Precision vs. Error in JPEG Compression

José Bins, Bruce A. Draper, Willem A.P. Böhm, and Walid Najjar
Computer Science Department, Colorado State University,
Fort Collins, CO, USA

ABSTRACT

By mapping computations directly onto hardware, reconfigurable machines promise a tremendous speed-up over traditional computers. However, executing floating-point operations directly in hardware is a waste of resources. Variable precision fixed-point arithmetic operations can save gates and reduce clock cycle times. This paper investigates the relation between precision and error for image compression/decompression. More precisely, this paper investigates the relationship between error and bit-precision for the Discrete Cosine Transform (DCT) and JPEG.

The present work is part of the Cameron project at the Computer Science Department of Colorado State University. This project is roughly divided in three areas: an C-like parallel language called SA-C that is targeted for image processing on reconfigurable computers, an implementation of the VSIP library for image processing in SA-C, and an optimizing compiler for SA-C that targets FPGAs.

Keywords: Image processing, Fixed-point arithmetic, Reconfigurable computer

1. INTRODUCTION

Field-programmable gate arrays (FPGAs) and other reconfigurable systems offer a fundamentally new model of computation in which programs are mapped directly onto hardware. FPGAs are composed of massive arrays of simple logic units, each of which can be programmed to compute arbitrary functions over small numbers of bits (usually four or five). In addition, the wires connecting the logic units have programmable interconnections. As a result, programs are mapped into circuits, which can then be dynamically loaded into the FPGA hardware. When the program is finished, the hardware is reconfigured into a new circuit for the next program. (Mangione-Smith[1,2] gives an introduction to reconfigurable computing, while Rose, et al[3] provide a general, if now somewhat dated, survey of reconfigurable processors, and Buell, et al[4] describe the application of FPGAs in practice.)

By mapping computations directly onto hardware, reconfigurable machines promise a tremendous speed-up over traditional computers. Part of this speed-up comes from *parallelism*, while the rest comes from increased *computational density*. The potential for parallelism is obvious; FPGAs are composed of thousands of logic blocks, and with reprogrammable interconnections the degree of parallelism is limited only by the data dependencies within the program (and I/O limitations). Computational density is equally important, however. Traditional processors implement a complex fetch-and-execute cycle that requires many special purpose units. Only a small fraction of the hardware (and time) is actually dedicated to the computation being performed. The computational density of these processors – the amount of space consumed per useful operation – is therefore very low. FPGA circuits, on the other hand, can be designed so that data "flows through" the circuit without repeated storage and retrieval, and without wasting resources on unused circuitry. This leads to more efficient processing and a potential speed-up of up to two orders of magnitude.[5,6]

One example of increased computational density in FPGAs is the ability to do variable precision arithmetic. While traditional processors have arithmetic units designed for fixed-width operations (either 16, 32 or 64 bits), FPGA circuits can be written for any bit precision. For example, if an algorithm calls for multiplying two seven-bit numbers, there is no need to allocate the resources of a 32-bit multiplier; a circuit for multiplying seven-bit numbers is created instead. Such variable precision arithmetic can save gates and reduce clock cycle times.

Variable precision arithmetic can be very powerful in image processing, where one-bit (binary), eight-bit (byte), and twelve-bit pixel values are common. It is less clear, however, whether variable precision arithmetic is useful in frequency-space applications where floating point numbers are typically used. One important example is JPEG image

E-mail: (bins,draper,bohm,najjar)@cs.colostate.edu

compression/decompression, which uses the Discrete Cosine Transform (DCT) and its inverse to convert images from the spatial domain to the frequency domain and back again. This paper investigates the relation between precision and error for JPEG image compression/decompression.

More precisely, this paper investigates the relationship between error and bit-precision first for the DCT and then for JPEG. In order to simplify the FPGA circuits (and because there is no floating point standard for anything shorter than 32 bits), we have implemented DCT and JPEG using fixed point, rather than floating point, arithmetic, and we measure the increase in reconstruction error as the precision of the fixed point values is decreased. Because DCT and JPEG depend on the frequency components of an image, we measure the precision/accuracy tradeoff for sets of real, artificial, and synthetic images created with different spectral components. Reconstruction error is measured in terms of total gray-level error, RMS gray-level error, RMS signal-to-noise ratio, and peak signal-to-noise ratio.[7] The results indicate that traditionally sized numbers are necessary for the addition tree in the DCT algorithm, but that a more moderate number of bits can be used for the multiplications. More significantly, far fewer bits are needed when the DCT is used within JPEG. These results are consistent across the spectrum of images tested.

2. THE DISCRETE COSINE TRANSFORM (DCT)

2.1. Definition of DCT

The DCT maps images from the intensity domain to the frequency domain. A simple, one-dimensional DCT is defined as:

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) cos \left[\frac{(2x+1)u\pi}{2N} \right] , \tag{1}$$

where

$$\alpha(u) \begin{cases} \sqrt{\frac{1}{N}} & for\ u = 0 \\ \sqrt{\frac{2}{N}} & for\ u = 1, 2, ..., N-1 \ . \end{cases} \tag{2}$$

The inverse DCT is similarly defined as:

$$f(x) = \sum_{u=0}^{N-1} \alpha(u) C(u) cos \left[\frac{(2x+1)u\pi}{2N} \right] . \tag{3}$$

For image compression, we need the two-dimensional form of the DCT, which is written as:

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y)\ cos \left[\frac{(2x+1)u\pi}{2N} \right] cos \left[\frac{(2y+1)v\pi}{2N} \right] . \tag{4}$$

Note that JPEG first divides the image into 8×8 sub-images, so that in the context of JPEG N always equals eight. The two dimensional inverse DCT (IDCT) is written as:

$$f(x,y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v) C(u,v)\ cos \left[\frac{(2x+1)u\pi}{2N} \right] cos \left[\frac{(2y+1)v\pi}{2N} \right] . \tag{5}$$

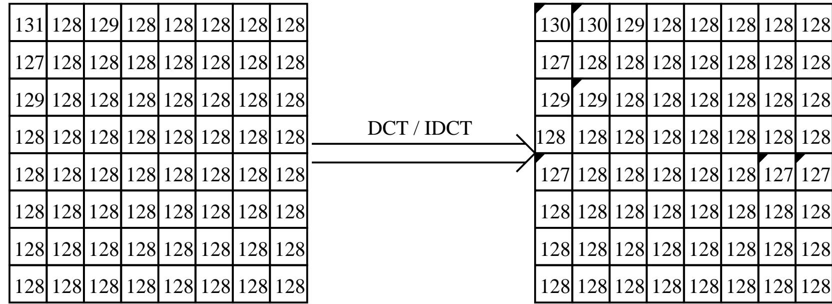
Blinn[8] provides an intuitive explanation of the DCT and some of its most important properties.

2.2. Minimum Error for DCT

Some readers may object to the idea of tolerating error in the DCT/IDCT. After all, we are taught that the DCT is an exactly invertible mapping between the spatial and frequency domains. Unfortunately, as commonly used the DCT/IDCT *does* introduce error. The question is how much error can be tolerated for a particular application.

To get some intuition about the error introduced by DCT, the equation for the one-dimensional DCT (Eq. 1) can be rewritten as:

$$c = Ax \tag{6}$$



131	128	129	128	128	128	128	128
127	128	128	128	128	128	128	128
129	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128

130	130	129	128	128	128	128	128
127	128	128	128	128	128	128	128
129	129	128	128	128	128	128	128
128	128	128	128	128	128	128	128
127	128	128	128	128	128	127	127
128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128

Figure 1. An 8 x 8 source image and the reconstructed image after DCT and IDCT. Errors are introduced into five pixels (shown with marked corners) as a result of rounding frequencies to the nearest integer.

where c is the vector (length $= N$) of frequency values, x is the initial vector of data, and A is a constant matrix of products of cosines and alphas. Note that the terms in A depend only on N, not on x, so A is constant as long as N is fixed.

As is clear from Eq. 6, the relationship between the spatial-domain pixels x and the frequency-domain values c is linear. Since A is non-degenerate, it can be inverted, leading to the inverse DCT. In fact, A is orthonormal, so $A^{-1} = A^T$, which explains the relationship between Eqs. 1 and 3. Unfortunately, as defined in the Vector, Signal and Image Processing Library (VSIPL*), the Intel Image Processing Library† and JPEG,[9] the output of the DCT is a frequency-domain image with integer pixels. This requires that the values of c be rounded to the nearest integer and introduces error into the reconstruction, even if infinite precision was used to calculate Ax. In particular, if a 1D signal is converted into the frequency domain by the DCT, has its values rounded to the nearest integer, and then is converted back into the spatial domain using the inverse DCT, there is a worst-case error in term x_i of $0.5 \times \sum_{j=0}^{N-1} |a_{i,j}^{-1}|$, where $a_{i,j}^{-1}$ is the ith row, jth column element of A^{-1}. For $N = 8$, this implies an error of up to 1.32 gray levels for one dimensional data. If the spatial-domain result of the IDCT is rounded back to the nearest integer (assuming the source data was integer), that still leaves the possibility of a reconstruction error of one gray level for any element x_i.

In two dimensions, the situation is similar. Once again, the frequency-space values are rounded to the nearest integer. Since rounding introduces an error of up to ± 0.5 in each frequency term, the maximum error $E_{x,y}$ for a pixel in the reconstructed image is

$$E_{x,y} \leq 0.5 \times \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} |\alpha(u)\alpha(v) cos\left(\frac{(2x+1)u\pi}{2N}\right) cos\left(\frac{(2y+1)v\pi}{2N}\right)| = 3.489$$

creating a possible final error of up to three gray levels. Figure 1 shows an image that is degraded by DCT/IDCT as a result of rounding the frequency values.

2.3. Implementation of DCT/IDCT

Because Eq. 4 is typically applied to 8×8 subwindows (and there are many such windows in any image), an efficient implementation is to precompute $cos\left[\frac{(2x+1)u\pi}{2N}\right] cos\left[\frac{(2y+1)v\pi}{2N}\right]$ and $\alpha(u)\alpha(v)$, storing them as $8 \times 8 \times 8 \times 8$ and 8×8 matrices, respectively. To compute the DCT, we then multiply the elements of these matrixes by the corresponding image pixels and sum the intermediate values, as shown in Fig. 2. Note that this implementation is designed to be fast on a parallel machine, where all the multiplications can be done in parallel, and the results can be added in a tree. A so-called Fast DCT algorithm,[10] which minimizes the total number of multiplications at the cost of a longer sequence of steps, is optimized for sequential processors.

In terms of precision, the variables in Fig. 2 can be grouped into two classes. The first are the stored cosine and α terms. Since these terms range from one to minus one, in fixed point notation they require at least two bits to

*see http://www.vsipl.org

†see http://developer.intel.com/vtune/perflibst/ipl

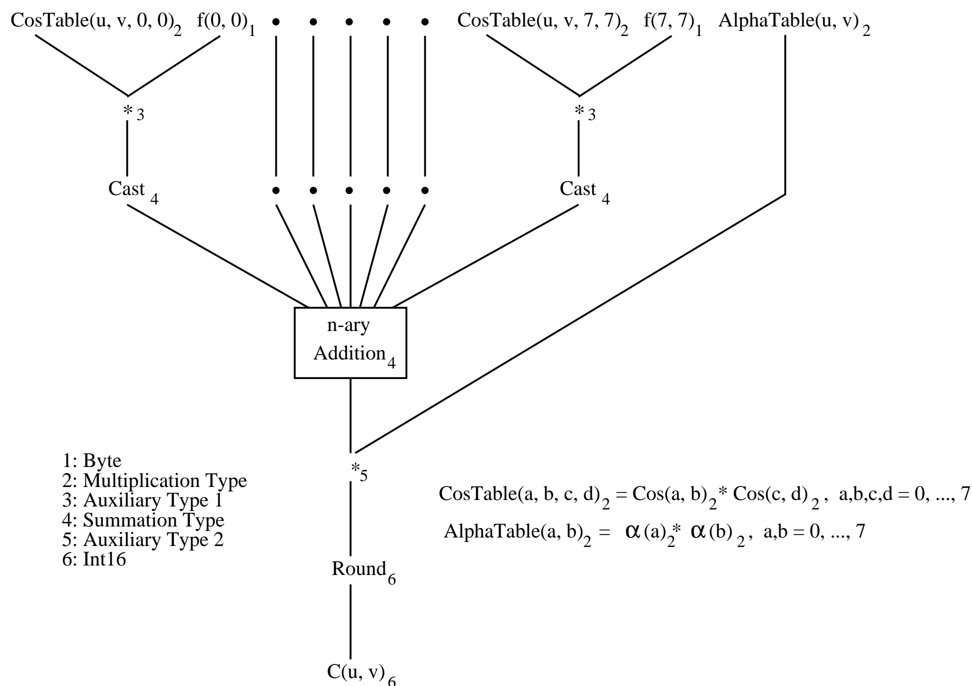


Figure 2. Data flow of the DCT computation for $C(u, v)$ showing types of variables and intermediate results. Auxiliary type 1 is the result of multiplying a byte with *multiplication type*, and auxiliary type 2 is the result of multiplying *summation type* and *multiplication type*. The type inference rules are given in Sect..?

the left of the binary point to represent their sign and integer magnitude. The number of bits to the right can be varied, however, to trade precision against accuracy. The precision of the cosine and α terms in turn determines the complexity of the multipliers in the circuit shown in Fig. 2. In the case of the multipliers at the top of the figure, these will be special-purpose circuits designed to multiply 8 bit integer pixel values to fixed point cosine terms. The higher the precision of the cosine terms, the greater the complexity of the multipliers, and therefore the overall circuit. We therefore refer to the precision of the cosine and α terms as the *multiplication type*.

The second group of variables are the terms used for the n-ary addition in Fig. 2. In hardware, the n-ary addition is implemented as a tree of binary additions. Since there are $8 \times 8 = 64$ terms being added, the n-ary addition is a depth-six binary tree of adders. In the forward DCT, the terms being added must have at least 16 bits to the left of binary point to hold the sign and integer magnitude of the final result, but we can once again vary the number of bits used to the right of the binary point to exchange precision for accuracy. We refer to the precision of these terms as the *summation type*.

For these experiments, we implemented the DCT and inverse DCT in SA-C,[11,12] a high-level programming language designed for FPGAs as part of the Cameron project.[13] The Cameron project is dedicated to making FPGAs available to non-circuit designers by creating a high-level language and optimizing compiler that target reconfigurable processors. SA-C is a single assignment dialect of C created for Cameron that, among other things, includes variable precision fixed point numbers. For example, in SA-C the datatype $fix14.8$ indicates a signed fixed point number with a total of 14 bits, eight of which are to the right of the binary point. (By subtraction, the other six bits are to the left of the binary point and signify the sign and integer size of the fixed point number.) In general, all fixed point types of the form $fixX.Y$, where $0 \leq Y \leq X \leq 32$, are supported by SA-C. (Unsigned fixed point numbers, written as $ufixX.Y$, are also supported.) SA-C's type inference rules define the result of an operation (e.g. addition or multiplication) between two fixed point values as a fixed point number with an integral size equal to the maximum integral sizes of the operands, and a fractional size equal to the maximum fractional size of the operands. The only limitation is that if the sum of the integral and fractional sizes exceeds 32 bits, the fractional size is reduced until the total size is 32.

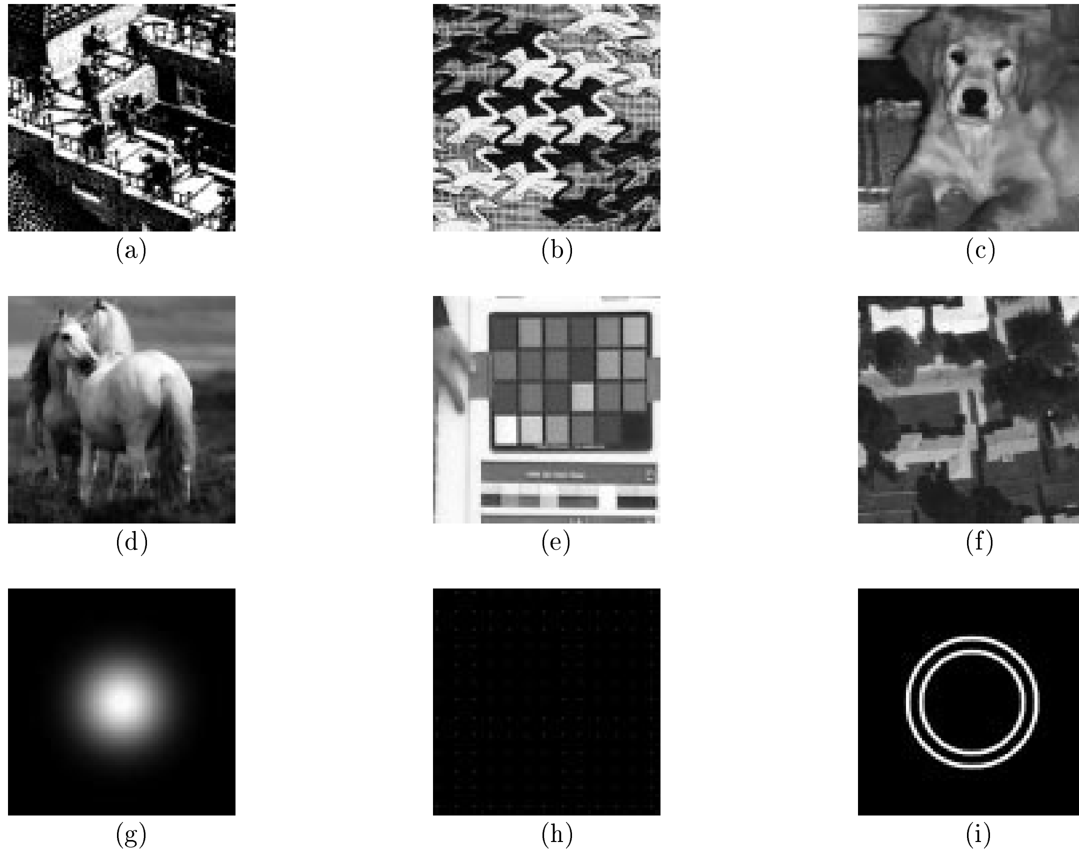


Figure 3. Subset of images used: (a) compacted/uncompacted drawing; (b) drawing; (c) compacted/uncompacted animal; (d) animal; (e) color palette; (f) houses at Fort Hood (TX); (g) Gaussian filter (Variance = 1000); (h) impulse images (impulse spacing = 20 pixels); and (i) concentric circumferences.

2.4. Evaluating DCT/IDCT

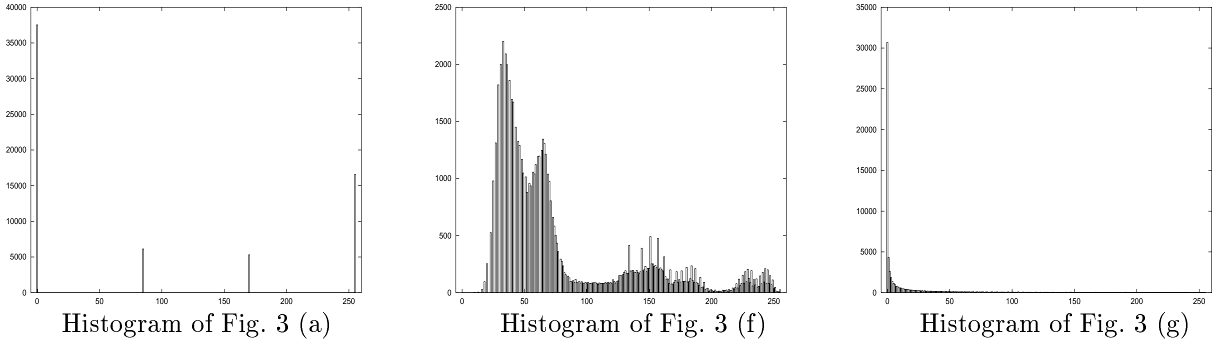
The data sets used to measure the relationship between precision and error in the DCT were composed of natural, artificial and synthetic images, some of which are shown in Fig. 3‡. The natural and artificial images were collected from the web. The artificial images are drawings extracted from Escher's work (five drawings). The natural ones are: six animal images, one color palette, one image of a seed, one aerial image of Fort Hood (TX), and one specular microscope image of cornea endothelial cells. All images were clipped to 256 x 256 and when necessary converted to black and white.

Because the DCT maps between the spatial and frequency domains, we also tested synthetic images containing controlled frequencies. We tested three Gaussian filter images with different variances, three noise images (uniform, Gaussian and exponential noise), two impulse images with different spacing between the impulses, one sinusoid image, one image of a constant-intensity circle against a constant background, and one image of two concentric rings.

Figure 4 shows the histogram for images *(a)*, *(f)*, and *(g)* of Fig. 3. As can be seen, image *(a)* (which had been previously compressed/uncompressed before we retrieved it off the web) has far fewer peaks. This image, together with image *(c)* (which was also previously compressed/uncompressed) were included to evaluate the influence of a sparse histogram on the reconstruction error. Some of the synthetic images also have sparse values due to the way they were constructed. At the extreme, images *(h)* and *(i)* are bitonal images (with values of 0 and 255).

To evaluate the effect of precision on the reconstruction error, we converted each image into the frequency domain using the DCT, rounded the frequency values to the nearest integer, and then converted them back into the spatial

‡The complete set of images can be seen at http://www.cs.colostate.edu/cameron/SPIE99.html



Histogram of Fig. 3 (a) Histogram of Fig. 3 (f) Histogram of Fig. 3 (g)

Figure 4. Histogram of images (a), (f), and (g) of Fig. 3.

Table 1. Reconstruction Error Measures: total error (upper left), RMS error (upper right), RMS signal-to-noise ratio (lower left), and peak signal-to-noise ratio (lower right).

$\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}[\hat{f}(x,y)-f(x,y)]$	$\sqrt{\frac{\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}[\hat{f}(x,y)-f(x,y)]^2}{N^2}}$
$\sqrt{\frac{\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}\hat{f}(x,y)^2}{\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}[\hat{f}(x,y)-f(x,y)]^2}}$	$10\ log_{10}\frac{[L-1]^2}{\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}[\hat{f}(x,y)-f(x,y)]^2}$

domain using the inverse DCT. We then compared the original images to the reconstructed versions. We repeated this process for all 26 images and for all combinations of the multiplication and summation types (i.e. precisions) The multiplication types tested were: float, fix32.30, fix28.26, fix26.22, fix18.16, fix17.15, fix16.14, fix15.13, fix14.12, fix12.10, fix10.8, fix8.6, fix6.4, and fix4.2. (Remember that the cosine and α terms always require two bits to the left of the binary point to represent their sign and integer magnitude.) The summation types require 16 bits to the left of the binary point to represent their integer magnitude and sign, so the summation types tested were: float, fix32.16, fix28.12, fix24.8, fix22.6, fix20.4, fix18.2 and integer. All 112 of these combinations were executed. The reconstructed images were compared to the originals using four measures (defined in Tab. 1)[7] : total error, RMS error, RMS signal-to-noise ratio, and peak signal-to-noise ratio.

Figure 5 shows the errors resulting from the DCT/IDCT reconstruction process on image *(f)* of Fig. 3. Each curve in Fig. 5 corresponds to one precision level for the summation type. The horizontal axes represent precisions associated with the multiplication type. In this case, restricting the multiplication type to sixteen bits to the right of the binary point (with two bits to the left) introduces a negligible amount of error[§], implying that 18 bits of precision are enough for these terms. This is significant, since hardware multipliers require more resources than adders. Unfortunately, the DCT is more sensitive to the summation precision. Using fixed point precision for the summation terms creates a significant amount of error. This implies that in a strict implementation of the DCT, the tree of additions (represented by the n-ary addition in Fig. 2) *must use floating point addition.*

Significantly, these results were consistent across the images tested. Figure 6 shows the results of the total error for three more images. In general, the natural and artificial images had almost indistinguishable error curves to each other, whether or not the images had been previously compressed. A small number of synthetic images presented slightly different results. The most significant difference was less error for some images where the background dominates. This is expected, since most of the 8×8 background windows are constant and equal to zero in this

[§]Where "negligible" means less than 0.01 gray levels per pixel on average.

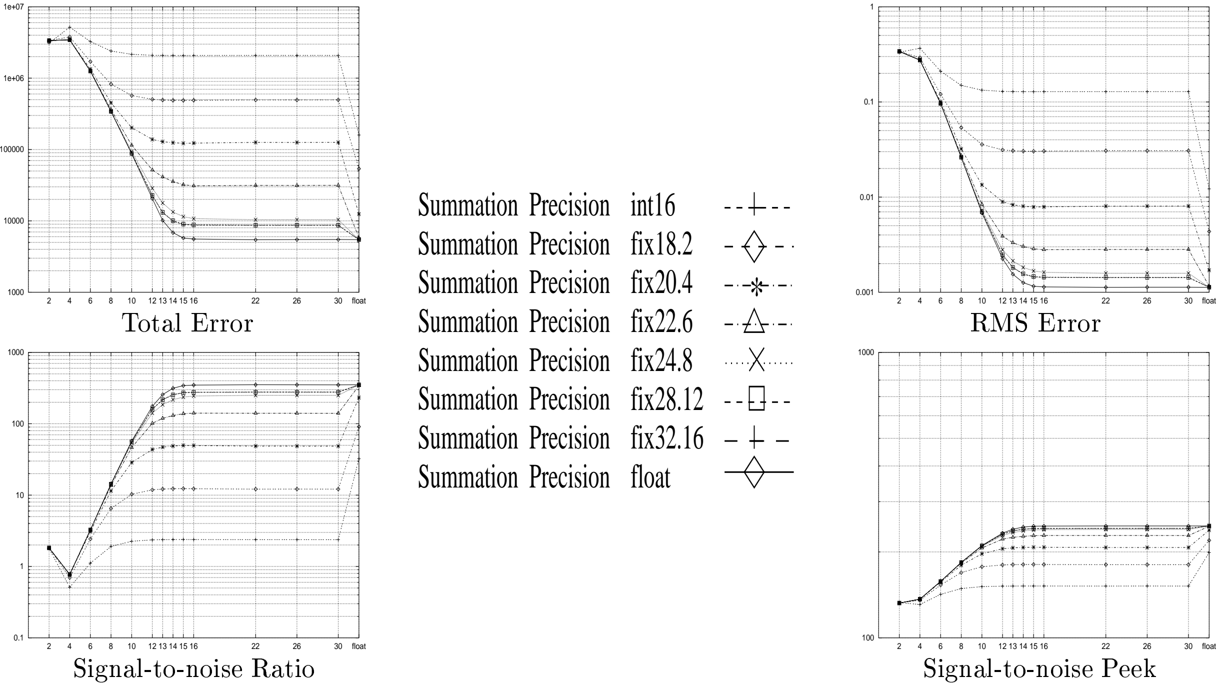


Figure 5. Result of DCT reconstruction for image *(f)* of Fig. 3 for each of the four measures.

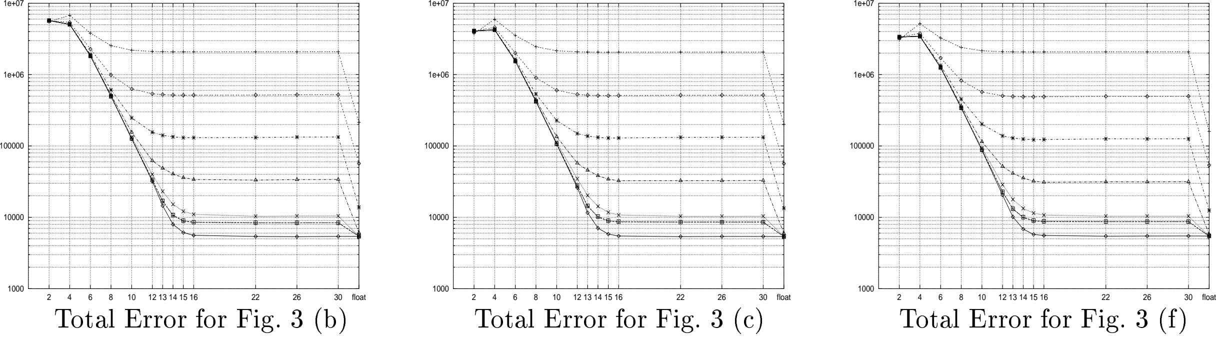


Figure 6. DCT Total Error for images (b), (c), and (f) of Fig. 3.

case. Nevertheless, the shape of the curves remain the same. As shown in Fig. 7, there is an unexplained anomaly in image *(i)* of Fig. 3, where the best result occurs for 12 bits of precision for the multiplication type.

In applications where a slight increase in error is tolerable for the DCT, faster circuits can still be constructed. Using floating point computation as the best case, we computed the difference between floating point arithmetic and combinations of fixed point precisions. This measures how much error is added by each combination of precisions. Figure 8 and Tab. 2 show the average increase in error for each pixel for all artificial and natural images. As can be seen in graph (b), although there is error, the error is very small for most precision combinations. For example, if an average increased error of one gray level or less is acceptable, a fractional precision of 12 bits for the multiplication type and 6 for the summation type are enough. That means 14 bits (2+12) for the cosine and α variables and 22 (16+6) for the internal summation variables, a savings of 18 and 10 bits respectively over a 32 bit floating point representation. Moreover, fixed point arithmetic requires fewer logic blocks and less time than floating point arithmetic does. The synthetic images have similar results, but the increase in error was even smaller. Figure 9 shows the average results for the synthetic images.

Total Error for Fig. 3 (g)

Total Error for Fig. 3 (i)

Figure 7. DCT Total Error for images (g) and (i) of Fig. 3.

(a)

(b)

(c)

Figure 8. Average increase in pixel error for all artificial and natural images. (a) shows the full graph, (b) shows the expanded graph, and (c) shows the thresholded graph (threshold error = 1).

Table 2. Average increase in pixel error for all artificial and natural images after DCT/IDCT. Rows represent multiplication types and columns represent summation types.

	float	fix32.16	fix28.14	fix24.8	fix22.6	fix20.4	fix18.2	int16
float	0.0000	0.0056	0.0056	0.0052	0.0108	0.1099	0.7142	2.5607
fix32.30	0.0000	0.0552	0.0558	0.0854	0.3994	1.8538	7.6533	31.4997
fix28.26	0.0001	0.0552	0.0558	0.0852	0.3974	1.8465	7.6313	31.4545
fix24.22	0.0000	0.0552	0.0559	0.0848	0.3953	1.8411	7.6246	31.4610
fix18.16	0.0020	0.0568	0.0584	0.0933	0.3985	1.8084	7.5531	31.4266
fix17.15	0.0079	0.0622	0.0639	0.1082	0.4258	1.8047	7.5216	31.3574
fix16.14	0.0326	0.0865	0.0886	0.1502	0.4920	1.8547	7.5523	31.3954
fix15.13	0.1187	0.1682	0.1703	0.2611	0.6195	1.9602	7.6588	31.5136
fix14.12	0.3638	0.4003	0.4003	0.4982	0.8215	2.1686	7.8722	31.7296
fix12.10	1.7140	1.7440	1.7440	1.8243	2.1502	3.5058	9.2145	33.0276
fix10.8	6.9594	6.9908	6.9908	6.9908	7.3059	8.6650	14.3759	38.1041
fix8.6	25.8551	25.8796	25.8796	25.8796	25.8796	27.1488	32.6963	55.9756
fix6.4	71.2694	71.2547	71.2547	71.2547	71.2547	71.2547	76.3535	97.3901
fix4.2	71.3656	71.5043	71.5043	71.5043	71.5043	71.5043	71.5043	66.9298

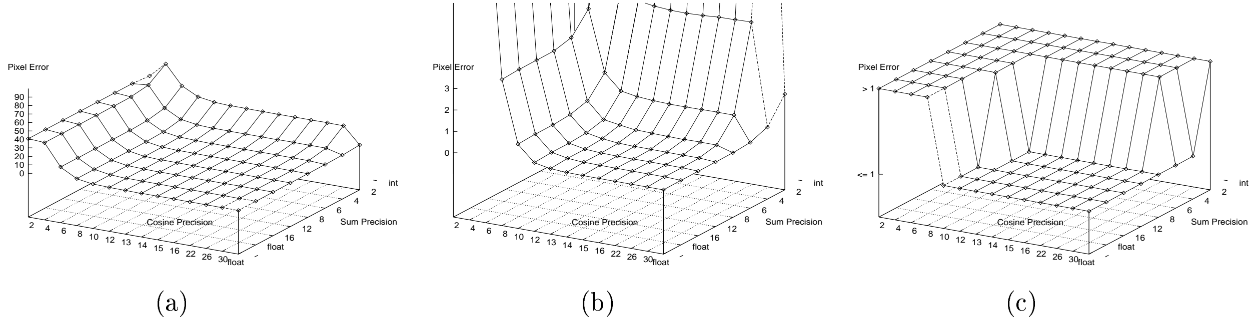


Figure 9. Average increase in pixel error for all synthetic images. (a) shows the full graph, (b) shows the expanded graph, and (c) shows the thresholded graph (threshold error = 1).

For each $8x8$ sub-image:

1. subtract 128 from pixel values
2. apply Discrete Cosine Transform (DCT)
3. normalize: $newval(x, y) = round(f(x, y)/table(x, y))$
4. linearize values into a vector of length 64 (S-pattern)
5. discard trailing zeroes
6. apply Huffman code
7. mark end of sub-image

Figure 10. Steps of the JPEG image compression algorithm[9]

3. JPEG

3.1. Definition of JPEG

The transmission and storage of large images has became more and more common, increasing the need for fast compression algorithms. The most widely-used compression algorithm for still images is JPEG,[9] a lossy compression technique common on the world wide web. As shown in Fig. 10, JPEG divides images into 8x8 sub-images. For each sub-image, JPEG applies the discrete cosine transform, converting the sub-window into frequency space. It then divides the frequency values according to a user-selected normalization table (rounding the result), and encodes the scaled frequencies with a Huffman code.[14] Images are uncompressed by inverting this process.

Of the seven steps in the JPEG algorithm, only two introduce noise. As discussed above, the discrete cosine transform introduces a small but measurable amount of noise, even when computed with 32 or 64 bit floating point numbers. Most of the error (and most of the compression), however, comes from the normalization table in step 3. In effect, this table allocates a fixed range of numbers for every frequency, dividing values accordingly. Users may select a compression level from 0 to 100, where compression level 100 is lossless (in terms of the normalization step) and level 0 is maximum compression. All the experiments in this paper were conducted at compression level 50.

3.2. Precision vs. Error in JPEG

When the DCT and inverse DCT are used within the context of JPEG, more error can be tolerated and therefore less precision is required. This occurs because of the normalization step of the JPEG algorithm (step 3 in Fig. 10). For compression purposes, this step allocates a fixed range of values for every frequency, in essence discarding the least significant bits on a frequency-by-frequency basis. As a result, small errors that may be significant when DCT/IDCT are considered in isolation become insignificant because of normalization within the context of JPEG.

Figure 11 presents the error curves for image *(f)* of Fig. 3 for different precisions of the DCT/IDCT terms when used in the context of JPEG (compression level 50). Suddenly, the bottom six curves are indistinguishable. This implies that floating point precision is no longer needed for the summation type. In fact four bits of precision to the

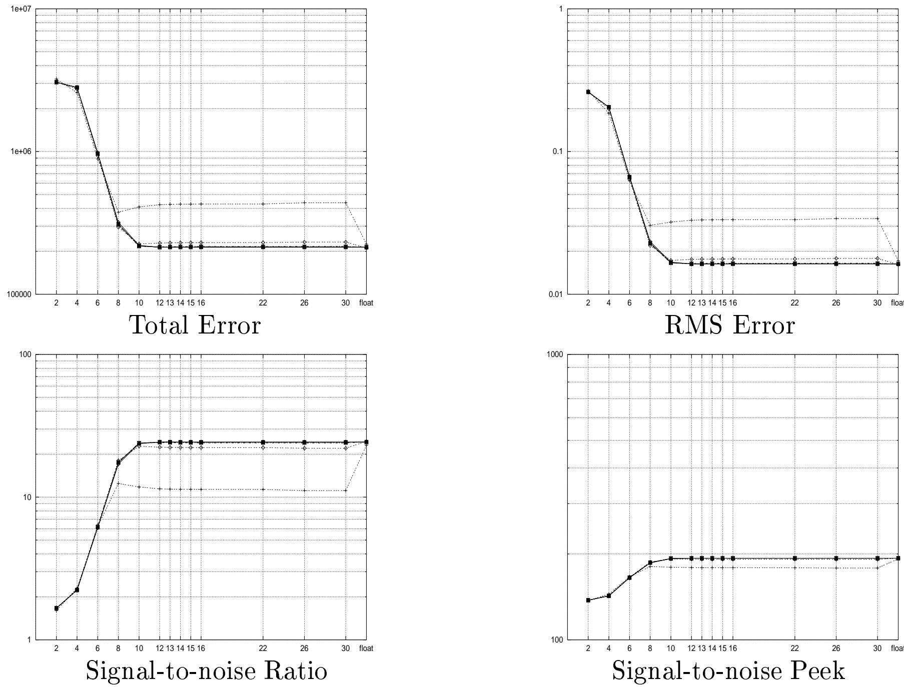


Figure 11. Result of JPEG reconstruction for image *(f)* of Fig. 3 for each of the four measures.

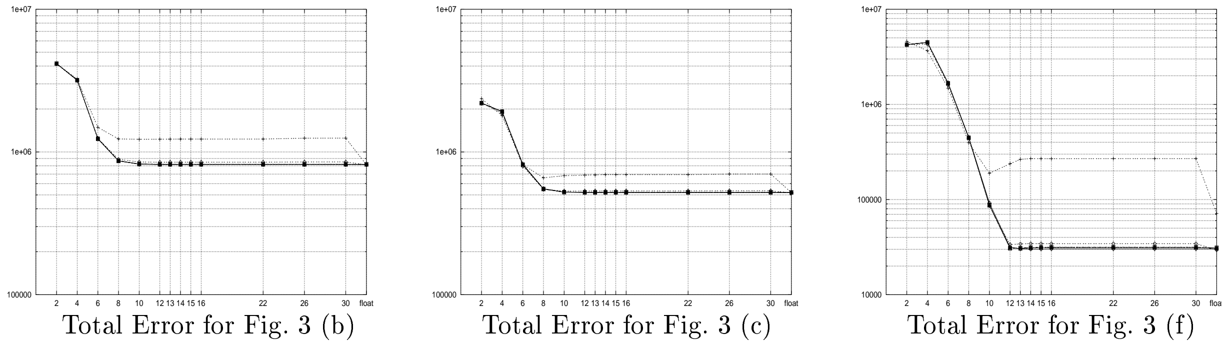


Figure 12. JPEG Total error for images (b), (c), and (f) of Fig. 3.

right of the binary point (20 bits total) is enough – the rest of the information is discarded during normalization anyway! Similarly, the multiplication type now require only 10 bits to the right of the binary points (12 total).

As shown in Fig. 12, although the absolute error varies between images, the shape of the curves and the consequences remain the same as for image *(f)* of Fig. 3. On the other hand, these results are highly dependent on the compression level being used. On one extreme, (level 100) no normalization is applied and float and fix18.16 precisions (respectively for summation and multiplication types) are required, as for the DCT in isolation. At level 50, precisions of fix20.4 and fix12.10 are sufficient (see Fig. 13 and Tab. 3), a saving of 12 and 20 bits respectively. Presumably, higher levels of compression would permit even smaller representations. A web site (www.cs.colostate.edu/cameron/SPIE99.html) presents all four error tables for all the precision combinations tested.

4. CONCLUSION

Reconfigurable processors can be more efficient than traditional machines in part because of increased computational density. Variable precision arithmetic is one mechanism for further increasing the computation density of reconfigurable processors. To test whether less precise arithmetic could be used for frequency-space computations, we

(a) (b) (c)

Figure 13. Average increase in pixel error for all artificial and natural images. (a) shows the full graph, (b) shows the expanded graph, and (c) shows the thresholded graph (threshold error = 1).

Table 3. Average increase in pixel error for all artificial and natural images after compression/uncompression using JPEG. Rows correspond to multiplication types and columns correspond to summation types. Values marked with † indicate that there was a decrease in error instead of an increase; these decreases are the result of random variation and suggest that the change in error as a result of reduced precision is not statistically significant.

	float	fix32.16	fix28.14	fix24.8	fix22.6	fix20.4	fix18.2	int16
float	0.0000	0.0071	0.0070	0.0058	0.0015	0.0133†	0.0589†	0.0490
fix32.30	0.0000	0.0115	0.0116	0.0125	0.0162	0.0382	0.2919	4.0913
fix28.26	0.0001	0.0115	0.0116	0.0124	0.0161	0.0380	0.2899	4.0891
fix24.22	0.0001†	0.0114	0.0116	0.0123	0.0156	0.0350	0.2636	3.9335
fix18.16	0.0002†	0.0111	0.0110	0.0119	0.0153	0.0347	0.2628	3.9331
fix17.15	0.0005†	0.0107	0.0107	0.0114	0.0151	0.0341	0.2613	3.9307
fix16.14	0.0004†	0.0100	0.0101	0.0102	0.0144	0.0336	0.2595	3.9258
fix15.13	0.0005†	0.0087	0.0087	0.0101	0.0135	0.0323	0.2564	3.9190
fix14.12	0.0021	0.0083	0.0083	0.0091	0.0123	0.0311	0.2498	3.8936
fix12.10	0.0946	0.0778	0.0778	0.0776	0.0781	0.0916	0.2866	3.8197
fix10.8	1.0245	0.9748	0.9748	0.9748	0.9711	0.9653	1.0685	3.9346
fix8.6	7.4684	7.3601	7.3601	7.3601	7.3601	7.3250	7.2531	8.7778
fix6.4	29.9439	29.8395	29.8395	29.8395	29.8395	29.8395	29.5013	29.5608
fix4.2	35.3315	35.3536	35.3536	35.3536	35.3536	35.3536	35.3536	36.3420

tested the results of various precisions on DCT and JPEG. We concluded that some savings can be applied to DCT by using 18 bit fixed point numbers for the multiplication type, but that floating point addition is still required. Within the context of JPEG (compression level 50), however, a much greater savings can be achieved by using 20 bit (summation) and 12 bit (multiplication) fixed point numbers for DCT. This should result in significantly smaller and faster circuits. Presumably, higher levels of compression would permit even smaller representations. Moreover the results are consistent over a great range of images, over different histogram and frequency spectra what indicates that the results are general.

REFERENCES

1. W. Mangione-Smith, "Seeking solutions in configurable computing," *IEEE Computer* **30**, pp. 38–43, Dec. 1997.
2. W. Mangione-Smith, "Application design for configurable computing," *Computer* **30**, pp. 115–117, Oct. 1997.
3. J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE* **81**(7), pp. 1013–1029, 1993.
4. D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE CS Press, 1996.
5. A. DeHon and J. Wawrzynek, "The case for reconfigurable processors." 1997.
6. A. DeHon, "Comparing computing machines: Technology and applications," in *Proceedings of SPIE 3526*, Nov. 1998.
7. S. E. Umbaugh, *Computer Vision and Image Processing: a pratical approach using CVIPtools*, Prentice-Hall, London, 1998.
8. J. F. Blinn, "What's the deal with the dct?," *IEEE Computer Graphics and Applications* **13**, pp. 78–83, July 1993.
9. G. K. Wallace, "The jpeg still picture compression standard," *Communications of the ACM* **34**(4), pp. 30–44, 1991.
10. C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-d dct algorithms with 11 multiplications," in *Internation Conference on Accoustics, Speech and Signal Processing*, pp. 988–992, 1989.
11. J. P. Hammes and W. Bohm, *The Sassy Language - Version 1.0*, 1999.
12. J. Hammes, B. Draper, and W. Bohm, "Sassy: A language and optimizing compiler for image processing on reconfigurable computing systems," in *Proceedings: International Conference on Vision Systems*, pp. 522–537, (Las Palmas de Gran Canaria, Spain), January 1999.
13. W. Najjar, B. A. Draper, W. Bohm, and J. R. Beveridge, "The cameron project: High-level programming of image processing applications on reconfigurable computing machines," in *Workshop on Reconfigurable Computing*, pp. 83–88, (Paris), Oct. 1998.
14. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Addison-Wesley, Reading, MA, 1992.