded format and typically sparse, sparse matrix techniques can be used to reduce the number of multiplies. But the computation is still too expensive to compute in real-time on general purpose workstations. The next section develops an approximation technique that reduces this cost to a few multiplies per pixel.

4 Condensation

This section describes a technique, called *condensation,* that approximates compressed domain operators so they can be efficiently computed. Condensation modifies the operator τ to produce a new operator τ' such that τ' is sparse and when τ' is used to compute an effect, the result will be nearly identical to that computed using τ. In other words, if $\mathbf{H}=\tau\mathbf{F}$ and $\mathbf{H'}=\tau'\mathbf{F}$, then $\mathbf{H} \approx \mathbf{H}'$.

Since τ' is sparse and the input vectors **F** are sparse, the resulting computation can be implemented efficiently. Two properties, one of τ and one of the input vectors, make condensation possible.

1. Most elements of τ have small absolute values [1]. For example, 90% of the elements in shrink-by-2 have an absolute value less than 0.05. You can see this by examining figure 4.
2. The input vectors **F** are sparse, and non-zero values are typically small integers. This property is expected, since the DCT concentrates the energy of the image into a few coefficients. Furthermore, JPEG quantizes high frequency components aggressively, leading to the small absolute values of these elements.

These two properties allow us to approximate τ as a sparse tensor as follows. An element in an output SC-vector is a linear combination of elements in a set of input SC-vectors. The elements themselves are small integers, and the

coefficients of this linear combination are stored in τ. Small elements of τ, called *insignificant* elements, will have little effect on the value of this sum, since they will be multiplied by small integers, and the result will be rounded off anyway in the next step (figure 6). In other words, why go to the trouble of computing the output to several decimal points if you are going to throw away the fraction anyway?

We can exploit this observation by setting insignificant elements in τ to zero. Doing so will reduce the number of operations required to compute **H**, but at the price of a small error in the output. Such errors are likely to be undetected because JPEG compression introduces the same type of loss. Since the majority of the elements of τ are insignificant (property 1), this optimization should save a large number of arithmetic operations. We call this process of setting elements of τ to zero *condensation*. In effect, condensation introduces a dead zone into the operator: elements in the operator below a threshold are set to zero. The question is: what elements of τ can we safely set to zero?

To answer this question, let us formulate the concept of condensation more precisely. Recall that the value of an output vector **H** is computed as a sum of matrix/SC-vector multiplies:

$$\mathbf{H_{wz}} = \sum_{\mathbf{xy}} \tau_{\mathbf{xywz}} \mathbf{F_{xy}} \qquad \text{(EQN 14)}$$

Let N be the number of matrix/SC-vector multiplies in this sum (e.g., N=4 for shrink-by-2). If we use the condensed operator τ' instead of τ, the error in $\mathbf{H_{wz}}$ is

$$\Delta\mathbf{H_{wz}} = \sum_{\mathbf{xy}} \tau_{\mathbf{xywz}} \mathbf{F_{xy}} - \sum_{\mathbf{xy}} \tau'_{\mathbf{xywz}} \mathbf{F_{xy}} = \sum_{\mathbf{xy}} \Delta\tau'_{\mathbf{xywz}} \mathbf{F_{xy}} \qquad \text{(EQN 15)}$$

where $\Delta\tau$ is the tensor composed of insignificant elements of τ. The worst case error occurs when all elements in **F** are at their maximum value. In *thresholding condensation*, the tensors τ are condensed by guaranteeing that no element in $\Delta\mathbf{H_{wz}}$ will never exceed a parameter *maxerr* in a worst-case scenario. If we let $\mathbf{max_k}$ denotes the worst-case (i.e., the largest) value of the *k*th element of the SC-vector, the heuristic to zero an element of τ is

$$\left|\tau_{\mathbf{wzxykl}}\right| < \frac{\mathbf{maxerr}}{64 \times \mathbf{N} \times \mathbf{max_k}} \qquad \text{(EQN 16)}$$

Max$_k$ can be chosen statistically using data gathered from a large set of images [1]. These ideas lead directly to the following algorithm

Algorithm (Thresholding Condensation)

```
1.  max: array [0..63] of integer;
2.  N := number of transform tensors
3.      for each tensor τ
4.      for k := 0 to 63 do begin
5.          threshold := maxErr/(64*N*max[k]);
6.          for l := 0 to 63 do
7.              if (τ[l][k] <= threshold) then
8.                  τ[l][k] = 0.0;
9.      end
```

In this code, the array τ is a block transform tensor and `max` stores the largest expected value of an AC component of any SC-vector (see appendix A). In lines 7-8, any insignificant element, as specified by equation 16, is set to zero.

The threshold is set so that the error in the output is bounded by *maxerr*. Unfortunately, when large values of *maxerr* are used, the block transform matrices τ_{wzxy} cannot be condensed independently. To see why, suppose you had an operator with *N=3* (i.e., the output SC-vector is a linear combination of three input SC-vectors), and the value of the first AC component of the output SC-vector is given by

$$H_1 = 2A_0 - B_0 - C_0$$

where A_0, B_0, and C_0 are the DC components of the three input SC-vectors. If $A_0 = B_0 = C_0$, the terms cancel and H_1 is zero. Now, suppose condensation uses a threshold such that the two terms with B_0 and C_0 drop out. Then the new value for H_1 is $H_1 = 2A_0$. Since A_0 is the DC component, A_0 can be large, and in such a case the output component H_1 will be large and result in highly visible artifacts in the output. Figure 9 shows a solid gray image filtered with a gaussian blur filter where each compressed domain tensor in the filter was condensed independently

using thresholding condensation. The pattern of dots are artifacts caused by setting AC components of the output SC-vector, such as H_1, to relatively large values (the output should be uniform gray).

This problem can be solved by introducing the concept of *tensor bias*. The tensor bias of τ is defined as

$$\mathbf{b_{wz}(k, l)} = \sum_{xy} \tau_{wzxykl} \qquad \text{(EQN 17)}$$

Intuitively, tensor bias is a measure of how much the cross-block terms in equation 14 tend to cancel each other out when the tensor is used to compute the output block at *w, z*. In the example above, $b_{wz}(0,1)$ is 2 - 1 - 1 = 0 before condensation, and $b_{wz}(0,1)$ is 2 after condensation. The change in tensor bias means that terms that cancelled each other out before condensation do not do so afterwards, resulting in artifacts such as those in figure 9.

We can remove these artifacts by adding the *constant bias* constraint to condensation: the tensor bias after condensation should be the same as before. To implement the constant bias constraint, we calculate and store the bias of the tensor before applying a condensation algorithm, condense the tensor, and then adjust the remaining non-zero elements to restore the bias to its previous value. More precisely, we adjust the elements in the tensors by distributing the change in bias δb equally among the non-zero elements remaining in the tensor after condensation. If no elements remain, a randomly selected tensor absorbs the change in bias. δb is given by

$$\delta \mathbf{b_{wzkl}} = \mathbf{b_{wz}(k, l) - b'_{wz}(k, l)} \qquad \text{(EQN 18)}$$

5 Implementation and Experimental Results

This section describes a set of experiments we performed to evaluate the effectiveness of compressed domain processing using condensation. The experiments characterize both the performance of the technique and the quality of the images computed using condensed operators. We first describe the implementation and then report the performance results.

Our implementation is divided into two phases. In phase one, the compressed domain tensor τ is computed, condensed, and stored in a file. In the second phase, τ is read from a file, the JPEG stream is entropy decoded to recover the SC-image, τ is applied to the SC-vectors to compute the output SC-image, and the result is encoded as a JPEG bitstream. Phase one is executed off-line, whereas phase two operates in real-time. Since we are not

concerned with the speed of off-line processing, our implementation is optimized for phase two. In practice, phase one takes a few seconds on a typical workstation.

To make phase two efficient, we must develop a data structure for efficiently calculating the output SC-image from the input SC-image. This calculation can be written:

1. **for all** w, z in the output image
2. zero the output SC-vector $\mathbf{H_{wz}}$
3. **for all** x, y in the input image such that τ_{wzxy} is not all zero
4. Compute $\mathbf{F}_{xy} * \tau_{wzxy}$ and add the result to $\mathbf{H_{wz}}$

An efficient data structure will exploit the sparseness of the operators and the SC-vectors, and the redundancy in the block transforms. We used the data structures are diagrammed in figure 10. The SC-vectors of $\mathbf{F}_{xy}$ are stored in the *SparseVector* data type. Each SparseVector has a field indicating the size of the array and an array of (*index, value*) pairs, which indicate the position and value of the SC-vector's non-zero elements. Each matrix τ_{wzxy} is stored in a *SparseMatrix* data structure, consisting of an array of 64 pointers to SparseVectors indexed by *k*. The SparseVectors in a SparseMatrix correspond to the columns of a block transform τ_{kl}. In this usage, *index* specifies the row index *l* and *value* contains τ_{kl}. In our implementation, each unique SparseMatrix is stored in a table called the SparseMatrixTable, and offsets in this table are used to reference a particular SparseMatrix. The set of block transforms needed to compute an output SC-vector is stored in a linked list of *SparseMatrixRef*s. A SparseMatrixRef is a tuple (*x, y, num*) where *x* and *y* indicate the block coordinates of the source SparseVector $\mathbf{F}_{xy}$ used in line 4, and *num* indicates the offset of the matrix τ_{wzxy} in the SparseMatrixTable. The entire compressed domain tensor is stored in the *TransformTable*, a two dimensional array of such lists.

1. **for all** w, z in the output image

```
2.      zero the output SC-vector H;
3.      SparseMatrixRef = TransformTable[w,z];
4.      while (SparseMatrixRef != NULL)
5.          SparseMatrix = SparseMatrixTable[SparseMatrixRef->n]
6.          ApplyBlockTransform (F[SparseMatrixRef->x, SparseMatrixRef->y], SparseMatrix, H);
7.          SparseMatrixRef = SparseMatrixRef->next;
```

ApplyBlockTransform multiplies an SC-vector (its first parameter) by a sparsely encoded block transform (its second parameter) and accumulates the result in its third parameter.

The advantage of this representation is that it exploits the sparseness of τ, it enables the inner loop of the compressed domain processing algorithm to be efficiently implemented, and it allows matrices to be shared. This last property allows operators with repeated tensors, such as convolutions, to be stored efficiently.

Experimental Results

Having sketched the data structures used in the implementation, we now answer the question "how well does it work?" This question gives rise to two questions: 1) how does the *maxerr* parameter of thresholding condensation affect the quality of the output image and 2) how fast can an operation be computed on a current generation workstation? We will answer these questions in turn.

The *maxerr* parameter in thresholding condensation affects the time needed to compute the result and the quality of the result. Experiments showed *maxerr* to be a non-intuitive measure. For example, with *maxerr = 2000*, reasonable quality images were produced. This is because thresholding condensation used worse case values for the input blocks, which rarely occur. Further investigation showed the average number of multiplies needed to calculate an output vector, which is a function of *maxerr*, provided a more meaningful measure of condensation than *maxerr*.

We evaluated the distortion introduced by thresholding condensation for two operations: the *blur* operation, which convolves the image with a 7x7 Gaussian filter, and the *shrink-by-2* operation, which shrinks an image by a factor of 2 along each dimension. In the experiment, 12 condensed operators, corresponding to 12 different values of

maxerr, were created for each test operator using thresholding condensation. We applied the resulting operators to 98 randomly selected grayscale images and measured two values: the average number of multiplies required to calculate an output vector and the signal-to-noise ratio (SNR) of the resulting images. SNR is defined as

$$\mathbf{SNR} = 10log\left[\frac{\mathbf{rms}(\mathbf{O})}{\mathbf{rms}(\mathbf{C}-\mathbf{O})}\right]$$

where *rms*(•) is the root mean squared over the image, *C* is the image calculated using the condensed tensor, and *O* is the image using the original (uncondensed) tensor.

Figure 11 show the effect of condensation on shrink-by-2. The left image was computed with the uncondensed operator (1100 mults/vector), and the right image was computed with the condensed operator at *SNR=33* (about 330 mults/vector). Figure 12 shows the effect of condensation on blur. The top image was computed with the uncondensed operator (about 5200 multiplies for each output vector), and the bottom image was computed with the condensed operator at *SNR=30* (about 90 multiplies for each output vector). These figures are intended to give the reader an intuitive feeling for the artifacts introduced by condensation and the relationship between SNR and image quality. Subjective evaluation by the authors indicate that at an SNR of about 30, the output quality is quite good, and at an SNR above 35, the output image is essentially identical to the image computed using the uncondensed operator. At SNR values less than 30, the quality of the image degrades rapidly. Figure 13 shows a graph of the mean SNR for blur and shrink-by-2 as a function of the number of multiplies.

Table 1 compares the performance of the image space method with our implementation using shrink-by-2 and blur at various levels of condensation. The experiments were performed on the same test suite of 98 images used for our earlier experiments. The tests used a prototype implementation on a DEC 3000/400 workstation with 64 MBytes of memory. The table shows that, with reasonable quality output, shrink-by-2 can be applied to a 640x480 image at about 5 frames per second (fps) on our test workstation, and blur can be applied to the same image at about 3 fps. Shrink-by-2 is faster that blur because the image produced by shrink-by-2 is smaller than the image produced by blur, which speeds image encoding. The results scale approximately linearly with image area, so shrink-by-2 and blur can be applied to 320x240 images at about 20 fps and 12 fps, respectively

Tables 2 and 3 show the results of profiling our implementation using the two test operators. The table divides the total execution time into four phases: Huffman Decoding, Huffman Encoding, Operator Application, and Overhead. Huffman Decoding is the time spent reading and decoding the JPEG file into SC-vectors. Huffman Encoding is the time spent encoding the output, including quantization, run length coding, DPCM, and bitstream generation. Operator Application is the time spent computing the product of the condensed operator and input SC-vectors, and Overhead is the time spent in control flow. In the blur transformation, less than half the time is spent in application of the condensed operator. For shrink-by-2, only about a quarter of the time is spent in operator application. The rest of the time is spent in overhead and in entropy coding operations. These results indicate that limited performance gains are possible by further condensation.

6 Applications

Using the techniques developed in this paper, many important image and video processing problems can be computed in the compressed domain:

1. *Geometric operations*. Image warping that use such operations such as translation, rotation, scaling, shearing, and other affine transformations can be expressed using tensors[3]. For example, in scaling a 640 by 480 image to 320 by 240, each pixel in the output image is the average (i.e., a linear combination) of the corresponding pixels in the input image.
2. *Finite impulse response (FIR) filters* used in signal processing are can be expressed using tensors. Such operations, which include smoothing, embossing effects, edge detection, and image enhancement, can be conveniently represented using convolution [4]. The convolution function specifies the linear combination of pixels in the input image that are used to calculate a pixel in the output image.
3. *De-interlacing*. In this operation, two images, called the odd and even fields, are combined to form a single frame. The fields contain sample data from every other line in the video source. A frame is formed by interleaving lines from two fields. To express this operation as a tensor, the two fields are first scaled by a factor of two in the vertical dimension, with the missing lines set to 0 (black). The two resulting frames can then be added together pixel-wise to create the deinterlaced frame.
4. *Sampling conversion*. Video is often represented as a luminance and two chrominance channels. The lumi-

nance channel is a gray scale image, whereas the chrominance channels contain the extra information necessary to produce a color image. To reduce storage and bandwidth costs, chrominance channels are often sampled at a different resolution than the luminance channel. For example, MPEG [5] uses 4:2:0 sampling, where the luminance image is stored at 352x240 resolution but the chrominance images are stored at 176x120 resolution. Other standards use different sampling, such as 4:2:2. To transform 4:2:2 video to 4:2:0 video, the chrominance images must be down-sampled, a process analogous to image scaling.

5. *Morphing* is a striking video effect where objects in two video sequences are deformed and the images are cross-dissolved to create the illusion that one object is transforming into another [6]. The effect is achieved by applying affine transformations to sections of each image pair, which can be expressed as a tensor. The pixels in the resulting images are then multiplied by a scalar constant and added together.
6. *Image composition*. In video composition, multiple video inputs are combined to form a single video output (e.g., chroma key). Such a compositions can be expressed using a combination of image translation and scaling on the input images, pixel-wise multiplication with a mask image, and pixel-wise additions [7, 8]. This type of video mixing has been proposed as the basis for next generation video conferencing systems that create a virtual conference table, where camera inputs from other conference participants are mixed to form a composite signal that displays the other attendees seated around a table [9].

In practice, the memory needed to store the compressed domain tensor τ is too large to be practical unless the spatial domain operation exhibits enough symmetry that the block transforms can be shared. Of the examples above, only those that use image warping, such as morphing and general affine geometric transforms, are impractical for this reason.

7 Related Work

Image and video data processing in the spatial domain is a well studied field. The work in this field can be divided into hardware designs for video processing [10, 11, 12], applications that perform video processing off-line [13, 14, 15], and software techniques and algorithms for image processing [3, 4, 7, 16]. Work on processing video data in the compressed domain has seen less study. Chitprasert and Rao developed a restricted form of the convolution theorem for the DCT similar to the DFT convolution theorem [17]. Chang and Messerschmitt have developed a

technique for compositing motion compensated video in the compressed domain [8, 18]. Their work can be viewed as a special case of a translational operator coupled with factoring for improved efficiency. More recently, Natarajan and Bhaskaran [19] have found an efficient method for the special case of the shrink-by-2 operation in the compressed domain operator. Their method approximates the elements in the compressed domain operator using powers of two, allowing the result to be computed using only shifts and adds. Shen and Sethi have similarly examined inner block transforms, operations whose range is confined to a single 8 by 8 block [20]. Arman has developed a technique to detect scene changes in motion-JPEG compressed video data in the DCT domain [21]. Seales has examined the problem of object recognition in the compressed domain [22]. Broadhead and Owen have extended these techniques MPEG compressed audio data [23].

Many extensions are possible to the work presented in this paper. Condensation algorithms can be developed that improve the overall image quality using better metrics for finding insignificant elements than thresholding condensation. For example, the first author's dissertation [1] explored an algorithm that bounds the average case error rather than the worst case error. Unfortunately, the results were no better than those obtained using thresholding condensation.

The technique of expressing compression and transformation as linear operators, composing the operators, and condensing them to produce good approximations of the operator can be applied to a wide variety of other transform-based coding strategies (e.g., wavelet encoding). An interesting research question is how the use of other transforms will affect the trade-off of output quality and computation time. Careful study in this area might lead to good transcoding techniques that efficiently convert between different compressed representations.

Finally, rather than asking "how fast can I process data in this compressed format?" perhaps a better question is "can a compression format be developed that allows fast processing?" If a compression format can be developed that allows rapid processing and transcoding to popular compression standards such as motion-JPEG or MPEG, it would make an excellent format for secondary storage on video servers with heterogenous clients. Video servers could then store a single format and convert it to the appropriate client format in real time.

References

1. B. C. Smith, *Implementation Techniques for Continuous Media Systems and Applications*, Ph. D. Dissertation, University of California, Berkeley, CA, September 1994.

2. B. C. Smith, L. A. Rowe, *Algorithms for Manipulating Compressed Images*, IEEE Computer Graphics and Applications, September 1993, Volume 13, Number 5, pp 34-42.

3. J. D. Foley, et. al., *Computer Graphics: Principles and Practice*, second edition, Reading, Mass. Addison-Wesley, 1990.

4. A.K.Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall, Inc. Edglewood Cliffs, New Jersey, 1989

5. D. Le Gall, *MPEG:A Video Compression Standard for Multimedia Applications*, Communications of the ACM, April 1991, Volume 34, Number 4, pp 46-58

6. T. Bier, S. Neely, Feature-Based Image Metamorphosis, Computer Graphics, July 1992, Volume 26, Number 2, pp 35-42

7. T. Porter, T. Duff, *Compositing Digital Images*, Computer Graphics, July 1984, Volume 18, Number 3, pp 253-259.

8. S. F. Chang, W. L. Chen, D.G. Messerschmitt, *Video Compositing in the DCT Domain*, IEEE Workshop on Visual Signal Processing and Communications, Raleigh, NC, September 1992, pp. 138-143

9. D. Boyer, M. Lukacs, *The Personal Presence System - A Wide Area Network Resource for the Real Time COmposition of Multipoint Multimedia Communications*, Proceedings of the Second ACM International Conference on Multimedia, October 1994, San Francisco, Calif.

10. K. Chen, C. Svensson, *A 512-Processor Array Chip for Video/image Processing*, Pixels to Features II. Parallelism in Image Processing. Proceedings of a Workshop, Bonas, France, August 1990

11. L. G. Chen, et. al., *A Real-time Video Signal Processing Chip*, IEEE Transactions on Consumer Electronics, May 1993, Volume 39, Number 2:82-92.

12. H. Yamauchi, et. al. *A Highly Parallel Single Chip DSP Architecture for Video Signal Processing*, 1991 International Conference on Acoustics, Speech and Signal Processing New York, NY, Volume 2, pp 197-200

13. J.W. Klingler, L.T. Andrews, C. Vaughan, *Fusion of Digital Image Processing and Video on the Desktop*, Digest of Papers. COMPCON 92. Thirty-Seventh IEEE Computer Society International Conference, San Francisco, CA, February 1992

14. E. Holsinger, *Avid Media Suite Pro 2.0*, MacWEEK Volume 8, Number 4, Jan 24, 1994, pp 40.

15. J. Wolfskill, *Video Editor Makes PC Premiere*, Windows Magazine, February 1994, Volume 5, Number 2, pp 116

16. B. K. P. Horn, *Robot Vision*, MIT Press, Cambridge, Mass, 1986

17. B. Chitprasert, K. R. Rao, *Discrete Cosine Transform Filtering*, Signal Processing, Volume 19, Number 3, pp. 233-245, March 1990

18. S. F. Chang, D. G. Messerschmitt, *A New Approach to Decoding and Compositing Motion Compensated DCT-Based Images*, IEEE Intern. Conf. on Accoustics, Speech, and Signal Processing, Minneapolis, Minnesota, pp. V421-V424, April 1993.

19. B. Natarajan, V. Bhaskaran, *A Fast Approximate Algorithm for Scaling Down Digital Images in the DCT Domain*, unpublished manuscript, Hewlett-Packard Laboratories, Palo Alto, CA.

20. B. Shen, I. Sethi, *Inner-block operations on compressed images*, to appear in Proceedings of the Third ACM International Conference on Multimedia, November 1995, San Francisco, Calif.

21. F. Arman, A. Hsu, M. Y. Chiu, *Image Processing on Compressed Data for Large Video Databases*, Proceedings of the First ACM International Conference on Multimedia, August 1993, Anaheim, Calif.

22. B. Seales, *Vision and Multimedia: Object Recognition in the Compressed Domain*, unpublished manuscript, available from ftp://sarod.dcs.uky.edu/pub/papers/seales/compressed.nofig.ps.gz.

23. M. A. Broadhead, C. B. Owen, *Direct Manipulation of MPEG Compressed Digital Audio*, to appear in Proceedings of the Third ACM International Conference on Multimedia, November 1995, San Francisco, Calif.

24. FTP archive at URL ftp://ftp.funet.fi/pub/amiga/graphics/pics

25. W. B Pennebaker, J. L. Mitchell, *JPEG still image data compression standard,* Van Nostrand Reinhold, New York, 1992.

Appendix A

This appendix lists the experimentally determined values for `maxK` used in thresholding condensation in section 4. The data was gathered by examining 622 images from 14 categories stored on an FTP archive [24], compressed using the default quantization tables presented in Annex K of the JPEG standard [25]. The value `maxK[i]` indicates the largest value of the SC-vector element `F[i]` seen in all 622 images. Thus, maxK[i] represents a practical upper limit of the absolute value of `F[i]`.

```
int max[64] = {
255, 128, 128, 128, 128, 128, 128, 112,
119, 128,  74,  70,  82,  88,  77,  40,
 58,  51,  50,  47,  52,  28,  28,  36,
 37,  36,  54,  34,  28,  28,  25,  24,
 25,  18,  20,  21,   9,  12,  16,  14,
 19,  22,  21,  19,  12,  11,  13,  11,
 10,   9,   7,  11,  11,  13,  12,  10,
  7,   6,  10,   6,   9,   9,   7,   9};
```

Operator	Test Conditions	Time (seconds)	Speedup
Blur	SNR = 25	0.290	43.4
Blur	SNR = 30	0.331	38.0
Blur	Not Condensed	4.45	2.83
Blur	Image Space	12.6	1
Shrink-by-2	SNR = 25	5.36	0.141
Shrink-by-2	SNR = 30	3.74	0.202
Shrink-by-2	Not Condensed	2.30	0.328
Shrink-by-2	Image Space	1	0.755

Table 1: Speed of the blur and shrink-by-2 operation

Test Conditions	Huffman Decoding	Huffman Encoding	Operator Application	Overhead
SNR = 25	16%	21%	39%	24%
SNR = 30	13%	19%	45%	23%
Not Condensed	1%	2%	95%	2%

Table 2: Breakdown of time in computing the blur operation

Test Conditions	Huffman Decoding	Huffman Encoding	Operator Application	Overhead
SNR = 25	47%	20%	23%	10%
SNR = 30	42%	18%	31%	9%
Not Condensed	15%	7%	75%	3%

Table 3: Breakdown of time in computing the shrink-by-2 operation

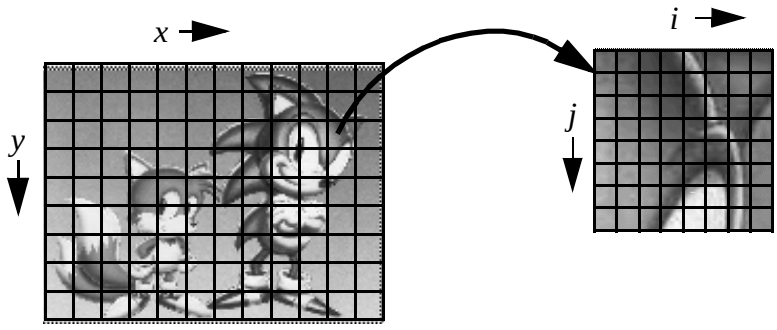

Figure 1: Block-oriented Pixel Addressing

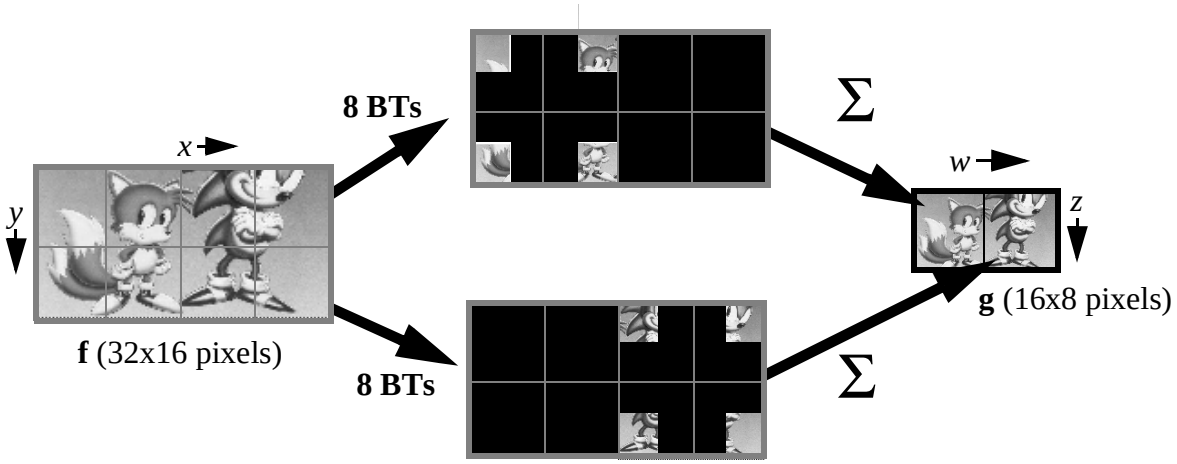

Figure 2: Shrink-by-2 example

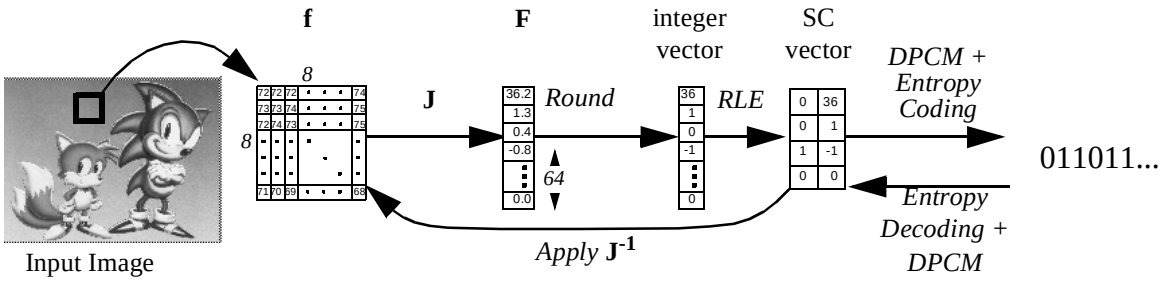

Figure 3: The JPEG compression/decompression process

```
#define PI    3.14159265358979323846
#define SQRT2 1.4142135623730950488
#define A(u)  ((u)? 0.5 : 0.5/SQRT2)
static double J[8][8][64], Jinv[64][8][8], C[8][8];

InitOperators () {
  int i, j, k, u, v;
  double tmp;

  for (i=0; i<8; i++) {
     for (u=0; u<8; u++) {
        C[i][u] = A(u)*cos((2*i+1)*u*PI/16.0);
     }
  }

  for (k=0; k<64; k++) {
     u = zzu[k];
     v = zzv[k];
     for (i=0; i<8; i++) {
        for (j=0; j<8; j++) {
           tmp = C[i][u]*C[j][v];
           J[i][j][k] = tmp/quantTable[k];
           Jinv[k][i][j] = tmp*quantTable[k];
        }
     }
  }
}
```

Figure 4: C code for computing `J` and its inverse, `Jhat`

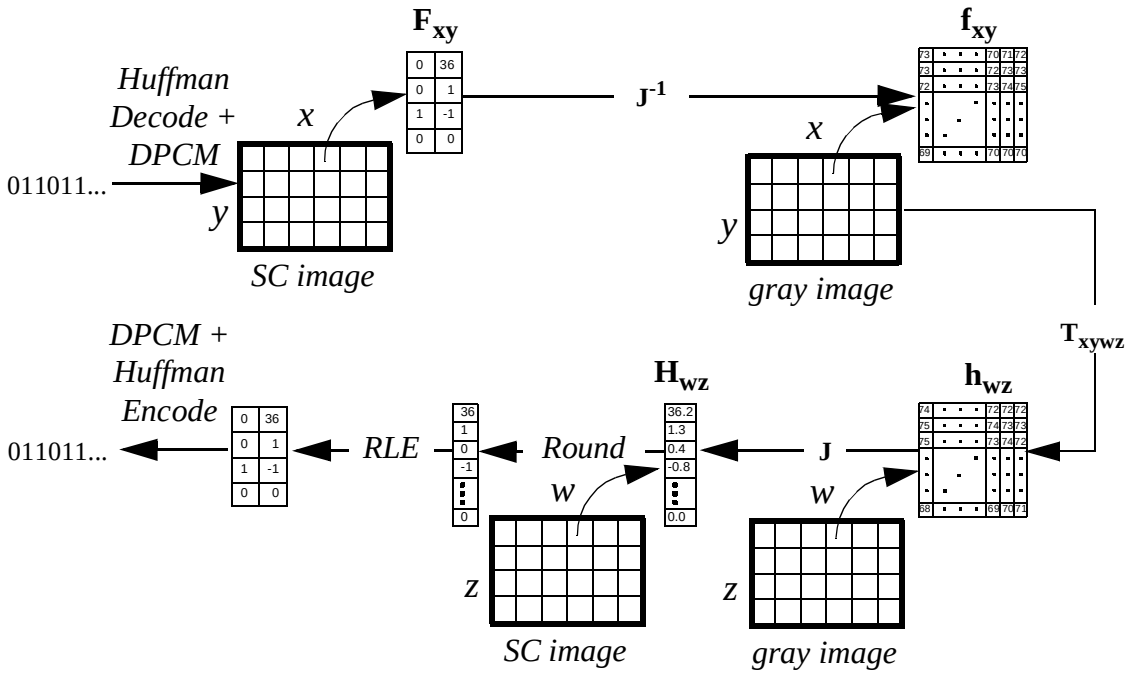

Figure 5: Graphical depiction of Spatial Domain Processing

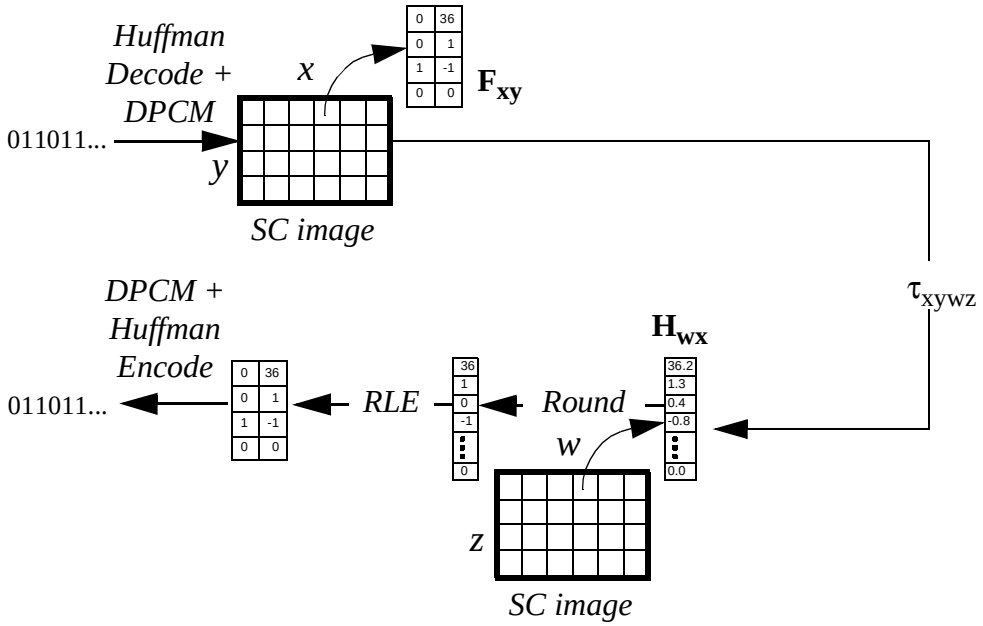

Figure 6: Graphical depiction of Compressed Domain Processing

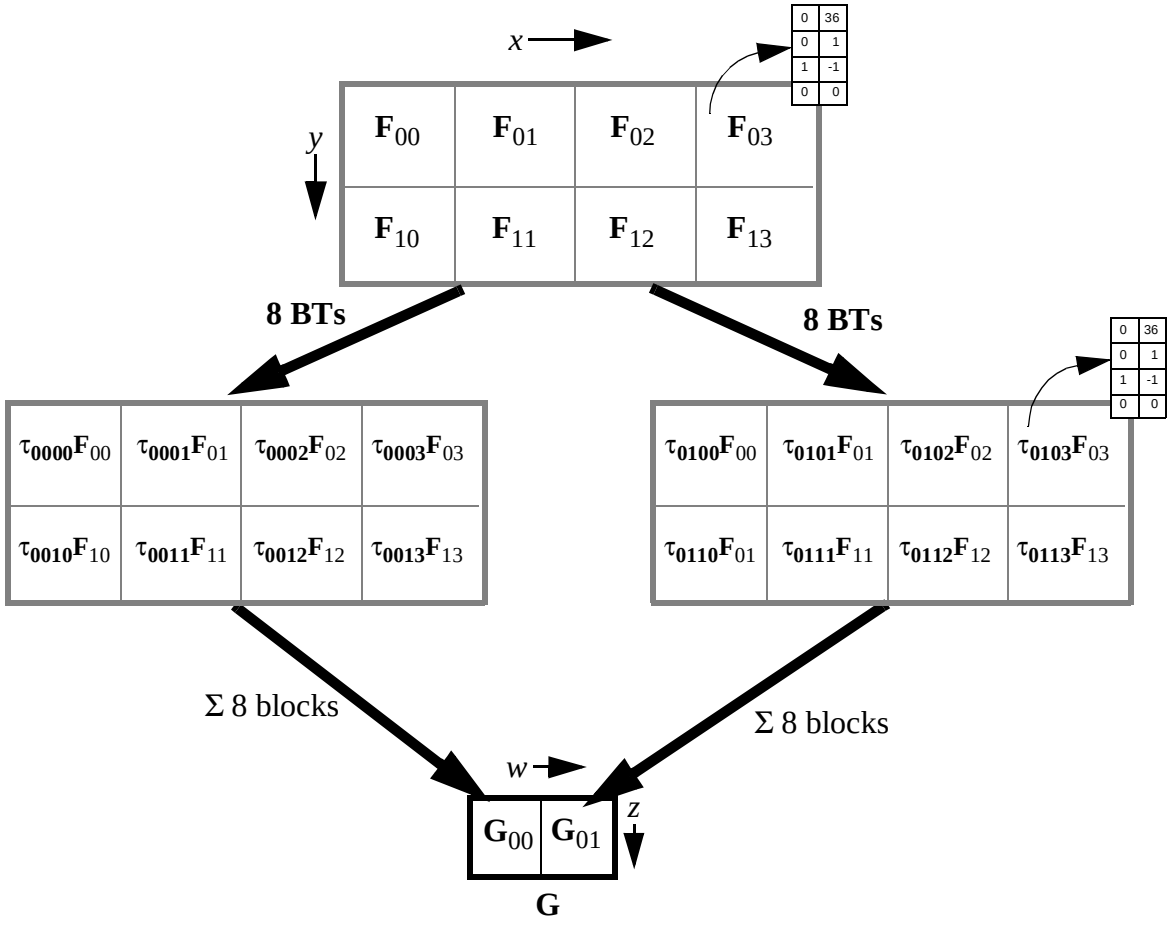

Figure 7: Shrink-by-2 filtering in the compressed domain

l \ *k*																
	0.250	0.330	0.302	0.000	0.274	0.000	-0.080	0.000	0.000	-0.091	0.000	-0.068	0.000	-0.061	0.000	0.021
	0.000	0.102	0.000	0.000	0.085	0.270	0.133	0.175	0.000	0.000	0.000	-0.021	0.000	0.102	0.000	-0.024
	0.000	0.000	0.102	0.210	0.092	0.000	0.000	0.000	0.205	0.166	0.000	0.124	0.000	-0.020	0.000	0.000
	0.000	0.000	-0.020	0.000	-0.018	0.000	0.000	0.000	0.000	0.119	0.180	0.088	0.000	0.004	0.000	0.000
	0.000	0.000	0.000	0.000	0.042	0.000	0.000	0.086	0.092	0.000	0.000	0.056	0.180	0.050	0.000	0.000
	0.000	-0.016	0.000	0.000	-0.013	0.000	0.074	0.000	0.000	0.000	0.000	0.003	0.000	0.057	0.096	0.044
	0.000	0.007	0.000	0.000	0.006	0.000	-0.020	0.000	0.000	0.000	0.000	-0.001	0.000	-0.015	0.000	0.041
	0.000	0.000	0.000	0.000	-0.008	0.000	0.000	0.000	-0.018	0.000	0.000	-0.011	0.000	0.036	0.000	0.000
	0.000	0.000	0.000	0.000	-0.008	0.000	0.000	-0.016	0.000	0.000	0.000	0.037	0.000	-0.009	0.000	0.000
	0.000	0.000	0.006	0.000	0.005	0.000	0.000	0.000	0.000	-0.020	0.000	-0.015	0.000	-0.001	0.000	0.000
	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	0.000	0.000	0.000	0.000	0.003	0.000	0.000	0.006	0.000	0.000	0.000	-0.008	0.000	0.003	0.000	0.000
	0.000	0.000	0.000	0.000	0.002	0.000	0.000	0.000	0.000	0.000	0.000	-0.008	0.000	-0.007	0.000	0.000
	0.000	0.000	0.000	0.000	0.003	0.000	0.000	0.000	0.007	0.000	0.000	0.004	0.000	-0.008	0.000	0.000
	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	0.000	-0.012	0.000	0.000	-0.010	0.000	0.034	0.000	0.000	0.000	0.000	0.002	0.000	0.026	0.000	-0.068

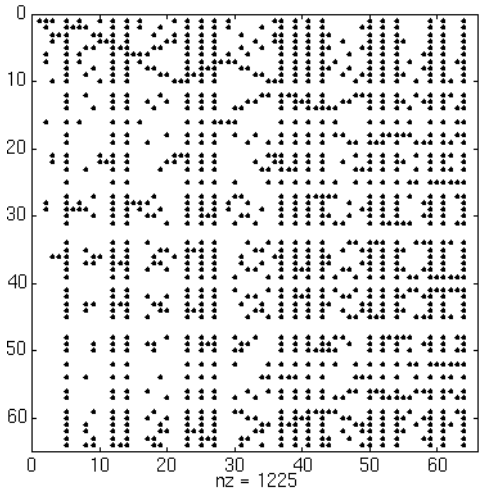

Figure 8: First 16 rows and columns of τ_{0000kl} for shrink-by-2 operation, and structure of its non-zero elements

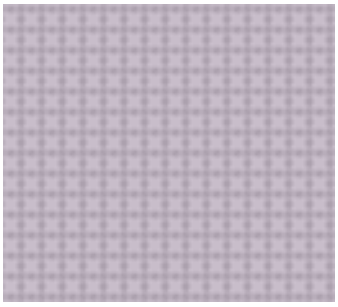

Figure 9: Gray image filtered with blur without constant bias

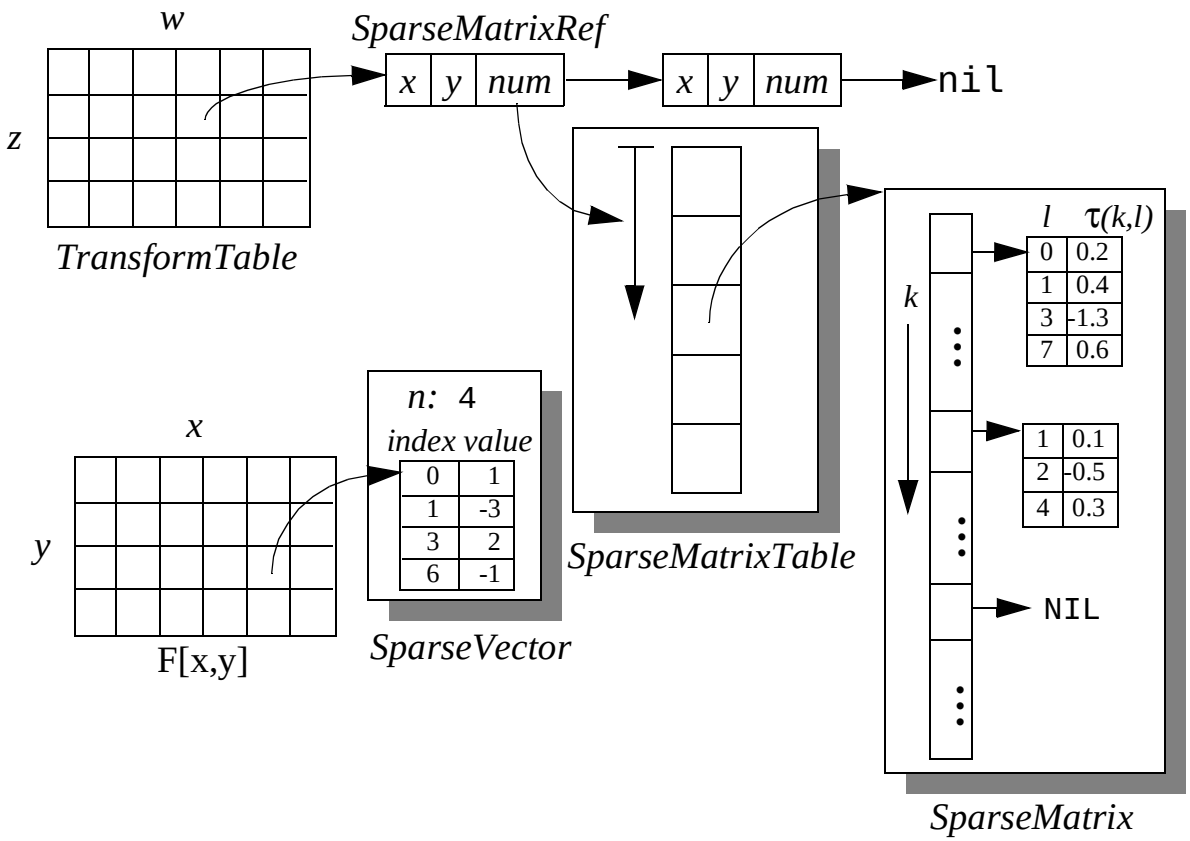

Figure 10: Data structures used in the implementation

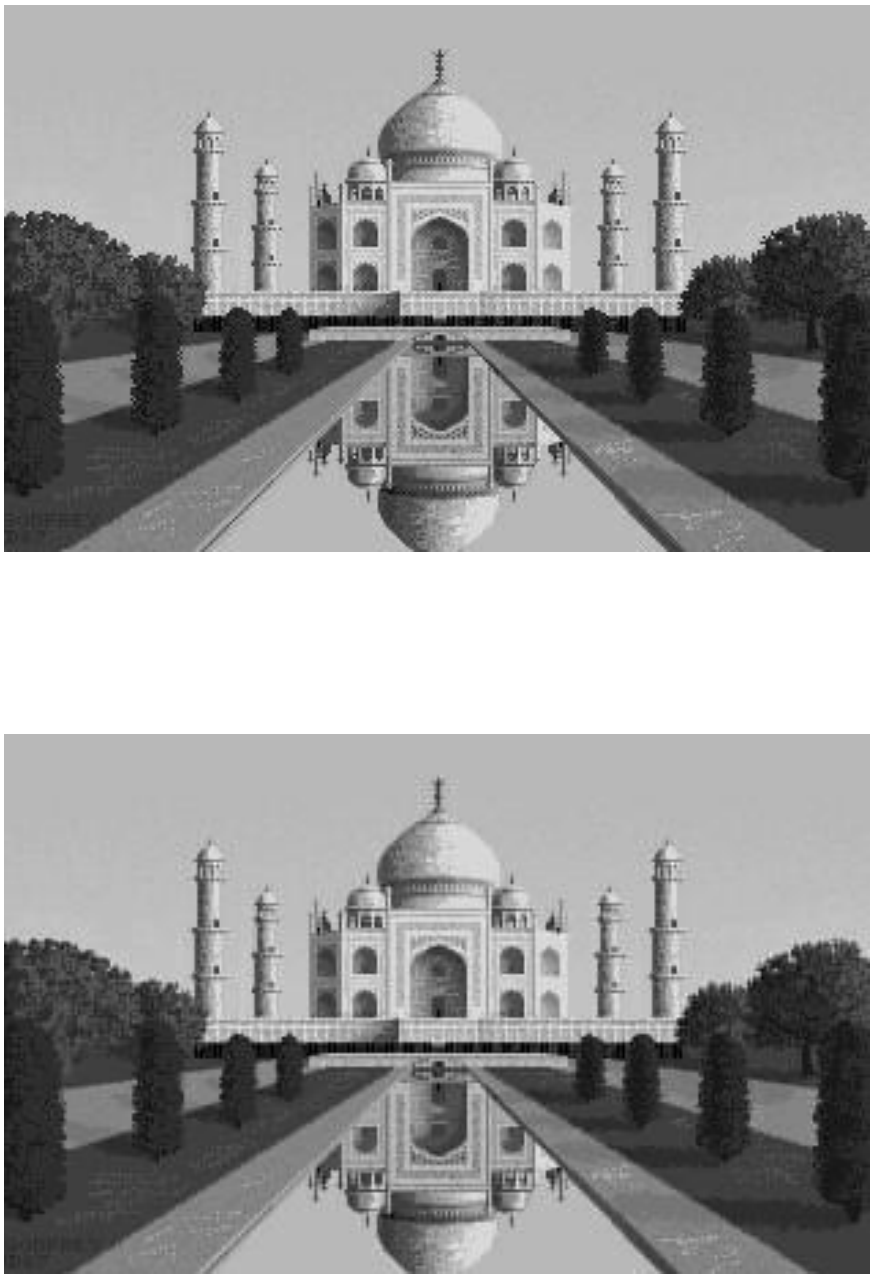

Figure 11: Effect of condensation for shrink-by-2. Top: uncondensed (1100 mults/vector). Bottom: condensed at SNR=30 (330 mults/vector)

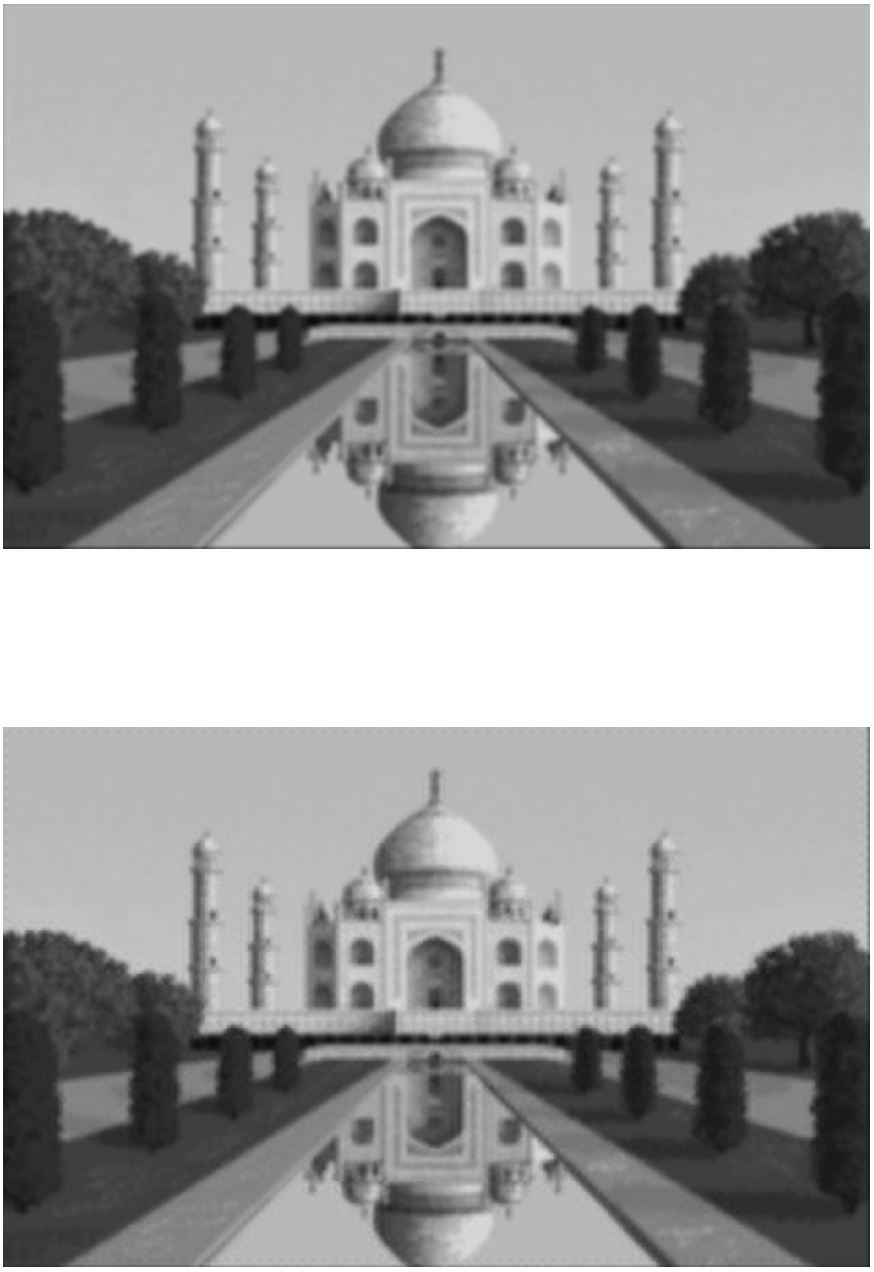

Figure 12: Effect of condensation for blur. Top: uncondensed (5270 mults/vector). Bottom: condensed at SNR=33 (90 mults/vector)

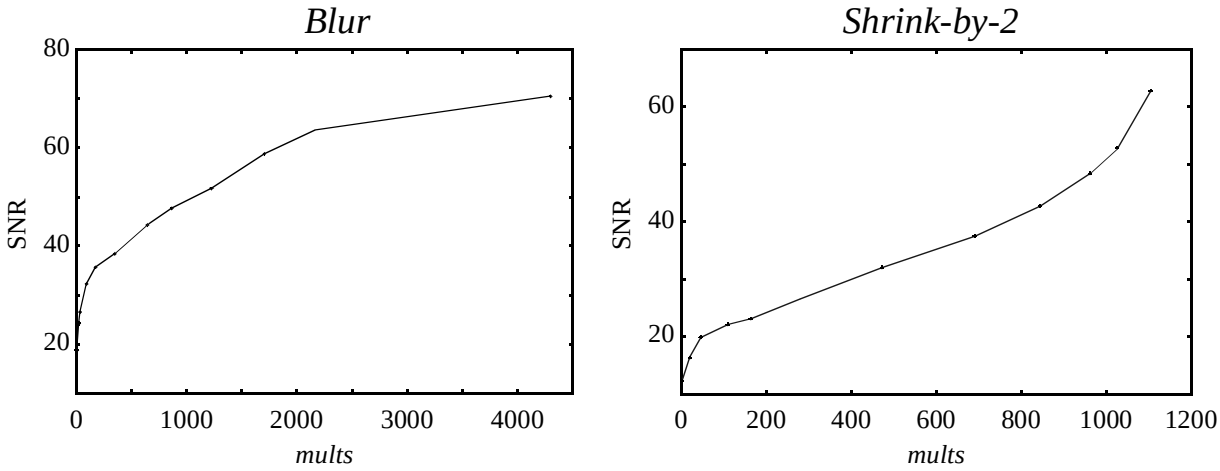

Figure 13: SNR of blur (left) and shrink-by-2 (right) vs. number of multiplies