Fast Algorithms for DCT-Domain Image Down-Sampling and for Inverse Motion Compensation

Neri Merhav* Vasudev Bhaskaran†

Abstract

Straightforward techniques for spatial domain processing of compressed video via decompression and re-compression are computationally expensive. We describe an alternative approach wherein the compressed stream is processed in the compressed, DCT domain without explicit decompression and spatial domain processing, so that the output compressed stream corresponds to the output image and it conforms to the standard syntax of 8×8 blocks. We propose computation schemes for down-sampling and for inverse motion compensation, that are applicable to any DCT-based compression method. Worst-case estimates of computation savings vary between 37% and 50% depending on the task. For typically sparse DCT blocks, the reduction in computations is more dramatic. A byproduct of the proposed approach is improvement in arithmetic precision.

*N. Merhav is with the Department of Electrical Engineering and HP Israel Science Center, Technion City, Haifa 32000, Israel. He is currently on Sabbatical leave at HP Laboratories, 1501 Page Mill Road, Palo Alto CA 94304, USA.

†V. Bhaskaran is with the Visual Computing Department, HP Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, USA.

1 Introduction

Many video compression methods, like MPEG and H.261, use transform domain techniques, in particular, the discrete cosine transform (DCT). Certain applications require real time manipulation of digital video in order to implement image composition and special effects, e.g., down-sampling, modifying contrast and brightness, translating, filtering, masking, rotation, inverse motion compensation, etc. There are two major difficulties encountered in this class of tasks: the computational complexity of image compression and decompression, and the high rates of the data to be manipulated. These difficulties rule out the possibility of running, on currently existing workstations, the traditional algorithms that first decompress the data, then perform one of these manipulations in the decompressed domain, and finally, compress again if necessary. For this reason there has been a great effort in recent years to develop fast algorithms that perform these tasks directly in the transform domain (see e.g., [?], [?], [?] and references therein) and thereby avoid the need of decompression or at least its computational bottleneck - the inverse DCT (IDCT) which requires 38.7% of the execution time on a typical workstation[?].

As an example, consider a video conferencing session of several parties, where each one of them can see everybody else in a separate window on his screen. Every user would like to have the flexibility to resize windows, move them from one location on the screen to another, and so on. Since each workstation is capable of handling one video stream only, the server must compose the streams from all parties to a single stream whose architecture depends on the user's requests. If one user wishes, say, to scale down by a factor of 2 a window corresponding to another user and move it to a different place on the screen, this might affect the entire image. This simple operation requires at least two processing functions to be carried out, one is the down-sampling operation and the other is inverse motion compensation, which removes dependency between successively encoded frames and hence enables the composition of two or more digital video streams. It should be pointed out that the operation of inverse motion compensation is useful not only in compositing video streams, but might have other applications as well, such as video editing (e.g., trimming) and transcoding from MPEG to JPEG.[1]

The traditional and expensive approach would be that all compressed video streams are

[1] In this case, the system depicted in Fig. ??(c) would have to be applied to all frames prior to the new start frame, back to the last intra-coded frame.

first fully decompressed at the server, then the desired change is translated into a suitable arithmetic operation on the decompressed video streams with the appropriate composition into a single stream, and finally, the composite stream is compressed again and sent to the user. A great deal of the computational load is in the DCT and IDCT operations and this drives us to seek fast algorithms that perform the desired modifications directly in the DCT domain (see Fig. **??**).

In this work, we focus on speeding up two types of processing operations and compare to the traditional approach. The first is down-sampling and the second is inverse motion compensation. Since both kinds of operations are linear, the overall effect in the DCT domain is linear as well and hence the basic operation can be represented as multiplication by a fixed matrix. Fast multiplication by a fixed matrix is possible if it can be factorized into a product of sparse matrices whose entries are mostly 0, 1 and -1. We will demonstrate that this can be done efficiently for both tasks of down-sampling and inverse motion compensation by taking advantage of the factorizations of the DCT and IDCT operation matrices that correspond to the fast 8-point Winograd DCT/IDCT due to Arai, Agui, and Nakajima [**?**] (see also [**?**]).

The resulting schemes for down-sampling save about 37% of the operations[2] for a down-sampling factor of 2, 39% for down-sampling by 3, 50% for a factor of 4, and 47% for inverse motion compensation. These are 'worst-case' estimates in the sense that nothing is assumed on sparseness in the DCT domain. Typically, in a considerably large percentage of the DCT blocks all the DCT coefficients are zero except for the upper left 4×4 quadrant that corresponds to low frequencies in both vertical and horizontal directions. If this fact is taken into account, then computation reductions can reach about 70-80%.

Another advantage of the proposed method is that it improves the precision of the computations as compared to the traditional approach. The reason for this will become apparent later on when we describe the method in detail. The degree of improvement in precision varies between 1.5-3dB.

The outline of the paper is as follows. In Section 2 we provide some preliminaries and describe the problems of down-sampling and inverse motion compensation. In Section 3, we provide a detailed derivation of the down-sampling method by a factor of 2, first, in

[2]Here the term "operation" corresponds to the basic arithmetic operation of a typical processor which is either "shift", "add", "shift-one-and-add" (SH1ADD), "shift-two-and-add" (SH2ADD), and "shift-three-and add" (SH3ADD).

the one-dimensional case and then in the two dimensional case. We also demonstrate how our method improves both computational complexity and arithmetic precision. In Section 4, we derive the inverse motion compensation algorithm and evaluate its computational complexity.

2 Preliminaries and Problem Description

The 8×8 2D-DCT transforms a block $\{x(n,m)\}_{n,m=0}^{7}$ in the spatial domain into a matrix of frequency components $\{X(k,l)\}_{k,l=0}^{7}$ according to the following equation

$$X(k,l) = \frac{c(k)}{2}\frac{c(l)}{2}\sum_{n=0}^{7}\sum_{m=0}^{7} x(n,m)\cos(\frac{2n+1}{16}\cdot k\pi)\cos(\frac{2m+1}{16}\cdot l\pi) \tag{1}$$

where $c(0) = 1/\sqrt{2}$ and $c(k) = 1$ for $k > 0$. The inverse transform is given by

$$x(n,m) = \sum_{k=0}^{7}\sum_{l=0}^{7}\frac{c(k)}{2}\frac{c(l)}{2}X(k,l)\cos(\frac{2n+1}{16}\cdot k\pi)\cos(\frac{2m+1}{16}\cdot l\pi). \tag{2}$$

In a matrix form, let $\boldsymbol{x} = \{x(n,m)\}_{n,m=0}^{7}$ and $\boldsymbol{X} = \{X(k,l)\}_{k,l=0}^{7}$. Define the 8-point DCT matrix $S = \{s(k,n)\}_{k,n=0}^{7}$, where

$$s(k,n) = \frac{c(k)}{2}\cos(\frac{2n+1}{16}\cdot k\pi). \tag{3}$$

Then,

$$\boldsymbol{X} = S\boldsymbol{x}S^{t} \tag{4}$$

where the superscript t denotes matrix transposition. Similarly, let the superscript $-t$ denote transposition of the inverse. Then,

$$\boldsymbol{x} = S^{-1}\boldsymbol{X}S^{-t} = S^{t}\boldsymbol{X}S \tag{5}$$

where the second equality follows from the unitarity of S. We next give a formal description of the problems of down-sampling and inverse motion compensation.

Down-sampling: Suppose we are given four adjacent 8×8 spatial domain data blocks $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$, and $\boldsymbol{x}_4$ that together form a 16×16 square, where $\boldsymbol{x}_1$ corresponds to northwest, $\boldsymbol{x}_2$ to northeast, $\boldsymbol{x}_3$ to southwest and $\boldsymbol{x}_4$ to southeast. Down-sampling (decimation) by a factor of 2 in each dimension means that every non-overlapping group of 4 pixels forming a small 2×2 block is replaced by one pixel whose intensity is the average of the 4 original

pixels.[3] As a result, the original blocks $\boldsymbol{x}_1, ..., \boldsymbol{x}_4$ are replaced by a single 8×8 output block $\boldsymbol{x}$ corresponding to the decimation of $\boldsymbol{x}_1, ..., \boldsymbol{x}_4$. Our task is to calculate efficiently $\boldsymbol{X}$, the DCT of $\boldsymbol{x}$, directly from the given DCT's of the original blocks $\boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{X}_3$, and $\boldsymbol{X}_4$. (See Figs ??(b) and ??.)

Motion compensation of compressed video [?], [?] (see also [?]) means predicting each 8×8 spatial domain block $\boldsymbol{x}$ of the current frame by a corresponding reference block $\hat{\boldsymbol{x}}$ from a previous frame [4] and encoding the resulting prediction error block $\boldsymbol{e} = \boldsymbol{x} - \hat{\boldsymbol{x}}$ by using the DCT. The best matching reference block $\hat{\boldsymbol{x}}$ may not be aligned to the original 8×8 blocks of the reference frame (see Figs ??(c) and ??). In general, the reference block may intersect with four neighboring spatial domain blocks, henceforth denoted $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$, and $\boldsymbol{x}_4$, that together form a 16×16 square, where $\boldsymbol{x}_1$ corresponds to northwest, $\boldsymbol{x}_2$ to northeast, $\boldsymbol{x}_3$ to southwest and $\boldsymbol{x}_4$ to southeast.

Our goal here is to compute the DCT $\boldsymbol{X}$ of the current block $\boldsymbol{x} = \hat{\boldsymbol{x}} + \boldsymbol{e}$ from the given DCT $\boldsymbol{E}$ of the prediction error $\boldsymbol{e}$ of the block in the current frame, and the DCT's $\boldsymbol{X}_1, ..., \boldsymbol{X}_4$ of the corresponding previous frame blocks, $\boldsymbol{x}_1, ..., \boldsymbol{x}_4$, respectively. Since $\boldsymbol{X} = \hat{\boldsymbol{X}} + \boldsymbol{E}$, $\hat{\boldsymbol{X}}$ being the DCT of $\hat{\boldsymbol{x}}$, the main problem that remains is that of calculating $\hat{\boldsymbol{X}}$ directly from $\boldsymbol{X}_1, ..., \boldsymbol{X}_4$.

Let the intersection of the reference block $\hat{\boldsymbol{x}}$ with $\boldsymbol{x}_1$ form a $h \times w$ rectangle (i.e., h rows and w columns), where $1 \leq h \leq 8$ and $1 \leq w \leq 8$. This means that the intersections of $\hat{\boldsymbol{x}}$ with $\boldsymbol{x}_2$,$\boldsymbol{x}_3$, and $\boldsymbol{x}_4$ are rectangles of sizes $h \times (8-w)$, $(8-h) \times w$, and $(8-h) \times (8-w)$, respectively. Following Chang and Messerschmitt [?], it is readily seen that $\hat{\boldsymbol{x}}$ can be expressed as a superposition of appropriate windowed and shifted versions of $\boldsymbol{x}_1, ...\boldsymbol{x}_4$, i.e.,

$$\hat{\boldsymbol{x}} = \sum_{i=1}^{4} c_{i1} \boldsymbol{x}_i c_{i2}, \tag{6}$$

where c_{ij}, $i = 1, ..., 4$, $j = 1, 2$, are sparse 8×8 matrices of zeroes and ones that perform window and shift operations accordingly. The basic idea behind the the work of Chang and Messerschmitt [?] is to use the distributive property of matrix multiplication w.r.t. the

[3] This simple averaging corresponds to a commonly used antialiasing filter. Other specific filters can be considered as well using the same methods that we present below. It is not guaranteed, however, that every reasonable anti-aliasing filter implemented by these methods would give a smaller complexity in the DCT domain than in the spatial domain.

[4] In some of the frames (B-frames) blocks are estimated from both past and future reference blocks. For the sake of simplicity, we shall assume here that only the past is used (P-frames). The extension to B-frames is straightforward.

DCT. Specifically, since $S^tS = I$, eq. (??) may be rewritten as

$$\hat{\boldsymbol{x}} = \sum_{i=1}^{4} c_{i1} S^t S \boldsymbol{x}_i S^t S c_{i2}. \tag{7}$$

Next, by pre-multiplying both sides of (??) by S, and post-multiplying by S^t, one obtains

$$\hat{\boldsymbol{X}} = \sum_{i=1}^{4} C_{i1} \boldsymbol{X}_i C_{i2}. \tag{8}$$

where C_{ij} is the DCT of c_{ij}. Chang and Messerscmitt [?] proposed to precompute the fixed matrices C_{ij} for every possible combination of w and h, and to compute $\hat{\boldsymbol{X}}$ directly in the DCT domain using eq. (??). Although most of the matrices C_{ij} are not sparse, computations can still be saved on the basis of typical sparseness of $\{\boldsymbol{X}_i\}$, and due to the fact the reference block might be aligned in one direction (either $w = 8$ or $h = 8$), which means that the right-hand side of eq. (??) contains two terms only, or in both directions ($w = h = 8$), in which case $\hat{\boldsymbol{x}} = \boldsymbol{x}_1$ and hence no computations at all are needed. In Section 4, we develop an algorithm that is efficient even when the sparseness or alignment constraints are not satisfied.

3 Down-sampling

3.1 The Basic Idea

For the sake of simplicity, let us confine attention first to the one dimensional case and a down-sampling factor of 2. The two dimensional case will be a repeated application for every row and then for every column of each block. In this case, we are given two 8-dimensional vectors $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ of DCT coefficients corresponding to adjacent time domain vectors of length 8, $\boldsymbol{x}_1 = S^{-1}\boldsymbol{X}_1$ and $\boldsymbol{x}_2 = S^{-1}\boldsymbol{X}_2$, and we wish to calculate $\boldsymbol{X}$, the DCT of the 8-dimensional vector $\boldsymbol{x}$, whose each component is the average of the two appropriate adjacent components in $\boldsymbol{x}_1$ or $\boldsymbol{x}_2$.

It is convenient to describe the decimation operation in a matrix form as follows.

$$\boldsymbol{x} = \frac{1}{2}(Q_1\boldsymbol{x}_1 + Q_2\boldsymbol{x}_2) \tag{9}$$

where

$$Q_1 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and

$$Q_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Therefore,

$$\boldsymbol{X} = \frac{1}{2}(SQ_1S^{-1}\boldsymbol{X}_1 + SQ_2S^{-1}\boldsymbol{X}_2). \tag{10}$$

We shall now focus on efficient factorizations of the matrices $U_1 = SQ_1S^{-1}$ and $U_2 = SQ_2S^{-1}$. To this end, we shall use a factorization of S that corresponds to the fastest existing algorithm for 8-point DCT due to Arai, Agui, and Nakajima [**?**] (see also [**?**]), which is based on the Winograd algorithm. According to this factorization S is represented as follows.

$$S = DPB_1B_2MA_1A_2A_3 \tag{11}$$

where D is a diagonal matrix given by

$$D = \text{diag}\{0.3536, 0.2549, 0.2706, 0.3007, 0.3536, 0.4500, 0.6533, 1.2814\}, \tag{12}$$

P is a permutation matrix given by

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

and the remaining matrices are defined as follows:

$$B_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix}$$

$$B_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.7071 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.9239 & 0 & -0.3827 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.7071 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.3827 & 0 & 0.9239 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_1 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

Thus, for $i = 1, 2$ we have

$$U_i = SQ_iS^{-1} = DPB_1B_2MA_1A_2A_3Q_iA_3^{-1}A_2^{-1}A_1^{-1}M^{-1}B_2^{-1}B_1^{-1}P^{-1}D^{-1} \tag{13}$$

The proposed decimation algorithm is based on the observation that the products

$$F_i = MA_1A_2A_3Q_iA_3^{-1}A_2^{-1}A_1^{-1}M^{-1} \qquad i = 1, 2 \tag{14}$$

are fairly sparse matrices, and most of the corresponding elements are the same, sometimes with a different sign. This means that their sum $F_+ = F_1 + F_2$ and their difference $F_- = F_1 - F_2$ are even sparser. These matrices are given as follows.

$$F_+ = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.8285 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.7071 & 0 & -1.7071 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.2929 & 0 & 0.7071 & 0 \\ 0 & 0 & 0 & 0 & -0.3827 & 0 & 0.9239 & 0 \end{pmatrix}$$

$$F_- = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.7653 & 0 & 1.8477 & 0 \\ 0 & 0 & 0 & 0 & -0.7653 & 0 & 1.8477 & 0 \\ 0.5412 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.7071 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1.3066 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.5000 & 0 & 0.7071 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Finally, we use the following simple relation:

$$\begin{aligned} \boldsymbol{X} &= \frac{1}{2}(U_1\boldsymbol{X}_1 + U_2\boldsymbol{X}_2) \\ &= \frac{1}{4}[(U_1 + U_2)(\boldsymbol{X}_1 + \boldsymbol{X}_2) + (U_1 - U_2)(\boldsymbol{X}_1 - \boldsymbol{X}_2)] \\ &= \frac{1}{4}DPB_1B_2[F_+B_2^{-1}B_1^{-1}P^{-1}D^{-1}(\boldsymbol{X}_1 + \boldsymbol{X}_2) + \\ &\quad F_-B_2^{-1}B_1^{-1}P^{-1}D^{-1}(\boldsymbol{X}_1 - \boldsymbol{X}_2)] \end{aligned} \tag{15}$$

Let us count the number of basic arithmetic operations on a typical processor (e.g., PA-RISC [?]) that are needed to implement the right-most side of (??) and compare it to the spatial domain approach. As explained earlier, here the term "operation" corresponds to the elementary arithmetic computation of such a processor which is either either "shift", "add", or "shift and add" (SH1ADD, SH2ADD, and SH3ADD). For example, the computation $z = 1.375x + 1.125y$ is implemented as follows: First, we compute $u = x + 0.5x$ (SH1ADD), then $v = x + 0.25u$ (SH2ADD), afterwards $w = v + y$ (ADD), and finally, $z = w + 0.125y$ (SH3ADD). Thus, overall 4 basic operations are needed in this example. The non-trivial multiplications of both the standard spatial domain approach and the algorithm proposed herein were efficiently converted to "operations" by using this type of implementation method.

When counting the operations, we will also use the fact that multiplications by D and D^{-1} can be ignored because these can be absorbed in the MPEG quantizer and dequantizer, respectively. The matrices P and P^{-1} cause only changes in the order of the components so they can be ignored as well. Thus we are left with the following (in parentheses, we detail also the number of additions/subtractions and the number of nontrivial multiplications):

Creating $\boldsymbol{X}_1 + \boldsymbol{X}_2$ and $\boldsymbol{X}_1 - \boldsymbol{X}_2$: 16 operations (16 additions).
Two multiplications by B_1^{-1}: 16 operations (8 additions).
Two multiplications by B_2^{-1}: 16 operations (8 additions).
Multiplication by F_+: 23 operations (5 multiplications + 5 additions).
Multiplication by F_-: 28 operations (6 multiplications + 4 additions).
Adding the products: 8 operations (8 additions).
Multiplication by B_2: 4 operations (4 additions).
Multiplication by B_1: 4 operations (4 additions).
Total: 115 operations (11 multiplications + 57 additions).

In the spatial domain approach, on the other hand, if we use the above mentioned Winograd DCT algorithm [?] (which is the fasted known DCT algorithm to date), we have the following:

Two IDCT's: 114 operations (10 multiplications + 58 additions).

Down-sampling by two in spatial domain: 8 operations (8 additions).

DCT: 42 operations (5 multiplications + 29 additions).

Total: 164 operations (15 multiplications + 95 additions).

It turns out, as can be seen, that the proposed approach saves about 30% of the operations in the one-dimensional case. We shall see later on, in the two-dimensional case, that by using the same ideas, we obtain even greater reductions in complexity.

As a byproduct of the proposed approach, it should be noted that arithmetic precision is gained. Since in the direct approach, we actually multiply by each one of the matrices on right hand side of (**??**) one at a time, then roundoff errors, associated with finite word length representations of the elements of these matrices, accumulate in each step. On the other hand, in the proposed approach, we can precompute F_i once and for all to any desired degree of precision, and then round off each element of these matrices to the allowed precision. The latter has, of course, better precision. More details will be provided in subsection 3.3.

3.2 The Two-dimensional Case

Let us now return to the two dimensional case. A 2D-DCT is just 1D-DCT applied to every column and every row of the spatial domain block. Therefore, the very same ideas can be applied to the two dimensional case as well. In this subsection we describe in detail the computation schemes for down-sampling by a factor of 2. Similar ideas can be applied to other down-sampling factors. (See [**?**] for more detail on factors of 3 and 4.)

Similarly as in the one dimensional case, we have in the spatial domain

$$\boldsymbol{x} = \frac{1}{4}(Q_1\boldsymbol{x}_1Q_1^t + Q_1\boldsymbol{x}_2Q_2^t + Q_2\boldsymbol{x}_3Q_1^t + Q_2\boldsymbol{x}_4Q_2^t) \tag{16}$$

and therefore, in the frequency domain,

$$\boldsymbol{X} = \frac{1}{4}(U_1\boldsymbol{X}_1U_1^t + U_1\boldsymbol{X}_2U_2^t + U_2\boldsymbol{X}_3U_1^t + U_2\boldsymbol{X}_4U_2^t) \tag{17}$$

Again, we would like to express the right-hand side of (**??**) in terms of $U_+ = U_1 + U_2 = DPB_1B_2F_+B_2^{-1}B_1^{-1}P^{-1}D^{-1}$ and $U_- = U_1 + U_2 = DPB_1B_2F_-B_2^{-1}B_1^{-1}P^{-1}D^{-1}$. To this end, let us define

$$\boldsymbol{X}_{+++} = \boldsymbol{X}_1 + \boldsymbol{X}_2 + \boldsymbol{X}_3 + \boldsymbol{X}_4, \tag{18}$$

$$\boldsymbol{X}_{+--} = \boldsymbol{X}_1 + \boldsymbol{X}_2 - \boldsymbol{X}_3 - \boldsymbol{X}_4, \tag{19}$$

$$\boldsymbol{X}_{-+-} = \boldsymbol{X}_1 - \boldsymbol{X}_2 + \boldsymbol{X}_3 - \boldsymbol{X}_4, \tag{20}$$

and

$$\boldsymbol{X}_{--+} = \boldsymbol{X}_1 - \boldsymbol{X}_2 - \boldsymbol{X}_3 + \boldsymbol{X}_4. \tag{21}$$

Note that to create all these linear combinations, we need only 8 (and not 12) additions/subtractions per frequency component: We first compute $\boldsymbol{X}_1 \pm \boldsymbol{X}_2$ and $\boldsymbol{X}_3 \pm \boldsymbol{X}_4$ and then $(\boldsymbol{X}_1 + \boldsymbol{X}_2) \pm (\boldsymbol{X}_3 + \boldsymbol{X}_4)$ and $(\boldsymbol{X}_1 - \boldsymbol{X}_2) \pm (\boldsymbol{X}_3 - \boldsymbol{X}_4)$. Now, eq. (**??**) can be rewritten as

$$\begin{aligned}
\boldsymbol{X} &= \frac{1}{16}(U_+ \boldsymbol{X}_{+++} U_+^t + U_- \boldsymbol{X}_{+--} U_+^t + U_+ \boldsymbol{X}_{-+-} U_-^t + U_- \boldsymbol{X}_{--+} U_-^t) \\
&= \frac{1}{16} DPB_1B_2 \cdot \\
&\quad [(F_+ B_2^{-1} B_1^{-1} P^{-1} D^{-1} \boldsymbol{X}_{+++} + F_- B_2^{-1} B_1^{-1} P^{-1} D^{-1} \boldsymbol{X}_{+--}) D^{-t} P^{-t} B_1^{-t} B_2^{-t} F_+^t + \\
&\quad (F_+ B_2^{-1} B_1^{-1} P^{-1} D^{-1} \boldsymbol{X}_{-+-} + F_- B_2^{-1} B_1^{-1} P^{-1} D^{-1} \boldsymbol{X}_{--+}) D^{-t} P^{-t} B_1^{-t} B_2^{-t} F_-^t] \cdot \\
&\quad B_2^t B_1^t P^t D^t.
\end{aligned} \tag{22}$$

If we count the the number of operations associated with the implementation of the rightmost side of eq. (**??**) (similarly as in the previous section), we find that the total is 2824. The traditional approach, on the other hand, requires 4512 operations. This means that 37.4% of the operations are saved. Similar down-sampling algorithms for factors of 3 and 4 [**?**] yield reductions of 39% and 50%, respectively, compared to the spatial domain counterparts.

Additional savings in computations can be made by taking advantage of the fact that in typical images most of the DCT blocks $\boldsymbol{X}_i$ have only a few nonzero coefficients, normally, the low frequency coefficients. A reasonable possibility might be to use a mechanism that operates in two steps. In the first step, DCT blocks are classified as being *lowpass* or *non-lowpass*, where the former is defined as a block where, say, only the upper left 4×4 sub-block is nonzero. The second step uses either the computation scheme described above for non-lowpass blocks, or a faster scheme that utilizes the lowpass assumption for the precomputation of the above matrix multiplications. It turns out that if $\boldsymbol{X}_1, ..., \boldsymbol{X}_4$ are all lowpass blocks, then the reduction in computations is about 80% in the case of down-sampling by a factor of 2. Of course, these figures correspond to the case where the competing spatial domain approach is 'hard-wired' in the sense that the DCT and IDCT operations are not allowed to take advantage of the sparseness of the DCT-domain data.

In the same manner, one may consider reduced versions of the fast DCT/IDCT algorithm that take into account sparseness as well.

3.3 Arithmetic Precision

As explained in the last paragraph of Section 3.1, the proposed computation scheme provides better arithmetic accuracy than the standard approach. To demonstrate this fact we have tested both schemes for the case of down-sampling by a factor of 2, where each element in the fixed matrices of eq. (??) is represented by 8 bits.

In the first experiment, we have the chosen the elements of $\boldsymbol{x}_1, ..., \boldsymbol{x}_4$ as statistically independent random integers uniformly distributed in the set $\{0, 1, ..., 255\}$. We first computed $\boldsymbol{x}$ (and then $\boldsymbol{X}$) directly from $\boldsymbol{x}_1, ..., \boldsymbol{x}_4$ for reference. We then computed the DCT's $\boldsymbol{X}_1, ..., \boldsymbol{X}_4$ where all DCT coefficients are quantized and then dequantized according to a given quantization matrix Δ. From $\boldsymbol{X}_1, ..., \boldsymbol{X}_4$, we have computed $\boldsymbol{X}$ using both the standard approach and the proposed approach, and compared to the reference version, where the precision in each approach was measured in terms of the sum of squares of errors (MSE) in the DCT domain (and hence also in the spatial domain). For the case where Δ was an all-one matrix, the MSE of the proposed approach was about 3dB better than that of the standard approach. For the case where Δ was the recommended quantization matrix of JPEG for luminance [?, p. 37], the proposed approach outperformed the standard approach by 1.2dB. These results are reasonable because when the step sizes of the quantizer increase, quantization errors associated with the DCT coefficients tend to dominate roundoff errors associated with inaccurate computations.

The second experiment was similar but that test data that was a real image ("Lenna") rather than random data. Now, for the case where Δ was the all-one matrix, the standard approach yielded SNR of 46.08dB while the proposed approach gave 49.24dB, which is again a 3dB improvement. For the case where Δ was the JPEG default quantizer, the figures were 36.63dB and 36.84dB, respectively. Here the degree of improvement is less than in the case of random data because most of the DCT coefficients are rounded to zero in both techniques.

4 Inverse Motion Compensation

We now turn to the problem of inverse motion compensation described earlier in Section 2. This problem as well as its proposed solution are completely independent of the issue of down-sampling. However, it is definitely possible to combine the down-sampling algorithm described above and the inverse motion compensation algorithm described below into a single function which provides both.

4.1 Mathematical Derivation

We mentioned earlier that eq. (**??**), for calculating the reference block $\hat{\boldsymbol{X}}$, can be made efficient if the data are sparse and or if the motion vectors are aligned at least in one direction. We now demonstrate that the computation of $\hat{\boldsymbol{X}}$ can be done even more efficiently by utilizing two main facts. First, we observe that some of the matrices c_{ij} are equal to each other for every given w and h. Specifically,

$$c_{11} = c_{21} = U_h \stackrel{\Delta}{=} \begin{pmatrix} 0 & I_h \\ 0 & 0 \end{pmatrix}$$

$$c_{12} = c_{32} = L_w \stackrel{\Delta}{=} \begin{pmatrix} 0 & 0 \\ I_w & 0 \end{pmatrix}$$

where I_h and I_w are identity matrices of dimension $h \times h$ and $w \times w$, respectively. Similarly,

$$c_{31} = c_{41} = L_{8-h},$$

and

$$c_{22} = c_{42} = U_{8-w}.$$

The second observation that helps in saving computations is that rather than fully precomputing C_{ij}, it might be more efficient to leave these matrices factorized into relatively sparse matrices. In particular, we shall use the factorization of S that was described in subsection 3.1.

The best way we have found to use the two observations mentioned above is the following: First, we precompute the fixed matrices

$$J_i \stackrel{\Delta}{=} U_i(MA_1A_2A_3)^t, \qquad i = 1, 2, ..., 8$$

and

$$K_i \stackrel{\Delta}{=} L_i(MA_1A_2A_3)^t, \qquad i = 1, 2, ..., 8$$

These matrices are very structured and therefore, pre-multiplication by K_i or J_i can be implemented very efficiently as we shall demonstrate shortly. Next, we compute $\hat{\boldsymbol{X}}$ by using the expression

$$\begin{aligned}\hat{\boldsymbol{X}} &= S[J_h B_2^t B_1^t P^t D(\boldsymbol{X}_1 DPB_1B_2J_w^t + \boldsymbol{X}_2 DPB_1B_2K_{8-w}^t) + \\ &\quad K_{8-h}B_2^tB_1^tP^tD(\boldsymbol{X}_3DPB_1B_2J_w^t + \boldsymbol{X}_4DPB_1B_2K_{8-w}^t)]S^t \end{aligned} \tag{23}$$

which can easily be obtained from eqs. (**??**) and (**??**), or by its dual form

$$\begin{aligned}\hat{\boldsymbol{X}} &= S[(J_h B_2^t B_1^t P^t D\boldsymbol{X}_1 + K_{8-h}B_2^tB_1^tP^tD\boldsymbol{X}_3)DPB_1B_2J_w^t + \\ &\quad (J_h B_2^t B_1^t P^t D\boldsymbol{X}_2 + K_{8-h}B_2^tB_1^tP^tD\boldsymbol{X}_4)DPB_1B_2K_{8-w}^t]S^t, \end{aligned} \tag{24}$$

depending on which one of these expressions requires less computations for the given w and h.

4.2 Implementation and Computational Complexity

We now demonstrate how to implement fast multiplication by J_i and K_i, which is the bottle neck of the computation load. As an example, we shall examine J_6. The other matrices are handled in a similar fashion. The matrix J_6 is the following:

$$J_6 = \begin{pmatrix} 1 & -1 & -a & 0 & b & a & c & 0 \\ 1 & 1 & -a & -1 & b & 0 & c & 0 \\ 1 & 1 & -a & -1 & -b & 0 & -c & 0 \\ 1 & -1 & -a & 0 & -b & -a & -c & 0 \\ 1 & -1 & a & 0 & c & -a & -b & 0 \\ 1 & 1 & a & 1 & c & 0 & -b & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where $a = 0.7071$, $b = 0.9239$, and $c = 0.3827$. To compute $u = J_6 v$, where $u = (u_1, ..., u_8)^t$ and $v = (v_1, ..., v_8)^t$, we calculate according to the following steps:

$$y_1 = v_1 + v_2 \tag{25}$$

$$y_2 = v_1 - v_2 \tag{26}$$

$$y_3 = av_3 \tag{27}$$

$$y_4 = av_6 \tag{28}$$

$$y_5 = y_1 - y_3 \tag{29}$$

$$y_6 = y_5 - v_4 \tag{30}$$

$$y_7 = y_3 - y_4 \tag{31}$$

$$y_8 = y_3 + y_4 \tag{32}$$

$$y_9 = (b+c)(v_5 + v_7) \tag{33}$$

$$y_{10} = cv_5 \tag{34}$$

$$y_{11} = bv_7 \tag{35}$$

$$y_{12} = y_9 - y_{10} - y_{11} \tag{36}$$

$$y_{13} = y_{10} - y_{11} \tag{37}$$

$$u_1 = y_2 - y_7 + y_{12} \tag{38}$$

$$u_2 = y_6 + y_{12} \tag{39}$$

$$u_3 = y_6 - y_{12} \tag{40}$$

$$u_4 = y_2 - y_8 - y_{12} \tag{41}$$

$$u_5 = y_2 + y_7 + y_{13} \tag{42}$$

$$u_6 = y_1 + y_3 + v_4 + y_{13} - v_8 \tag{43}$$

$$u_7 = 0 \tag{44}$$

$$u_8 = 0. \tag{45}$$

This implementation requires 5 multiplications and 22 additions, which is equivalent to 43 elementary processor operations in our above model. By developing similar implementation schemes of matrix multiplication for all matrices $J_1, ..., J_8$, we find that the numbers $\{N_i\}$ of operations required to multiply by $\{J_i\}$, $1 \leq i \leq 8$, are given by $N_1 = 18$, $N_2 = 24$, $N_3 = 38$, $N_4 = 39$, $N_5 = 40$, $N_6 = 43$, $N_7 = 44$, and $N_8 = 46$. Since the matrix K_i has a structure similar to that of J_i for every $1 \leq i \leq 8$, multiplication by K_i costs also N_i operations. Thus, for a general position reference block (i.e., $1 \leq w \leq 7$, $1 \leq h \leq 7$), we have the following:

1. Six multiplications by B_1 or B_1^t: $6 \times 32 = 192$ operations.

2. Six multiplications by B_2 or B_2^t: $6 \times 32 = 192$ operations.

3. Two multiplications by J_w and K_{8-w}, and one by J_h and K_{8-h}, or vice versa: $8 \cdot (N_h + N_{8-h} + N_w + N_{8-w} + \min\{N_h + N_{8-h}, N_w + N_{8-w}\}$ operations.

4. One 2D-DCT (using eq. (**??**)): $42 \times 16 = 672$ operations.

Total: $1056 + 8 \cdot (N_h + N_{8-h} + N_w + N_{8-w} + \min\{N_h + N_{8-h}, N_w + N_{8-w}\})$ operations.

Note that we have not counted additions of the products in eqs. (??) and (??) because the different summands are nonzero on disjoint subsets of indices of matrix elements. When the reference block is aligned in the vertical direction only, i.e., $h = 8$ and $1 \leq w \leq 7$, then $K_{8-h} = K_0 = L_0(MA_1A_2A_3)^t = 0$, and therefore eqs. (??) and (??) contain two terms only. Furthermore, since $J_h = J_8 = U_8(MA_1A_2A_3)^t = (MA_1A_2A_3)^t$, eq. (??) degenerates to

$$\hat{\boldsymbol{X}} = (\boldsymbol{X}_1 DPB_1B_2J_w^t + \boldsymbol{X}_2 DPB_1B_2K_{8-w}^t)S^t \tag{46}$$

which requires the following steps:

1. Two multiplications by B_1: $2 \times 32 = 64$ operations.
2. Two multiplications by B_2: $2 \times 32 = 64$ operations.
3. One multiplication by J_w and one by K_{8-w}: $8(N_w + N_{8-w})$ operations.
4. One multiplication by S^t: $8 \times 42 = 336$ operations.

Total: $464 + 8(N_w + N_{8-w})$ operations.

Similarly, for the horizontally aligned case, where $w = 8$ and $1 \leq h \leq 7$, the number of computations is $464+8(N_h+N_{8-h})$. As mentioned earlier, when $w = h = 8$ no computations are required at all since $\hat{\boldsymbol{X}} = \boldsymbol{X}_1$ and hence already given.

By using the above expressions, we find that the number of computations for the worst case values of h and w is 2928, and the average number, assuming a uniform distribution on the pairs $\{(w, h) : 1 \leq w \leq 8, 1 \leq h \leq 8\}$, is 2300.5. On the other hand, the brute-force approach of performing IDCT to $\boldsymbol{X}_1, ..., \boldsymbol{X}_4$, cutting the appropriate reference block in the spatial domain, and transforming it back, requires a total of 4320 operations. This means that the reduction in computational complexity, in comparison to the brute-force method, is 32% for the worst case and 46.8% for the average.

So far we have not assumed that the input DCT matrices are sparse. Typically, a considerable percentage of the DCT blocks have only a few nonzero elements, normally, those corresponding to low spatial frequencies in both directions. For simplicity, we shall refer to a DCT block as *sparse* if only the top left 4×4 quadrant (corresponding to low frequencies) is nonzero.

We have redesigned the implementation of multiplication by J_i and K_i, $1 \leq i \leq 8$, when $\boldsymbol{X}_1, ..., \boldsymbol{X}_4$ are assumed sparse in the above sense, and found that the number of

computations is $672 + 8 \cdot (N'_w + N'_{8-w} + N'_h + N'_{8-h})$ for $1 \leq w \leq 7$ and $1 \leq h \leq 7$, $336 + 4 \cdot (N'_w + N'_{8-w})$ for $h = 8$ and $1 \leq w \leq 7$, $336 + 4 \cdot (N'_h + N'_{8-h})$ for $w = 8$ and $1 \leq h \leq 7$, and zero when $w = h = 8$, where $N'_1 = 15$, $N'_2 = 20$, $N'_3 = 26$, $N'_4 = 33$, $N'_5 = 36$, $N'_6 = 40$, $N'_7 = 41$, and, $N'_8 = 42$. This means that the are 1728 computations in the worst case and 1397.2 on the average, corresponding to reductions of 60% and 68%, respectively, compared to the brute force approach.

For comparison with earlier results, Chang and Messerschmitt [?] have shown computation savings only if the DCT matrices are sparse enough and if a large percentage of the reference blocks are aligned at least in one direction. Specifically, these authors introduced three parameters: the reciprocal of the fraction of nonzero coefficients β, the fraction α_1 of reference blocks aligned in one direction, and the fraction α_2 of completely unaligned reference blocks.

Let us consider first the worst case situation in terms of block alignment, i.e., $\alpha_1 = 0$ and $\alpha_2 = 1$. Our above definition of sparseness corresponds to $\beta = 4$. Chang et al. provide exact formulas for the number multiplications and additions associated with their approach in terms of α_1, α_2, β and the block size N ($N = 8$ in MPEG). According to these formulas, their approach require in this case 16 multiplications per pixel and 19 additions per pixel. To compare with typical processor operations, let us assume that on the average every multiplication requires up to 4 SHIFTs and 3 ADDs and that SHIFTs and ADDs can be done simultaneously. This means that a conservative estimate of the total number of operations per block is $(16 \times 3 + 19) \times 64 = 4288$, which is much larger than 1728 operations (see above) in the proposed approach under the same circumstances.

As another point of comparison, note that a uniform distribution over w and h in our case corresponds to $\alpha_1 = 14/64 = 0.219$ and $\alpha_2 = 49/64 = 0.766$, which is more pessimistic than the upper curve in Fig. 5 of [?], where $\alpha_1 = 0.2$ and $\alpha_2 = 0.1$. Nevertheless, for $\beta = 1$ we are able to speedup the computations by a factor of $4320/2300.5 = 1.87$ compared to 0.6 in [?], and for $\beta = 4$ our speedup is $4320/1397.2 = 3.13$ compared to approximately 2.0 in [?]. Furthermore, if we assume $\alpha_1 = 0.2$ and $\alpha_2 = 0.1$ as in [?] we obtain speedup factors of 9.06 for $\beta = 1$ and about 15 for $\beta = 4$, which means an improvement by an order of magnitude compared to [?].