Vector implementation of color-image wavelet transform

Richard Crandall and Jason Klivington
Advanced Computation Group, Apple Computer

Abstract. There are various approaches to wavelet processing of color images, and machine architecture dictates in large measure which algorithm is optimal. We describe herein a Velocity-Engine (G4) implementation in which pixels are processed as four-dimensional (RGBA) vector entities. In this mode the vector machinery performs the (Daubechies D4) wavelet algebra in only three vector operations per pixel. We also implemented a more standard, channel-correlation scenario, with YUV-decomposed RGB images (with UV subsampling) and a biorthogonal (Burt 5/7) wavelet transform applied thrice. A key to these fast vector implementations is the adoption of certain rational approximations—we call "shift-rational" forms—to the true wavelet coefficients, allowing for efficient Velocity Engine arithmetic. Other Velocity Engine enhancements include very fast subsampling for the UV channels, via vector-average instructions. Timing experiments show a Velocity Engine speedup of 5x or more over corresponding scalar (G3) implementation in the RGBA approach. For the YUV approach, the speedup is likewise impressive, with a complete inverse-wavelet-YUV image reconstruction on a 320-by-240 full color image taking less than $\frac{1}{200}$ second on 300 MHz. G4.

25 October 1999

1. Wavelet choices

For the Velocity Engine timings we eventually report herein, the wavelets used were:

Daubechies compact wavelet (D4), in RGBA mode
Biorthogonal wavelet (Burt 5/7), in YUV mode

The former wavelet choice was effected on all four RGBA color channels at once (where A is "alpha," or opacity channel), while the latter wavelet was used in YUV-decomposition mode whereby the three RGB channels were transformed as in Section 2. Thus YUV mode is the more standard approach, and is likely to allow superior results in actual applications (such as image compression) because, of course, we expect interchannel correlations which are exploited by YUV transformation—a great deal of signal energy ends up in the Y channel.

The D4 wavelet coefficients are taken to be:

$$\{h_0, h_1, h_2, h_3\} = \{\frac{1+\sqrt{3}}{4}, \frac{3+\sqrt{3}}{4}, \frac{3-\sqrt{3}}{4}, \frac{1-\sqrt{3}}{4}\},$$

which are often manifest in software as floating-point values (they are all irrational), yet we shall be able to give such coefficients a satisfactory rational form suitable for Velocity Engine work. The D4 coefficients satisfy normalization:

$$h_0^2 + h_1^2 + h_2^2 + h_3^2 = 2,$$

and moment conditions

$$h_0 - h_1 + h_2 - h_3 = 0,$$

$$0h_0 - 1h_1 + 2h_2 - 3h_3 = 0,$$

which conditions admit of clear interpretation (e.g. the moment conditions tend to suppress linear and parabolic signal behavior, respectively). The actual D4 wavelet transform is built from matrix operators, each having an upper "scalar" half and a lower "difference" half:

$$W_n = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 & 0 & \cdots & & \\ 0 & 0 & h_0 & h_1 & h_2 & h_3 & 0 & \cdots \\ \vdots & & & & & & & \vdots \\ h_3 & -h_2 & h_1 & -h_0 & 0 & \cdots & & \\ 0 & 0 & h_3 & -h_2 & h_1 & -h_0 & 0 & \cdots \\ \vdots & & & & & & & \vdots \end{pmatrix}$$

where the matrix is n-by-n and one normally applies this matrix at successive resolutions, to an original length-N signal column, so that an overall wavelet transform to "depth k" can be symbolized:

$$W = W_{n/2^k} ... W_{N/4} W_{N/2} W_N.$$

It is understood that in this chain a matrix W_m applies only to the upper m entries of the current signal column, and that the boundary conditions at the edges of the various

matrices need be further defined (one way to proceed is to effect wrap-around and superposition, but there are other ways to handle the edges). The convenient property of the W matrices is that they are unitary: the inverse transform—up to correct normalization—is given simply by the matrix transpose.

The example we use of biorthogonal wavelets is the Burt 5/7 device, which involves coefficients

$$\{H_0, H_1, H_2\} = \{\frac{3}{5}\sqrt{2}, \frac{1}{4}\sqrt{2}, -\frac{1}{20}\sqrt{2}\},$$

$$\{G_0, G_1, G_2, G_3\} = \{-\frac{170}{280}\sqrt{2}, \frac{73}{280}\sqrt{2}, \frac{15}{280}\sqrt{2}, -\frac{3}{280}\sqrt{2}\}.$$

This wavelet is known to have improved distortion properties on images (for one thing it is entirely symmetrical in filter form, unlike the D4 wavelet). The matrix forms for such biorthogonal wavelets are reminiscent of the D4 form given above, in the sense that the Burt 5/7 "scalar" half-matrix has within a typical row:

$$...H_2, H_1, H_0, H_1, H_2...$$

while the"difference" half has rows such as:

$$...G_3, G_2, G_1, G_0, G_1, G_2, G_3...$$

with, again, proper boundary conditions assumed for the (finite) matrices. For this kind of biorthogonal transform the inverse is given—again, up to normalization—by the transpose of a matrix having rows of the types:

$$... - H_2, H_1, -H_0, H_1, -H_2...$$

$$...G_3, -G_2, G_1, -G_0, G_1, -G_2, G_3...$$

with, as expected, some detailed boundary conditions in force at matrix edges. For completeness we give here the explicit forward and inverse biorthogonal-wavelet matrices respectively, for length-8 signals (to be interpreted as 8-element column vectors):

$$\begin{pmatrix}
H_0 & 2\,H_1 & 2\,H_2 & 0 & 0 & 0 & 0 & 0 \\
H_2 & H_1 & H_0 & H_1 & H_2 & 0 & 0 & 0 \\
0 & 0 & H_2 & H_1 & H_0 & H_1 & H_2 & 0 \\
0 & 0 & 0 & 0 & H_2 & H_1 & H_0 + H_2 & H_1 \\
G_1 & G_0 + G_2 & G_1 + G_3 & G_2 & G_3 & 0 & 0 & 0 \\
G_3 & G_2 & G_1 & G_0 & G_1 & G_2 & G_3 & 0 \\
0 & 0 & G_3 & G_2 & G_1 & G_0 & G_1 + G_3 & G_2 \\
0 & 0 & 0 & 0 & 2\,G_3 & 2\,G_2 & 2\,G_1 & G_0
\end{pmatrix}$$

$$\begin{pmatrix}
-G_0 & -2\,G_2 & 0 & 0 & 2\,H_1 & 0 & 0 & 0 \\
G_1 & G_1 + G_3 & G_3 & 0 & -H_0 - H_2 & -H_2 & 0 & 0 \\
-G_2 & -G_0 & -G_2 & 0 & H_1 & H_1 & 0 & 0 \\
G_3 & G_1 & G_1 & G_3 & -H_2 & -H_0 & -H_2 & 0 \\
0 & -G_2 & -G_0 & -G_2 & 0 & H_1 & H_1 & 0 \\
0 & G_3 & G_1 & G_1 + G_3 & 0 & -H_2 & -H_0 & -H_2 \\
0 & 0 & -G_2 & -G_0 - G_2 & 0 & 0 & H_1 & H_1 \\
0 & 0 & 2\,G_3 & 2\,G_1 & 0 & 0 & -2\,H_2 & -H_0
\end{pmatrix}$$

For the exact H, G coefficients as given above (with the $\sqrt{2}$ factors) the product of these two 8-by-8 matrices is precisely an identity matrix.

For either style of wavelet, the primary consideration for Velocity Engine machinery is that for pixel values $x_0, x_1, ...$, or perhaps a simple reordering of same, one needs to calculate components of the matrix-vector product, a typical such component being:

$$c_0 x_k + c_1 x_{k+1} + c_2 x_{k+2} + ...,$$

for some wavelet coefficients c_i. In the shift-rational paradigm, what we actually end up calculating is:

$$\frac{c_0 x_k + c_1 x_{k+1} + c_2 x_{k+2} + ... + 2^{b-1}}{2^b}$$

with the offset of 2^{b-1} aiding on average the natural rounding error. As we shall see this algebra can be parallelized in, perhaps surprisingly, more than one way.

2. Shift-rational coefficients

By "shift-rational" for a coefficient c we mean a form:

$$c \approx \frac{n}{2^b},$$

for integer n, so that the implied division is a convenient right-shift by b bits. Happily, we were able to find adequate shift-rational forms for either wavelet style. For the D4 wavelet, the approximate coefficients:

$$\{h_0, h_1, h_2, h_3\} = \{\frac{11}{16}, \frac{19}{16}, \frac{5}{16}, -\frac{3}{16}\}$$

enjoy the properties:

$$h_0^2 + h_1^2 + h_2^2 + h_3^2 = \frac{129}{64} \approx 2.016,$$

and moment conditions

$$h_0 - h_1 + h_2 - h_3 = 0,$$

$$0h_0 - 1h_1 + 2h_2 - 3h_3 = 0,$$

of which, remarkably, the last two are exact. For the Burt 5/7 wavelet, we found coefficients:

$$\{H_0, H_1, H_2\} = \{\frac{216}{256}, \frac{92}{256}, -\frac{18}{256}\},$$

$$\{G_0, G_1, G_2, G_3\} = \{-\frac{220}{256}, \frac{94}{256}, \frac{20}{256}, -\frac{4}{256}\},$$

for which we have the exact relation

$$G_0 + 2G_1 + 2G_2 + 2G_3 = 0.$$

The fortunate aspect of this particular shift-rational representation is that, for the forward matrix B_n or length-n and the (formal, formed as above from $\pm H, \pm G$ terms) backward matrix C it happens that BC is *very* close to the identity. In fact, we calculated the exact error for $n = 8$ in the form:

$$B_8\, C_8 \;=\; I \;-\; \begin{pmatrix} 0 & 0 & 2\delta & 0 & 0 & 0 & 0 & 0 \\ 0 & \delta & 0 & \delta & 0 & 0 & 0 & 0 \\ \delta & 0 & 0 & \delta & 0 & 0 & 0 & 0 \\ 0 & \delta & \delta & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \delta & \delta & 0 \\ 0 & 0 & 0 & 0 & \delta & 0 & 0 & \delta \\ 0 & 0 & 0 & 0 & \delta & 0 & \delta & 0 \\ 0 & 0 & 0 & 0 & 0 & 2\delta & 0 & 0 \end{pmatrix}$$

where

$$\delta = \frac{1}{8192},$$

and similarly impressive near-identity relations for longer lengths n.

Along similar lines we found an accurate shift-rational YUV transformation, in the form of matrix operator:

$$L = \frac{1}{128} \begin{pmatrix} 43 & 42 & 43 \\ -42 & -43 & 85 \\ 85 & -43 & -42 \end{pmatrix}$$

acting on RGB columns. Technically of course this is not the standard YUV transformation, but as is well known this simpler kind of operator does well to exploit the channel correlations. At any rate, our goal was to have a very fast inverse operation, as close as possible to a matrix having only $0, -1, +1$ values. Indeed, the particular matrix yuv above has the fortunate property:

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & 0 \end{pmatrix} L = \begin{pmatrix} 1 & -\frac{1}{128} & \frac{1}{128} \\ 0 & 1 & 0 \\ \frac{1}{128} & -\frac{1}{128} & 1 \end{pmatrix},$$

so that for example the recovered R channel will sustain some small G, B interference; the G channel reconstructs perfectly, and so on.

Because of the high accuracy of these shift-rational approximations, the typical scenario for either YUV-decomposed images or our direct RGBA transform was this: upon inversion, the error due to the shift-rational coefficients is manifest as some small fraction (typically less than 10 per cent) of all pixel values are off by ± 1 or ± 2 out of 255 maximum, and all the rest are exactly reconstructed. Because one expects eventually to use such transforms together with deep quantization—for deep resulting compression—this level of error is, being on the order of 50 db. or more, entirely acceptable—very nearly lossless.

3. Vectorization details

Let us first consider the case of a two-dimensional Daubechies D4 wavelet transform on a four channel (RGBA) image. Because of the interleaved nature of the data, and to best take advantage of the Velocity Engine's performance, we chose to implement a separate algorithm for horizontal and vertical transforms of the image data. Since data is interleaved, and the transform is performed on the separate channels (i.e. only red pixel data is used to calculate the red channel transform), some manipulation is required to get the data into the correct vector form for our calculations.

In the case of the horizontal forward transform, we loop through each row six source pixels at a time. Since each pixel contains four 16-bit elements (padded from 8 bits to accomodate expansion from the transform), this requires three 128-bit Velocity Engine registers, so that each vector contains two complete four-channel pixels. So, starting with the 0-th pixel in a row, we have three input vectors:

$$\mathrm{vIn}_1 = (a_0, r_0, g_0, b_0, a_1, r_1, g_1, b_1)$$

$$\mathrm{vIn}_2 = (a_2, r_2, g_2, b_2, a_3, r_3, g_3, b_3)$$

$$\mathrm{vIn}_3 = (a_4, r_4, g_4, b_4, a_5, r_5, g_5, b_5)$$

Where, for example, a_2 is the alpha component of pixel 2. From the above shift-rational matrix for the D4 transform, we have, for the alpha channel of the 0-th transform element:

$$a_0 = \frac{a_0 h_0 + a_1 h_1 + a_2 h_2 + a_3 h_3 + 2^{b-1}}{2^b}$$

In order to have the data in a form appropriate for the necessary Velocity Engine operations, we use the Velocity Engine's permute operation on vIn_1 and vIn_2 to generate the multiplicand vectors

$$\mathrm{vMul}_1 = (a_0, a_1, r_0, r_1, g_0, g_1, b_0, b_1)$$

$$\mathrm{vMul}_2 = (a_2, a_3, r_2, r_3, g_2, g_3, b_2, b_3).$$

Then, defining two coefficient multiplicand vectors

$$\mathrm{vCoeff}_1 = (h_0, h_1, h_0, h_1, h_0, h_1, h_0, h_1)$$

$$\mathrm{vCoeff}_2 = (h_2, h_3, h_2, h_3, h_2, h_3, h_2, h_3),$$

a "rounding corrector" vector

$$\mathrm{vCorrector} = (2^{b-1}, 2^{b-1}, 2^{b-1}, 2^{b-1}, 2^{b-1}, 2^{b-1}, 2^{b-1}, 2^{b-1}),$$

and a final "shift-rational" vector of four 32-bit elements

$$\mathrm{vShift} = (b, b, b, b),$$

we can, through the use of the `msum` operation, calculate the entire transform components for all four channels RGBA in three instructions. Thus it can be said that the wavelet algebra *per se* takes three vector operations per pixel, to which effort one must of course add the permute, load/store, pack instructions described above. For those unfamiliar with the Velocity Engine instruction set, the `msum` operation takes three input vectors vA, vB, vC, and produces a single result vector vResult. Given

$$\text{vA} = (vA_0, vA_1, vA_2, vA_3, vA_4, vA_5, vA_6, vA_7) \;\; (16 - \text{bit elements})$$

$$\text{vB} = (vB_0, vB_1, vB_2, vB_3, vB_4, vB_5, vB_6, vB_7) \;\; (16 - \text{bit elements})$$

$$\text{vC} = (vC_0, vC_1, vC_2, vC_3) \;\; (32 - \text{bit elements})$$

the operation

$$\text{vResult} = \texttt{msum}(\text{vA},\ \text{vB},\ \text{vC})$$

produces the vector

$$\begin{aligned}\text{vResult} = (&vA_0vB_0 + vA_1vB_1 + vC_0,\\ &vA_2vB_2 + vA_3vB_3 + vC_1,\\ &vA_4vB_4 + vA_5vB_5 + vC_2,\\ &vA_6vB_6 + vA_7vB_7 + vC_3).\end{aligned}$$

If we take vTransform to be our transform result (with four 32-bit elements), then the instructions

$$\text{vTransform} = \texttt{msum}(\text{mul}_1,\ \text{coeff}_1,\ \text{corrector});$$

$$\text{vTransform} = \texttt{msum}(\text{mul}_2,\ \text{coeff}_2,\ \text{vTransform});$$

$$\text{vTransform} = \texttt{sra}(\text{vTransform},\ \text{vShift});$$

will give us the result vector:

$$\begin{aligned}\text{vTransform} = (&\frac{a_0h_0 + a_1h_1 + a_2h_2 + a_3h_3 + 2^{b-1}}{2^b},\\ &\frac{r_0h_0 + r_1h_1 + r_2h_2 + r_3h_3 + 2^{b-1}}{2^b},\\ &\frac{g_0h_0 + g_1h_1 + g_2h_2 + g_3h_3 + 2^{b-1}}{2^b},\\ &\frac{b_0h_0 + b_1h_1 + b_2h_2 + b_3h_3 + 2^{b-1}}{2^b}).\end{aligned}$$

Note, however, that this is a vector of 32-bit elements, which is required capacity for the pre-shift sum of 16-bit products, although the final `sra` (shift right arithmetic) instruction produces elements that will fit within 16 bits. If we perform the same sequence of above

operations, and substitute vIn_2 and vIn_3 for vIn_1 and vIn_2, respectively, then we can create a second transform result vector, vTransform_2, with

$$\text{vTransform}_2 = (\frac{a_2h_2 + a_3h_3 + a_4h_4 + a_5h_5 + 2^{b-1}}{2^b},$$

$$\frac{r_2h_2 + r_3h_3 + r_4h_4 + r_5h_5 + 2^{b-1}}{2^b},$$

$$\frac{g_2h_2 + g_3h_3 + g_4h_4 + g_5h_5 + 2^{b-1}}{2^b},$$

$$\frac{b_2h_2 + b_3h_3 + b_4h_4 + b_5h_5 + 2^{b-1}}{2^b}).$$

Using the Velocity Engine `pack` instruction, we can then generate a single 8-element vector that contains the truncated 16-bit elements of vTransform and vTransform_2, which are in actuality the first two pixels of the transform data, and can be stored with a single store instruction (which is the motivation for the step size of three input vectors).

If the above steps are repeated with the same input vectors, but with the differencing coefficients of the wavelet:

$$\text{vCoeff}_1 = (h_3, -h_2, h_3, -h_2, h_3, -h_2, h_3, -h_2)$$

$$\text{vCoeff}_2 = (h_1, -h_0, h_1, -h_0, h_1, -h_0, h_1, -h_0),$$

we end up with vTransform and vTransform_2, which, given a data stream of length n four-channel elements, correspond to element $\frac{n}{2}$ and $\frac{n}{2} + 1$ of the four-channel transform, respectively.

In the case of the forward, vertical transform, the strategy is similar, but the required permutations on the input data are different. For example, given input rows 0-3, we have input vectors:

$$\text{vIn}_1 = (A_{00}, R_{00}, G_{00}, B_{00}, A_{10}, R_{10}, G_{10}, B_{10})$$

$$\text{vIn}_2 = (A_{01}, R_{01}, G_{01}, B_{01}, A_{11}, R_{11}, G_{11}, B_{11})$$

$$\text{vIn}_3 = (A_{02}, R_{02}, G_{02}, B_{02}, A_{12}, R_{12}, G_{12}, B_{12})$$

$$\text{vIn}_4 = (A_{03}, R_{03}, G_{03}, B_{03}, A_{13}, R_{13}, G_{13}, B_{13})$$

we perform a permute on these vectors to get the multiplicand vectors

$$\text{vMul}_1 = (A_{00}, A_{01}, R_{00}, R_{01}, G_{00}, G_{01}, B_{00}, B_{01})$$

$$\text{vMul}_2 = (A_{02}, A_{03}, R_{02}, R_{03}, G_{02}, G_{03}, B_{02}, B_{03})$$

which, if we employ the same `msum, msum, sra` sequence given above, allows us to calculate the transform for the 0-th pixel of column zero of the transform image. Employing

different permute vectors, we can generate a similar set of multiplicand vectors for calculating the 0-th pixel of column one of the transform. In this way, we proceed to calculate the transform on two image columns per loop.

Let us next examine the problem of a YUV transform of an RGB image, with U and V channels subsampled 4 to 1. We begin with an interleaved RGB image (with no alpha channel). Since we will be subsampling, we loop across rows, but read two rows per loop to allow for averaging for the U and V channels. Beginning with rows 0 and 1, we start with six input vectors

$$\text{vRGB}_{00} = (r_{00}, g_{00}, b_{00}, r_{10}, g_{10}, b_{10}, ..., r_{50})$$

$$\text{vRGB}_{10} = (g_{50}, b_{50}, r_{60}, g_{60}, b_{60}, ..., r_{10,0}, g_{10,0})$$

$$\text{vRGB}_{20} = (b_{10,0}, r_{11,0}, g_{11,0}, b_{11,0}, ..., r_{15,0}, g_{15,0}, b_{15,0})$$

$$\text{vRGB}_{01} = (r_{01}, g_{01}, b_{01}, r_{11}, g_{11}, b_{11}, ..., r_{51})$$

$$\text{vRGB}_{11} = (g_{51}, b_{51}, r_{61}, g_{61}, b_{61}, ..., r_{10,1}, g_{10,1})$$

$$\text{vRGB}_{21} = (b_{10,1}, r_{11,0}, g_{11,0}, b_{11,0}, ..., r_{15,1}, g_{15,1}, b_{15,1})$$

The matrix L for conversion from RGB to YUV is given in shift-rational form in the previous section, and the goal is to calculate a matrix-vector product:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = L \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

To facilitate this computation, we form eight "quad expanded" RGB vectors of the form

$$\text{vRGB}_0 = (r_0, g_0, b_0, 0, r_1, g_1, b_1, 0, r_2, g_2, b_2, 0, r_3, g_3, b_3, 0)$$

so that the calculation of the Y channel for four pixels can be obtained by the operations:

$$\text{vY}_0 = \texttt{msum}(\text{vRGB}_0,\ \text{vYTransform},\ \text{vCorrector})$$

$$\text{vY}_0 = \texttt{sra}(\text{vY}_0,\ \text{vShiftRational})$$

with

$$\text{vYTransform} = (43, 42, 43, 0, 43, 42, 43, 0, 43, 42, 43, 0, 43, 42, 43, 0)$$

$$\text{vCorrector} = (64, 64, 64, ...)$$

$$\text{vShiftRational} = (7, 7, 7, ...)$$

yielding

$$\mathrm{vY}_0 = (\frac{43r_0 + 42g_0 + 43b_0 + 64}{128},$$
$$\frac{43r_1 + 42g_1 + 43b_1 + 64}{128},$$
$$\frac{43r_2 + 42g_2 + 43b_2 + 64}{128},$$
$$\frac{43r_3 + 42g_3 + 43b_3 + 64}{128}).$$

Note that this vector has four 32-bit elements. By calculating the Y transform vector for the adjacent four pixels, and then packing these two vectors into a single 8-element vector of 16-bit elements, we end up with a single vector that can be used to store eight 16-bit pixels in one instruction. This method is used to generate a Y channel transform for pixels in both input rows.

The U and V conversions are more involved, since we must subsample. Since the transform is a linear one, we can average the RGB pixels before the transform, thus reducing the number of required multiplies for the transform, which are relatively expensive operations. We begin by averaging (using the Velocity Engine `avg` operation) quad expanded vectors from our two input rows, for example:

$$\mathrm{vRGB}_{00} = \mathtt{avg}(\mathrm{vRGB}_{00}, \mathrm{vRGB}_{01})$$

Then, the permute operation is employed to generate vectors of even and odd entries of these averaged pixels:

$$\mathrm{vRGB}_{\mathrm{even}} = (r_0, g_0, b_0, 0, r_2, g_2, b_2, 0, r_4, g_4, b_4, 0, r_6, g_6, b_6, 0)$$
$$\mathrm{vRGB}_{\mathrm{odd}} = (r_1, g_1, b_1, 0, r_3, g_3, b_3, 0, r_5, g_5, b_5, 0, r_7, g_7, b_7, 0),$$

and then the `avg` operation is applied again to yield a row of four pixels that have been averaged (2 by 2) from the sixteen original input pixels. From here, the process for calculating U and V is the same as for Y, but with a different transform vector to match the appropriate row in the YUV transform matrix. It is fortunate that we can average in this way 16 pixels' worth of RGB information down to 4 pseudopixels, in only four vector instructions to effect the UV subsampling procedure.

The Burt 5/7 wavelet is somewhat more complex than the D4 discussed above, but the general strategy with respect to Velocity Engine implementation is essentially the same. For the Burt 5/7 wavelet, with a data stream $(x_0, ..., x_{n-1})$, and transform elements t_0 through $t_{\frac{n}{2}} - 1$ the equation for each element is

$$t_k = x_{2k-2}H_2 + x_{2k-1}H_1 + x_{2k}H_0 + x_{2k+1}H_1 + x_{2k+2}H_2.$$

whereas, for elements $t_{n/2}$ through t_{n-1}, the equation for each element is

$$t_{k+n/2} = x_{2k-3}G_3 + x_{2k-2}G_2 + x_{2k-1}G_1 + x_{2k}G_0 + x_{2k+1}G_1 + x_{2k+2}G_2 + x_{2k+3}G_3$$

We take the example of the latter in analyzing the steps necessary for the Velocity Engine implementation of the transform. In order to use the `msum` operation for this calculation, and to minimize the number of multiplies required, we need to generate the following vectors:

$$\begin{aligned}\text{vMul}_1 = (&x_{2k-3} + x_{2k+3}, x_{2k-2} + x_{2k+2},\\ &x_{2k-1} + x_{2k+5}, x_{2k} + x_{2k+4},\\ &x_{2k+1} + x_{2k+7}, x_{2k+2} + x_{2k+6},\\ &x_{2k+3} + x_{2k+9}, x_{2k+4} + x_{2k+8})\end{aligned}$$

$$\begin{aligned}\text{vMul}_2 = (&x_{2k-1} + x_{2k+1}, x_{2k},\\ &x_{2k+1} + x_{2k+3}, x_{2k+2},\\ &x_{2k+3} + x_{2k+5}, x_{2k+4},\\ &x_{2k+5} + x_{2k+7}, x_{2k+6})\end{aligned}$$

Given the input vectors

$$\text{vIn}_1 = (x_{2k-3}, x_{2k-2}, ..., x_{2k_4})$$

$$\text{vIn}_2 = (x_{2k+5}, x_{2k+6}, ..., x_{2k+12})$$

we can generate vMul_1 and vMul_2 with six instructions (one permute, two shifts, one select, and two adds). Given our familiar form of coefficient vectors:

$$\text{vGCoefficient}_1 = (G_3, G_2, G_3, G_2, G_3, G_2, G_3, G_2)$$

$$\text{vGCoefficient}_2 = (G_1, G_0, G_1, G_0, G_1, G_0, G_1, G_0)$$

and the "rounding corrector" and "shift-rational" vectors appropriate to our coefficients, we can then calculate the result vector vResult with the sequence of operations:

$$\text{vResult} = \texttt{msum}(\text{vMul}_1,\ \text{vGCoefficient}_1,\ \text{corrector});$$

$$\text{vResult} = \texttt{msum}(\text{vMul}_2,\ \text{vGCoefficient}_2,\ \text{vResult});$$

$$\text{vResult} = \texttt{sra}(\text{vResult},\ \text{vShift});$$

so that

$$\text{vResult} = (t_k, t_{k+1}, t_{k+2}, t_{k+3}).$$

Again, a similar technique is employed for the vertical transform (with four columns being calculated at a time) but with modified permutations as necessary to generate the appropriate multiplicand vectors.

Finally, we consider the subsampled YUV to RGB conversion. As with the forward RGB to YUV transform, we operate on two rows at a time to accomodate the subsampled U and V channels. The matrix for our transform is:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} \approx \begin{pmatrix} 1 & 0 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} Y \\ U \\ V \end{pmatrix}$$

which as we have explained is an accurate approximation in that the matrix here is a simple yet close approximation to the true L^{-1}. Starting with Y input vectors (the second index is the "row"):

$$\text{vY}_{11} = (Y_{00}, Y_{10}, Y_{20}, ..., Y_{70})$$

$$\text{vY}_{21} = (Y_{80}, Y_{90}, Y_{10,0}, ..., Y_{15,0})$$

$$\text{vY}_{12} = (Y_{01}, Y_{11}, Y_{21}, ..., Y_{71})$$

$$\text{vY}_{22} = (Y_{81}, Y_{91}, Y_{10,1}, ..., Y_{15,1})$$

and U and V input vectors

$$\text{vU} = (U_{00}, U_{10}, ...U_{70})$$

$$\text{vV} = (V_{00}, V_{10}, ...V_{70})$$

We use the Velocity Engine `mergeh` and `mergel` operations to form the vectors

$$\text{vUExpanded}_1 = (U_{00}, U_{00}, U_{10}, U_{10}, ..., U_{30}, U_{30})$$

$$\text{vUExpanded}_2 = (U_{40}, U_{40}, U_{50}, U_{50}, ..., U_{70}, U_{70})$$

and similarly form vVExpanded_1 and vVExpanded_2 for our V data, to expand the subsampled data to be the correct width. From this point, reconstructing R, G, and B from the YUV data is simply a matter of summing the appropriate vectors according to the transform matrix. So, for example, we have

$$\text{vRed} = (R_{00}, R_{10}, ..., R_{70}) = \text{vY}_{11} + \text{vVExpanded}_1.$$

To compensate for the vertical component of the subsampling, the expanded U and V vectors are used to generate two rows of RGB data.

Our resulting separate R, G, and B vectors have signed 16-bit elements, which we must convert to unsigned 8-bit elements. To do this, we must first ensure that our recovered data fits withing the range $[0, 255]$. This is accomplished using the Velocity Engine `min` and `max` operations to constrain the data. Then, Velocity Engine permute and shift instructions are used to recombine the RGB channels back into their original interleaved 8-bit form.

4. Performance results

Here we present timing and accuracy data for the various modes and image dimensions. In what follows "vector" means G4 300 MHz., while "scalar" means G3 engine at equivalent clock. By "RMSY" we mean the root-mean-square error:

$$RMSY = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(Y_i' - Y_i)^2},$$

where the sum is taken over all N pixels, is for the Y channel only, with Y' being the reconstructed Y after a full cycle of YUV conversion, subsampling of UV, 3 transforms, 3 inverse transforms, YUV deconversion. For full-color errors, we adopted the error measure:

$$RMSE = \sqrt{\frac{1}{3N}\sum((R' - R)^2 + (G' - G)^2 + (B' - B)^2)}.$$

Perhaps the most impressive data is the full-cycle timing for YUV mode with Burt 5/7 wavelet, below, in which we report a 320-by-240 image takes about 1/100 second, so the reconstruction (inverse half) component of the operation on the entire image takes about 1/200 second. (For 640-by-480 image the inversion half-cycle takes about 1/30 second.)

Performance data: Velocity Engine wavelet implementations

Daubechies D4 (RGBA interleaved) wavelet timing:

	640x480	320x240
vector forward	0.111 sec.	0.018 sec.
vector inverse	0.106 sec.	0.018 sec.
scalar forward	0.558 sec.	0.089 sec.
scalar inverse	0.559 sec.	0.093 sec.

Root-mean-square error (RMSE) for D4/RGBA:

	640x480	320x240
abe	0.758	0.773
fruit	1.010	1.060
rainbow	0.239	0.338
mallorca	0.881	1.030

YUV mode with Burt 5/7 wavelet:

	640x480	320x240
full forward-inverse cycle	0.057 sec.	0.009 sec.

RMSE for YUV mode (UV subsampled):

	640x480	320x240
Abe:		
RMSY	0.571	0.573
RMSE	1.450	3.370
Fruit:		
RMSY	0.546	0.545
RMSE	3.040	4.340
Rainbow:		
RMSY	0.533	0.513
RMSE	1.300	1.510
Mallorca:		
RMSY	0.564	0.563
RMSE	2.420	3.800

Acknowledgments

The authors are indebted to G. Miranker, J. Lu, H-J Wu, K. Chu, and A. Sazegari for insight and support relevant to this research.