

- Commun. COM-32, December, 1984, pp 1316-1322.
- [10] MPEG Video CD Editorial Committee, *MPEG Video Committee Draft*, December 18, 1990.
  - [11] Cosmos Nicolaou, *An Architecture for Real-Time Multimedia Communication Systems*, IEEE Journal on Selected Areas in Communications, Volume 8, No 3, pp. 391-400, April 1990
  - [12] Thomas D. C. Little, Arif Ghafoor, *Synchronization and Storage Models for Multimedia Objects*, IEEE Journal on Selected Areas in Communications, Volume 8, No 3, pp. 413-427, April 1990
  - [13] Wendy E. Mackay, Glorianna Davenport, *Virtual Video Editing in Interactive Multimedia Applications*, Communications of the ACM, Volume 32, No 7, pp 802-810, July 1989
  - [14] David Ripley, *DVI -- Digital Multimedia Technology*, Communications of the ACM, Volume 32, No 7, pp 811-822, July 1989
  - [15] Shiro Sakata, *Development and Evaluation of an In-House Multimedia Desktop Conference System*, IEEE Journal on Selected Areas in Communications, Volume 8, No 3, pp. 413-427, April 1990
  - [16] Robert J. Sciabassi, Robert Leichner, et. al., *The Multi-Media Medical Monitoring, Diagnosis, and Consultation Project*, 24th Hawaii International Conference on System Sciences, Koloa, Hawaii, January 8-11, 1991.
  - [17] B. Chitprasert, K. R. Rao, *Discrete Cosine Transform Filtering*, Signal Processing, Volume 19, No 3, pp. 233-245, March 1990
  - [18] Jae S. Lim, *Two-dimensional signal and image processing*, Prentice Hall, Englewood Cliffs, N.J., 1990

this computation, which is what we found. The situation is analogous to memory systems with a cache where cache misses are much slower than cache hits, and the hit rate is 50% to 75%: most of the time in the system is spent resolving cache misses.

Therefore, the problem of efficiently computing nonlinear operations is still open.

## Global Image Operations and Extensions To Other Compression Standards

Another area for future research lies in extending the class of operations that can be performed on compressed images. For example, scaling, rotating, translating, shearing, and perspective transformations would all be useful operations to perform on compressed images. Although the block oriented nature of transform based coding presents difficulties, we have found the basis of a general solution which we believe captures most of the nuances of this problem. Our results will be reported in a later publication.

Finally, there is the problem of extending these methods to the motion video compressing standards such as the CCITT H.261 standard (often called  $p \times 64$ ), and the proposal of the ISO Motion Picture Experts Group (MPEG). Since these standards use motion compensation, the extension is not as straightforward as could be hoped.

## References

- [1] T. Porter and T. Duff, *Compositing Digital Images*, SIGGRAPH '84 Proceedings, Volume 18, pp 253-259, July 1984
- [2] Andy Hopper, *Pandora - an experimental system for multimedia applications*, Olivetti Research Laboratory, Trumpington St., Cambridge, UK, CB21QA, June, 1990.
- [3] William B Pennebaker, *JPEG Draft Technical Specification Revision 8*, August 17, 1990.
- [4] Gregory K. Wallace, *The JPEG Still Picture Compression Standard*, CACM, Volume 34, No 4, pp 30-44, April 1991.
- [5] James D. Foley, Andries Van Dam, *Fundamentals of Interactive Computer Graphics, Second Edition*, Addison-Wesley Publishing Company.
- [6] K.R. Rao and P. Kip, *Discrete Cosine Transform -- Algorithms, Advantages, Applications*, Academic Press, Inc. London, 1990
- [7] Shih-Fu Chang, Wen-Lung Chen, David G. Messerschmitt, *Video Compositing in the DCT Domain*, Submitted to IEEE ICASSP '92, March 1992
- [8] Wu-Hon F. Leung, Thomas J. Baumgartner, Yeou H. Hwang, Mike J Morgan, Shi-Chuan Tu, *A Software Architecture for Workstations Supporting Multimedia Conferencing in Packet Switching Networks*, IEEE Journal on Selected Areas in Communications, Volume 8, No 3, pp. 380-390, April 1990
- [9] H. Lohscheller. *A subjectively adapted image communication system*, IEEE Trans.

for basic operations on images, many operations cannot be computed as a combination of additions and multiplications. For example, consider the operation of contrast adjustment. In this operation, the value of the luminance component of a pixel in the output image,  $h[i,j]$ , is a function  $\phi$  of the value of the luminance component of the corresponding pixel in the input image,  $f[i,j]$ . The function  $\phi$  is shown in figure 11 below. As can be seen from the figure, if the luminance component of a pixel is below the threshold value  $t_1$ , it is mapped to black, if it is above the threshold value  $t_2$ , it is mapped to white, and if it between the two values, it is linearly scaled. Although this operation is straightforward on uncompressed images, when we applied the techniques used in section 3 to find the corresponding operation on compressed images, the non-linearity of the mapping function  $\phi$  induced mathematical problems to which we could find no simple solution.

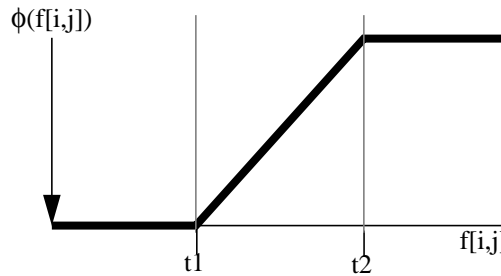


Figure 11: Mapping Function for Contrast Adjustment

To circumvent these difficulties, we tried the following strategy. We know the function  $\phi$  is piecewise linear. If we could quickly determine the range of pixel values in a block, and if those values lay within one of the linear sections of  $\phi$ , then we could use the results of section 3 apply the appropriate linear function to the block. If, however, the range of values of the block spanned linear sections of the mapping function, then we could fall back on the more expensive brute force approach to apply the operation. We expect this strategy would work well since, in most pictures, the pixels within most 8 by 8 blocks are similar, and so the range of values within the block is small. Indeed, this is one of the reasons that transform based coding methods compress data well.

To quickly find the range of values in a block, we used the following method, which we state without proof: the range of values in a block is within the range defined by

$$\frac{DC}{8} \pm \frac{AC}{4}$$

where  $DC$  is the value  $F[0,0]$ , and  $AC$  is the sum of the absolute values of the terms  $F[i,j]$ , where  $[i,j] \neq [0,0]$ . The  $DC$  value is the mean value of the block, derived from *direct current*, which is the mean value of an electrical signal. Similarly, the  $AC$  values are the deviation from the mean value, derived from *alternating current*, which is the deviation from the mean value of an electrical signal.  $AC$  can be computed with a single pass over the SC block with a small number of adds.

When we implemented this method, we found that it ran only 2-4 times faster than the brute force method, depending on the image. The reason is that even if 75% of the blocks have a range of pixels values that allow an SC algorithm to be applied, the remaining 25% must still go through the time consuming compression and decompression of the brute force algorithm. The execution time of decompression and compression is much larger than any other part of the computation, so we would expect the overall execution time to be four times faster if 25% of the data must go through

---

```

static gammaSH[64];

SubtitleInit (fQT, sQT, hQT)
float *fQT, *sQT, *hQT;

{
int x;

ConvolveInit (1/256.0, fQT, sQT, hQT);
for (x=0; x<64; x++)
    gammaSH[x] = sQT[x]/hQT[x];
}

```

Figure 10b: C Code Implementation of SubtitleInit Function

---

Algorithm	Time (sec)	
	Mean	Std Dev
Brute Force	33.84	0.64
Semi Comp.	0.68	0.13

Table 3: Performance Measurements of Dissolve Operation

---

## 5. Discussion

This section describes the problems encountered when we tried to extend the technique to other basic operations, our attempts to circumvent these problems, and suggested areas for further research. It also discusses related work.

Much work has been done on image processing techniques in the spatial domain to perform image enhancement (e.g., [18]). Most of these techniques can be extended into the DCT domain using the results of sections 3 and 4. Duff and Porter [1] suggest that  $\alpha$  channel composition is an effective technique for compositing digital images. The results of sections 3 and 4 show how  $\alpha$  channel techniques can be applied to compressed images.

In the area of DCT domain image processing, Chitprasert and Rao [17] presented a convolution algorithm for the DCT, and showed how it could be used for image processing in certain special cases. His technique is concerned mainly with high and low pass filtering, and doesn't adapt well to the block by block encoding nature of most compression technologies.

Chang [7] apparently independently discovered a technique similar to the one presented this paper for compositing images in the DCT domain. His work is a special case of the results of section 3, and does not contain experiment results. However, it also presents the solution to a related problem. When two images are to be composited and are not correctly aligned, one image may require translation. Chang shows how such a translation operation can be performed in the DCT domain.

## Limitations

Although the algebraic primitives discussed in section 3 represent a solid mathematical basis

as

$$H = \gamma_{s,h} S + \text{Convolve}(M, F, \frac{1}{256}, q_F, q_S, q_H)$$

with

$$M[x] = -S[x] + \frac{1024\delta(x)}{q_S[0]}$$

The C code in figures 10a and 10b implements this operation. The code is divided into two phases, the `SubtitleInit` function, which is called once when the QTs are defined for the image or sequence of images, and the `Subtitle` function, which is called for each SC block in the image. Like the dissolve operation, the `Subtitle` function uses a zig-zag vector `hzz` to stored the intermediate results.

---

```
Subtitle (s, f, h, fQT, sQT, hQT)
SC_Block *s, *f, *h;
float *fQT, *sQT, *hQT;

{
float hzz[64];
SC_Block *tmp;
int x;

Zero (hzz);
for (x=0, tmp=s; tmp != NULL; tmp = tmp->next)
{
x += tmp->skip;
hzz[x] += gammaSH[x]*tmp->value;
tmp->value = -tmp->value;
}
/* Convert S into mask... */
if (s->skip) /* No (0,0) value! */
{
s = NewSC_Block(s->next); /* Insert a 0 value */
s->skip = 0;
s->value = 0;
}
s->value += Round(1024.0/sQT[0]);
Convolve (f, s, hzz);
PartialCompress (hzz, h);
}
```

Figure 10a: C Code Implementation of `Subtitle` Function

---

As with the dissolve operation, the performance of programs that implemented the brute force algorithm and the SC algorithm on images resident in main memory were compared. The test parameters were the same as with the dissolve operation. Table 3 summarizes the results, showing a speedup of nearly 50 to 1 over the brute force algorithm.



Figure 9: Two sample images and their mixing with  $\alpha = 0.5$  using  
a) the brute for algorithm, and b) the semi-compressed algorithm

## The Subtitle Operation

The second example operation is *subtitle*, which overlays a subtitle on a compressed image  $f$ . Although a workstation could support this operation in many ways (such as displaying the text of the subtitle in a separate window), we chose this operation because it is a common operation which most people are familiar with and it serves as a specific example of the common operation of *image masking*, which is used when a portion of one image is to be combined with another image. Another example where this technique is commonly used is in the *chroma-key* operation. In this operation, a foreground image (e.g., a weather forecaster) is overlaid onto a background image (e.g., a weather map). The mask is generated by positioning the foreground object (i.e., the weather forecaster) in front of a bright blue or green screen. Only the non-blue (or green) pixels of this image are present in the mask.

The subtitle is assumed to be a compressed image of white letters on a black background denoted  $S$ , with white and black represented by pixel values 127 and -128, respectively (these are the values obtained from the luminance component of an image). The output image can be constructed by adding together  $S$  and an image obtained by multiplying  $f$  by a mask that will blacken the areas on  $f$  where the text should go. Such a mask,  $m$ , can be constructed from  $S$  if we define  $m[i, j] = 127 - s[i, j]$ . The output image with subtitling,  $h$ , is then given by:

$$h[i, j] = s[i, j] + \frac{1}{256} (128 - s[i, j]) f[i, j] \quad (\text{EQ 15})$$

Using table 1, we see that the corresponding operation on an SC block is symbolically represented

defined by

$$\text{gamma1}[x] = \alpha \gamma_{S1,D}(x)$$

$$\text{gamma2}[x] = (1 - \alpha) \gamma_{S2,D}(x)$$

These values can be precomputed once for each image or sequence of images with the same QTs, whereas the Dissolve function is called for each SC block in an image.

---

```
Dissolve (f, g, h, gamma1, gamma2)
SC_Block *f, *g, *h;
float *gamma1, *gamma2;

{
float hzz[64];
int x;

Zero (hzz);
for (x=0; f != NULL; f = f->next)
{
x += f->skip;
hzz[x] += gamma1[x]*f->value;
}
for (x=0; g != NULL; g = g->next)
{
x += g->skip;
hzz[x] += gamma2[x]*g->value;
}
PartialCompress (hzz, h);
}
```

Figure 8: C Implementation of the Dissolve Operation

---

To test the performance of this implementation, we wrote programs that executed both the brute force and the SC algorithm on images resident in main memory and compared the performance. Both algorithms were executed on 25 separate pairs of images on a sparcstation 1+ with 28 MBytes of memory. The test images were 640 X 480, and 24 bits per pixel. A sample image pair is shown in Figure 9. The images were compressed to approximately one bit per pixel (24 to 1 compression). Table 2 summarizes the results. As can be seen from the table, the speedup of the semi-compressed algorithm is more than 100 to 1 over the brute force algorithm.

---

Algorithm	Time (sec)	
	Mean	Std Dev
Brute Force	36.86	0.01
Semi Comp.	0.34	0.00

Table 2: Performance Measurements of Dissolve Operation

---

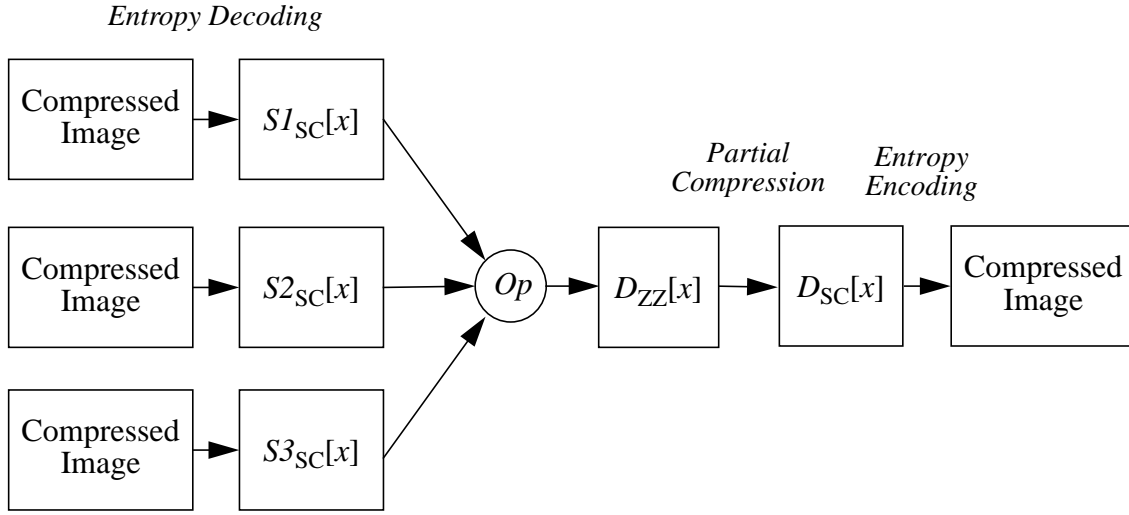


Figure 7: Strategy for Manipulating Images

---

The remainder of this section presents two examples that illustrate this strategy and compares the performance of these new algorithms with the brute force algorithm.

## The Dissolve Operation

The entropy encoding and decoding steps will be omitted to simplify the presentation. Suppose a sequence of images  $S1[t]$  is to be dissolved into a sequence of images  $S2[t]$  in a time  $\Delta t$  (typically 0.25 seconds). In other words, at  $t=0$  we should display  $S1[0]$ , at  $t=\Delta t$ , we should display  $S2[\Delta t]$ , and in between we want to display the linear combination of the images:

$$D[t] = \alpha(t) S1[t] + \{1 - \alpha(t)\} S2[t] \quad (\text{EQ 14})$$

where  $\alpha(t)$  is a linear function that is 1 at  $t=0$  and 0 at  $t=\Delta t$ .

Using table 1, we can map this operation into the corresponding operation on SC blocks as follows. From the table, we know that the scalar multiplies can be performed directly on the SC blocks if the coefficient  $\alpha$  is changed to  $\alpha\gamma_{S1,D}(x)$  in the first half of the expression and a similar substitution is performed for the multiplication by  $\{1 - \alpha\}$ . Also from the table, we know we can add the coefficients together directly to get the desired result, since the QTs of these two new SC blocks are the same, namely  $q_D(x)$ . Thus, the expression in equation 14 can be implemented as

$$D_{ZZ}[x] = \alpha\gamma_{S1,D}(x) S1_{ZZ}[x] + \{1 - \alpha\}\gamma_{S2,D}(x) S2_{ZZ}[x]$$

We can implement this equation efficiently by noticing that the SC format of the data will skip over zero terms. The C code to implement this operation on an SC block is shown in figure 8. The function `Zero` zeros the array passed to it, and the function `PartialCompress` performs the partial compression step into `h`. The arrays `gamma1` and `gamma2` have the precomputed values

## Summary of Operations

This section showed how pixel addition, pixel multiplication, scalar addition, and scalar multiplication can be implemented on quantized matrices. As noted earlier, these transformations can be implemented using zig-zag ordering, so they can operate directly on SC blocks. Table 1 summarizes the mapping of image operations into operations on SC blocks. In the table, the symbol  $\gamma_{F,H}(x)$  is defined as

$$\gamma_{F,H}(x) \equiv \frac{q_F[x]}{q_H[x]} \equiv \frac{q_F[u,v]}{q_H[u,v]}$$

and the function  $Convolve(F, G, \alpha, q_F, q_G, q_H)$  is defined in equation 13 and implemented in figures 5 and 6.

Operation	Image Space Definition for $h[i, j]$	SC Definition for $H_{ZZ}[x]$
Scalar Multiplication	$\alpha f[i, j]$	$\alpha \gamma_{F,H}(x) F_{ZZ}[x]$
Scalar Addition	$f[i, j] + \beta$	$\gamma_{F,H}(x) F_{ZZ}[x] + \frac{8\beta\delta(x)}{q_H[0]}$
Pixel Addition	$f[i, j] + g[i, j]$	$\gamma_{F,H}(x) F_{ZZ}[x] + \gamma_{G,H}(x) G_{ZZ}[x]$
Pixel Multiplication	$f[i, j] g[i, j]$	$Convolve(F, G, \alpha, q_F, q_G, q_H)$

Table 1: Mapping of Operations

## 4. Applications

Video data is typically transmitted as a sequence of compressed images. While the entropy encoded data cannot be directly manipulated, section 3 showed how pixel or scalar addition and multiplication can be performed on SC blocks. Thus, if we entropy decode the images, perform the operation, and then entropy encode the result, we can shortcut the rest of the decoding and encoding of the images which will be a faster algorithm.

These simple operations can be combined to form more powerful operations such as *dissolve* (the simultaneous fade out and fade in of two sequences of images) and subtitling. The implementation of these operations typically involves the computation of an output image that is an algebraic combination of one or more input images. One way to perform the combination on a pair of SC blocks is to use zig-zag vectors as the intermediate representation to compute the expression. Thus, to multiply two SC blocks and add a third, we would zero the zig-zag vector, call the *Convolve* function of figure 6 on the first two SC blocks (the zig-zag vector is  $hzz$  in the figure), add the third SC block to the zig-zag vector, and then perform the partial compression and entropy coding steps. Figure 7 graphically depicts our strategy.

---

```

ConvolveInit (alpha, fQT, gQT, hQT)
float alpha, *fQT, *gQT, *hQT;

{
int u1, u2, v1, v2, w1, w2;
int x, y, z;
float t1, t2;

for (u1=0; u1<8; u1++)
  for (v1=0; v1<8; v1++)
    for (w1=0; w1<8; w1++)
      if ((t1 = W[u1][v1][w1]) != 0.0)
        for (u2=0; u2<8; u2++)
          for (v2=0; v2<8; v2++)
            for (w2=0; w2<8; w2++)
              if ((t2 = W[u2][v2][w2]) != 0.0)
                {
x = ZigZag(u1, u2);
y = ZigZag(v1, v2);
z = ZigZag(w1, w2);
W = t1*t2*alpha*fQT[x]*gQT[y]/hQT[z];
comb[x,y] = AddCombElt(z,W,comb[x,y]);
                }
}

```

Figure 5: Initialization of the Combination Array

---

```

Convolve (f, g, hzz)
SC_Block *f, *g; /* The input images */
float *hzz; /* Array of 64 elements */

{
int x, y, z;
float W, tmp;
SC_BLOCK *gtmp;
COMB_LIST *cl;

for (x=0; f != NULL; f = f->next) {
  x += f->skip;
  for (y=0, gtmp = g; gtmp != NULL; gtmp = gtmp->next) {
    y += gtmp->skip;
    tmp = f->val*gtmp->val;
    for (cl = comb[x,y]; cl != NULL; cl = cl->next) {
      z = cl->z;
      W = cl->W;
      hzz[z] += tmp*W;
    }
  }
}
}

```

Figure 6: Implementation of the Convolve Function

---

$W_Q$  is very sparse.

When we implement this method, care must be taken to evaluate only those terms that might contribute to the sum. We take advantage of (1) when we implement this method on SC blocks, since the zeros are easily skipped. To take advantage of (2), the data structure described in the following paragraphs is used. Since the algorithm operates on SC blocks, the zig-zag ordered indices are used to reference data elements. By convention, we will let  $x$ ,  $y$  and  $z$  represent the zig-zag ordered indices of the pairs  $(v_1, v_2)$ ,  $(w_1, w_2)$  and  $(u_1, u_2)$ , respectively. With this substitution, equation 12 can be written as

$$H(z) = \sum_{x,y} F(x) G(y) W_Q(v_1, v_2, w_1, w_2, u_1, u_2) \quad (\text{EQ 13})$$

with the sum running for  $x$  and  $y$  running from 0 to 63.

We introduce the following data structure to compute equation 13 efficiently. A *combination element* is a pair of numbers  $z$  and  $W$ , where  $z$  is an integer and  $W$  is a floating point value. A *combination list* is a list of combination elements. A *combination array*, is a 64 by 64 array of combination lists. The C code shown in figure 5 initializes the combination array `comb[x, y]`. The array contains empty lists when the code is entered. The function `AddCombElt(z, W, comb[x, y])` appends the combination element  $(z, W)$  to the combination list stored in the global combination array `comb[x, y]` and returns the modified combination list. The array `W[8][8][8]` is assumed to be initialized with the values of the  $W$  function of equation 12.

Using the initialized combination array, the C code shown in figure 6 efficiently implements equation 13 on two SC blocks  $f$  and  $g$ . We call this algorithm the *convolution algorithm*. Notice that `comb[ ]` is a constant in the code; it need only be computed once for the given QTs.

The code operates as follows: The array `hzz`, which represents a zig-zag vector, is assumed to be all zero. For each pair of SC values in the two input images  $f$  and  $g$ , we compute the zig-zag indices  $x$  and  $y$ , and the product of their data values, which is stored in `tmp`. We then use the  $z$  value of each combination element in the combination list stored in `comb[x, y]` to determine which elements in the output array `hzz` are affected and accumulate the product  $W * tmp$  into each element. In this way, only the multiplies which result in non-zero products accumulate in `hzz`. When used in a program, a final pass is needed to run-length encode the zeros, perform integer rounding, and construct the resulting SC block.

where  $\alpha$  is a scalar value. The scalar  $\alpha$ , although mathematically superfluous, is convenient to scale pixel values as they are multiplied. This formulation is used, for example, when the image  $g$  contains pixel values in the range [0..255] and we want to interpret them as the range [0..1), as is the case when  $g$  is a mask. This operation is realized by setting  $\alpha$  to 1/256.

Let  $F(v_1, v_2)$ ,  $(w_1, w_2)$  and  $H(u_1, u_2)$  be the quantized values of the compressed images for  $f$ ,  $g$ , and  $h$ , respectively. Then using equations (1), (2) and (11), we can compute the value of  $H(u_1, u_2)$  as follows:

$$\begin{aligned}
H(u_1, u_2) &= \frac{1}{4q_H[u_1, u_2]} \sum_i \sum_j C(i, u_1) C(j, u_2) h(i, j) \\
&= \frac{\alpha}{4q_H[u_1, u_2]} \sum_i \sum_j C(i, u_1) C(j, u_2) f(i, j) g(i, j) \\
&= \frac{\alpha}{4q_H[u_1, u_2]} \sum_i \sum_j C(i, u_1) C(j, u_2) \\
&\quad \left( \frac{1}{4} \sum_{v_1} \sum_{v_2} C(i, v_1) C(j, v_2) q_F[v_1, v_2] F(v_1, v_2) \right) \\
&\quad \left( \frac{1}{4} \sum_{w_1} \sum_{w_2} C(i, w_1) C(j, w_2) q_G[w_1, w_2] G(w_1, w_2) \right) \\
&= \sum_{v_1, v_2, w_1, w_2} F(v_1, v_2) G(w_1, w_2) W_Q(v_1, v_2, w_1, w_2, u_1, u_2) \quad (\text{EQ 12})
\end{aligned}$$

where

$$\begin{aligned}
W_Q(v_1, v_2, w_1, w_2, u_1, u_2) \\
&= \frac{\alpha q_F[v_1, v_2] q_G[w_1, w_2]}{64q_H[u_1, u_2]} W(u_1, v_1, w_1) W(u_2, v_2, w_2)
\end{aligned}$$

with

$$W(u, v, w) = \sum_i C(i, u) C(i, v) C(i, w)$$

We can compute this rather lengthy sum efficiently by noticing several facts:

- (1) for typical compressed images,  $G(w_1, w_2)$  and  $F(v_1, v_2)$  are zero for most values of  $(v_1, v_2)$  and  $(w_1, w_2)$ .
- (2) Of the 256K elements in the function  $W_Q(v_1, v_2, w_1, w_2, u_1, u_2)$ , only about 4% of the terms are non-zero. In other words, the matrix

assumes a particularly simple form expressed in the equations:

$$H_Q[0, 0] = \gamma F_Q[0, 0] + \frac{8\beta}{q_H(0, 0)} \quad (\text{EQ 8b})$$

$$H_Q[u, v] = \gamma F_Q[u, v] \quad (\text{EQ 8c})$$

Again we see that the operation of scalar addition can be performed directly on the quantized coefficients. More importantly, in the common case where  $\gamma=1$ , i.e., when the quality of the output image is the same as the quality of the input image, this operation involves much less computation than the corresponding operation on uncompressed images, since only the (0,0) coefficient of the quantized matrix is affected.

## Pixel Addition

The operation of pixel addition is described by the equation

$$h[i, j] = f[i, j] + g[i, j] \quad (\text{EQ 9})$$

Using equations (1), (2) and (9), the quantized coefficients in the output image are

$$\begin{aligned} H_Q[u, v] &= \frac{1}{4q_H(u, v)} \sum_i \sum_j C(i, u) C(j, v) h[i, j] \\ &= \frac{1}{4q_H(u, v)} \sum_i \sum_j C(i, u) C(j, v) (f[i, j] + g[i, j]) \\ &= \frac{q_F(u, v)}{q_H(u, v)} F_Q[u, v] + \frac{q_G(u, v)}{q_H(u, v)} G_Q[u, v] \end{aligned} \quad (\text{EQ 10a})$$

Once again we see that if we account for the QTs of the images, the operation can be performed directly on the quantized coefficients, and that if the QTs for all the images are proportional (with proportionality constants  $\gamma_F$  and  $\gamma_G$ ), the equation is simplified to:

$$H_Q[u, v] = \gamma_F F_Q[u, v] + \gamma_G G_Q[u, v] \quad (\text{EQ 10b})$$

## Pixel Multiplication

Finally, the operation of pixel multiplication is expressed in the equation

$$h[i, j] = \alpha f[i, j] g[i, j] \quad (\text{EQ 11})$$

zag vector. When implemented this way, useless multiplies where  $F_Q[u, v]$  is zero are avoided. For these reasons, the operation is very fast.

## Scalar Addition

Now consider the operation of scalar addition. In this operation, if the value of a pixel in the original image is  $f[i, j]$ , the value of the corresponding pixel in the output image  $h[i, j]$  is given by

$$h[i, j] = f[i, j] + \beta \quad (\text{EQ 6})$$

Using equations (1), (2) and (6), the quantized coefficients in the output image are

$$\begin{aligned} H_Q[u, v] &= \frac{1}{4q_H(u, v)} \sum_i \sum_j C(i, u) C(j, v) h[i, j] \\ &= \frac{1}{4q_H(u, v)} \sum_i \sum_j C(i, u) C(j, v) (f[i, j] + \beta) \\ &= \frac{q_F(u, v)}{q_H(u, v)} (F_Q[u, v] + \frac{\beta}{4q_F(u, v)} \sum_i \sum_j C(i, u) C(j, v)) \quad (\text{EQ 7}) \end{aligned}$$

The last term can be summed explicitly. If we use the result

$$\sum_i \sum_j C(i, u) C(j, v) = 32\delta(u) \delta(v)$$

where

$$\delta(u) = \begin{cases} 1 & u = 0 \\ 0 & u \neq 0 \end{cases}$$

we can simplify equation 7 to

$$H_Q[u, v] = \frac{q_F(u, v)}{q_H(u, v)} F_Q[u, v] + \frac{8\beta}{q_H(u, v)} \delta(u) \delta(v) \quad (\text{EQ 8a})$$

If the QTs of both images are proportional, with proportionality constant  $\gamma$ , this equation

QT of the compressed image  $H$ , and the letters  $x$ ,  $y$ , and  $z$  will represent zig-zag ordered indices. Often we will index QT's by a single zig-zag index (such as  $x$ ). In such cases the conversion to indices such as  $[u, v]$  is implied and will be clear from context.

## Scalar Multiplication

Consider the operation of scalar multiplication of pixel values. In this operation, if the value of a pixel in the original image is  $f[i, j]$ , the value of the corresponding pixel in the output image  $h[i, j]$  is given by

$$h[i, j] = \alpha f[i, j] \quad (\text{EQ 4})$$

Using equations (1), (2) and (4), the quantized coefficients in the output image are

$$\begin{aligned} H_Q[u, v] &= \frac{1}{4q_H(u, v)} \sum_i \sum_j C(i, u) C(j, v) h[i, j] \\ &= \frac{\alpha}{4q_H(u, v)} \sum_i \sum_j C(i, u) C(j, v) f[i, j] \\ &= \frac{\alpha q_F(u, v)}{q_H(u, v)} \left( \frac{1}{4q_F(u, v)} \sum_i \sum_j C(i, u) C(j, v) f[i, j] \right) \end{aligned}$$

where  $q_F(u, v)$  and  $q_H(u, v)$  are the QTs of the input and output images, respectively. Recognize that the term in parentheses is the quantized coefficients of the input image,  $F_Q[u, v]$ , so this becomes

$$H_Q[u, v] = \frac{\alpha q_F(u, v)}{q_H(u, v)} F_Q[u, v] \quad (\text{EQ 5})$$

with the final integer rounding of the right hand side implicit. In other words, to perform the operation of scalar multiplication on a compressed image, we can perform it directly on the quantized coefficients, as long as we take the QTs of the images into account. Note that if the QTs of both images are proportional with proportionality constant  $\gamma$ , the equation simplifies to

$$H_Q[u, v] = \alpha \gamma F_Q[u, v]$$

If the quality of the input and output images are the same, we have the common special case where  $\gamma=1$ . Note also that if a value in the input,  $F_Q[u, v]$ , is zero, the corresponding value in the output,  $H_Q[u, v]$ , is also zero. Thus, we can implement this operation on an SC block by simply scaling the values in the data structure - there is no need to reconstruct the quantized array or even the zig-

DCT (IDCT):

$$y [i, j] = \frac{1}{4} \sum_u \sum_v C (i, u) C (j, v) Y [u, v] \quad (\text{EQ 3})$$

Note that equation 3 is very similar to equation 1, but the summation is over  $u$  and  $v$  rather than  $i$  and  $j$ . Figure 4 graphically depicts this process.

Using this method, the compression ratio can be adjusted by altering the values in the QTs. Experience indicates that a compression ratio of about 24 to 1 (i.e., one bit/pixel) can be achieved without serious loss of image quality. At a compression ratio of 10 to 1, the decompressed image is usually indistinguishable from the original. The entropy coding reduces the data size by about 2.5 to 1, so the data size of the SC blocks is typically 10 times smaller than that of the original image, if we assume 25 to 1 overall compression.

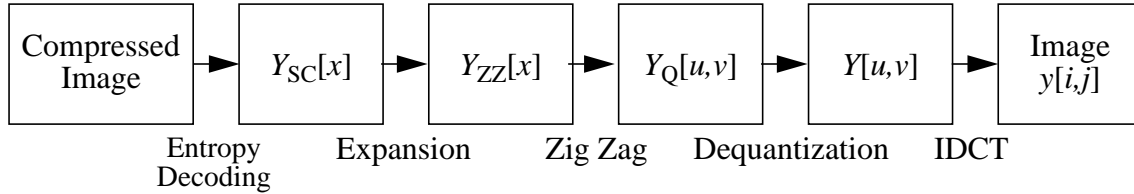


Figure 4: Decompression of a Block

### 3. Algebraic Operations

This section shows how the four algebraic operations of scalar addition, scalar multiplication, pixel-wise addition and pixel-wise multiplication of two images are performed on SC blocks.

In the calculations that follow, we will be deriving equations of the form

$$H_{SC} = \phi (F_{SC}, G_{SC})$$

where  $F_{SC}$  and  $G_{SC}$  are the SC representations of the input images,  $H_{SC}$  is the SC representation of the output image, and  $\phi$  is a real-valued function. In an implementation, the values stored in the  $H_{SC}$  data structure would be integers, so the value returned by the function  $\phi$  must be rounded to the nearest integer. To simplify the notation in the calculations that follow, this rounding will be implicit.

To further simplify the notation, we perform all calculations on the quantized arrays  $F_Q [u, v]$ ,  $G_Q [u, v]$  and  $H_Q [u, v]$ . Since an SC block is a data structure that represents these arrays, the derived equations will be valid on SC blocks provided the appropriate index conversion is performed.

Finally, we will use capital letters such as  $F$ ,  $G$  and  $H$  indexed by  $u$ ,  $v$  and  $w$  to represent compressed images and lower case letters such as  $f$ ,  $g$ , and  $h$  indexed by  $i$ ,  $j$  and  $k$  to represent uncompressed images. Greek letters such as  $\alpha$  and  $\beta$  will be used for scalars,  $q_H [u, v]$  will stand for the

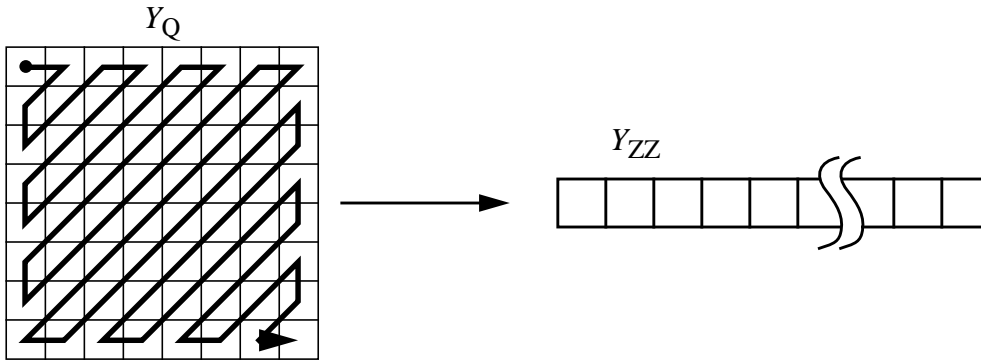


Figure 2: Zig Zag Scan Ordering

In most images, the zig-zag vector  $Y_{ZZ}$  will contain a large number of sequential zeros, so the next step in the algorithm, called the *partial compression* step, run length encodes the zeros into an array of (*skip, value*) pairs. *Skip* indicates how many indices in the  $Y_{ZZ}$  vector to skip to reach the next non-zero value, which is stored in *value*. By convention, the pair (0,0) indicates that the remaining values in  $Y_{ZZ}$  are all zero. We call the block at this point a *semi-compressed* (SC) block. A (*skip, value*) pair is called an *SC value*, and sequences of SC values that represent a zig-zag vector is called an *SC block*. The SC block is denoted  $Y_{SC}[x]$ , with  $Y_{SC}[x].skip$  and  $Y_{SC}[x].value$  denoting the *skip* and *value* of the  $x$ th element in the array. Our algorithms will operate on SC blocks.

In the final step, a conventional entropy coding method such as arithmetic compression or Huffman coding compresses the SC blocks. Figure 3 graphically displays all the steps in the processing of one block.

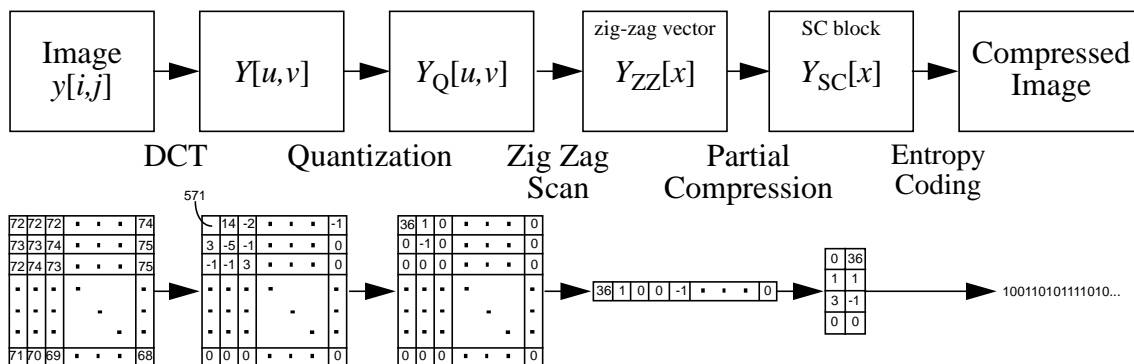


Figure 3: Compression of a Block

The process is reversed to decompress the data. The first step of decompression recovers an SC block from the entropy coded bit stream. By making a single pass through the SC block  $Y_{SC}[x]$ , we recover the zig-zag vector  $Y_{ZZ}[x]$ . From  $Y_{ZZ}[x]$  we recover  $Y_Q[u,v]$  by inverting the zig-zag scan. Then we multiply each element of  $Y_Q[u,v]$  by  $q[u,v]$  from the appropriate QT to recover an approximation of  $Y[u,v]$ . In the final step, we obtain the image block  $y[i,j]$  from  $Y[u,v]$  using the inverse

The second step in the algorithm converts this 8 by 8 matrix into a new 8 by 8 matrix using the two dimensional discrete cosine transform (DCT) [6]. This step is called the *DCT* step. If we call the new matrix  $Y[u, v]$ , with  $u, v \in 0 \dots 7$ , we have by definition

$$Y[u, v] = \frac{1}{4} \sum_i \sum_j C(i, u) C(j, v) y[i, j] \quad (\text{EQ 1})$$

where

$$C(i, u) = A(u) \cos \frac{(2i+1)u\pi}{16}$$

$$A(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{for } u \neq 0 \end{cases}$$

The third step in the algorithm divides each element of  $Y[u, v]$  by an integer and rounds the result. This step, called *quantization*, is defined by

$$Y_Q[u, v] = \text{IntegerRound} \left( \frac{Y[u, v]}{q[u, v]} \right) \quad u, v \in 0 \dots 7 \quad (\text{EQ 2})$$

The matrix of integers  $q[u, v]$  is called the *quantization table* (QT). In most images, different QTs are used for the luminance and chrominance components. The choice of the QT determines both the amount of compression and the quality of the decompressed image [4]. The JPEG standard includes recommended luminance and chrominance QTs. Very often, different image qualities are obtained by scaling the values of the default QTs: in other words, given two images with QTs  $q_1[u, v]$  and  $q_2[u, v]$ , then for all  $u, v$  and some constant gamma, it is very often the case that

$$\frac{q_1[u, v]}{q_2[u, v]} = \gamma$$

We will use this fact later.

Step four of the algorithm converts the 8 by 8 matrix  $Y_Q[u, v]$  into a 64 element vector  $Y_{ZZ}[x]$  using the “zig zag” ordering shown in Figure 2 below. The vector  $Y_{ZZ}$  is called the *zig-zag vector*, and this step is called the *zigzag scan* step. By definition, the function  $\text{ZigZag}(u, v)$  returns the zig-zag ordered index of the pair  $(u, v)$ . For example,  $\text{ZigZag}(0, 0) = 1$ , and  $\text{ZigZag}(0, 3) = 6$ .

the compressed data in the DCT domain. Since the volume of data has been significantly reduced by compression, typically by factors of 25 or more, algorithms in this family can run much faster than the corresponding brute force algorithms. Along with the speedup resulting from the smaller data volume, most of the computation associated with compression and decompression is eliminated, and the traffic to and from memory is reduced.

This paper describes how to apply this technique and evaluates the performance of some representative algorithms. Section 2 describes the compression model and introduces associated terminology. Section 3 shows how the algebraic operations of pixel-wise and scalar addition and multiplication, which are the basis for many image transformations, can be implemented on compressed images. Section 4 uses these operations to implement two common video transformations: dissolving one video sequence into another and subtitling. The operations have been implemented and their performance is compared with the brute force approach. Lastly, section 5 discusses the limitations of the technique, extensions to other compression standards, and the relationship of this research to other work in the area.

## 2. Compression Model

This section describes the compression model used in transform based coding. A detailed description of the CCITT Joint Photographic Expert Group (JPEG) proposed standard for transform based coding is available elsewhere ([3], [4]). A detailed description of image formats is presented by Foley and Van Dam [5].

The discussion presented here is simplified by choosing a specific source image format. Suppose the source image is a 24 bit image 640 pixels wide by 480 pixels high, and is composed of three components, one *luminance* ( $Y$ ) and two *chrominance* ( $I$  and  $Q$ ) components. That is, for each pixel in the source image, we associate a triplet of 8 bit values ( $Y, I, Q$ ). Since each component is treated similarly, we will describe the algorithm for only one component (e.g., the  $Y$  component).

The  $Y$  component can be broken up into contiguous squares 8 pixels wide and 8 pixels high called *blocks*. Each block is an 8 by 8 matrix of integers in the range  $0 \dots 255$ . The first step of the algorithm, called the *normalization* step, brings all values into the range  $-128 \dots 127$  by subtracting 128 from each element in the matrix (this step is skipped on the  $I$  and  $Q$  components since they are already in the range  $-128 \dots 127$ ). Let the resulting matrix be  $y[i, j]$ , where  $i, j \in 0 \dots 7$ . Figure 1 illustrates the relationship of  $y[i, j]$  to the whole image.

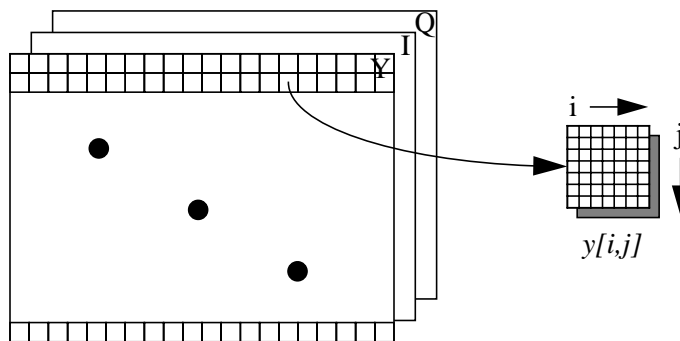


Figure 1. Definition of  $y[i, j]$ .

# A New Family of Algorithms for Manipulating Compressed Images<sup>1</sup>

*Brian C. Smith*  
*Lawrence A. Rowe*

Computer Science Division-EECS  
University of California  
Berkeley, CA 94720

## Abstract

This paper describes a new technique to implement operations on compressed digital video images that allows many image manipulation operations to be performed 50 to 100 times faster than the corresponding algorithms operating on uncompressed images. This is accomplished by performing the operations directly on the compressed data in the DCT domain. In this paper, we show how to transform image space operations into DCT space operations, we describe several representative algorithms in the DCT domain and report their performance.

## 1. Introduction

Multimedia applications that operate on audio and video data will enable many new applications of computers. For example, a collaborative work system can include video conferencing windows, or a hypermedia training system can include audio and video instructional material. While most research on multimedia applications has focused on compression standards ([3], [9], [10], [14]), synchronization issues ([11], [12]), storage representations ([12]), software architectures ([2], [8], [11]), and application design ([13], [15], [16]), little work has been reported on techniques for manipulating digital video data in real time, such as implementing special effects and image composition. The difficulties encountered in this problem area stem from several sources: the volume of data to be manipulated, the computational complexity of image compression and decompression, and video data rates (i.e., 27 MBytes/sec) all combine to make traditional algorithms, which operate on uncompressed images, infeasible on current workstations. For example, an algorithm to implement brightening of a compressed image might decompress the image, modify each pixel value, and compress the resulting image. Such algorithms are *brute force* algorithms.

This paper describes a new technique to implement operations on digital video data that allows many traditional image manipulation operations to be performed 50 to 100 times faster than the corresponding brute force algorithms. This is accomplished by performing the operations directly on

---

1. This research was supported by NSF grant MIP-9014940 and a grant from the Semiconductor Research Corporation.