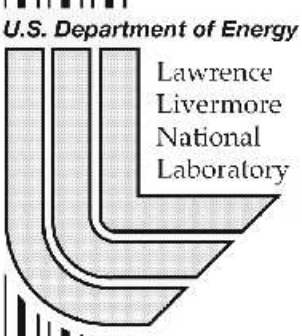
UCRL-CONF-200745

Reversible n–Bit to n–Bit Integer Haar–Like Transforms

Joshua Senecal
Mark Duchaineau
Kenneth I. Joy

This paper was submitted to the 2004 Data Compression Conference, to be held at Snowbird, Utah, March 23–25, 2004



November 4, 2003

DISCLAIMER

Reversible n–Bit to n–Bit Integer Haar–Like Transforms

Joshua Senecal[*‡] Mark Duchaineau[†] Kenneth I. Joy[‡]

[*]Institute for Scientific Computing Research
[†]Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

[‡]Center for Image Processing and Integrated Computing
Computer Science Department
University of California, Davis

Abstract

We introduce a wavelet–like transform similar to the Haar transform, but with the properties that it packs the results into the same number of bits as the original data, and is reversible. Our method, called TLHaar, uses table lookups to replace the averaging, differencing, and bit shifting performed in a Haar Integer Wavelet Transform (IWT). TLHaar maintains the same coefficient magnitude relationships for the low– and high–pass coefficients as true Haar, but reorders them to fit into the same number of bits as the input signal, thus eliminating the sign bit that is added to the Haar IWT output coefficients. Eliminating the sign bit avoids using extra memory and speeds the transform process. We tested TLHaar on a variety of image types, and when compared to the Haar IWT TLHaar is significantly faster. For image data with lines or hard edges TLHaar coefficients compress better than those of the Haar IWT. Due to its speed TLHaar is suitable for streaming hardware implementations with fixed data sizes, such as DVI channels.

1 Introduction

The Haar transform is probably the simplest and best known of the wavelet transforms [9]. It consists of a series of averaging and difference steps, each step operating on two adjacent low–pass values A and B and producing a low–pass value $\tilde{L} = (A+B)/2$ and a high–pass value $\tilde{H} = (A-B)/2$. The next iteration of the transform is performed on the low–pass values resulting from the previous iteration, and the process repeats until there is only a single low–pass value remaining. The original Haar transform, even if operating on integer values, produces floating–point coefficients due to the averaging step. For speed, and to ensure that data is not lost in the transform process, a Haar integer wavelet transform (IHaar) approach is often used [7], where $\hat{L} = \lfloor (A+B)/2 \rfloor$ and $\hat{H} = B - A$[§].

[*]L–419, PO Box 808, Livermore, CA 94551, Tel: 925–422–3764, Fax: 925–422–7819 *senecal1@llnl.gov*

[†]L–561, PO Box 808, Livermore, CA 94551, *duchaine@llnl.gov*

[‡]One Shields Ave, Davis, CA 95616, *kijoy@ucdavis.edu*

[§]Throughout this paper $\tilde{H}$ and $\tilde{L}$ denote coefficients produced by Haar, $\hat{H}$ and $\hat{L}$ denote those produced by IHaar, and H and L denote those produced by our method, TLHaar.

A shortcoming of the IHaar transform is that, due to the subtraction that occurs in the transform procedure, it is necessary to store a sign bit for all nonzero high–pass coefficients. These sign bits present some problems. First, there is raw data inflation: the number of bits required to store the transformed data is greater than that required to store the original data. Second, an actual implementation that operates (for example) on 8–bit values must represent them using 16 bits. Third, the sign bits often appear to have a random distribution, and if not handled with care they can hurt the compression potential of the transformed data. In [11] a wavelet transform for binary images (bilevel, 1 bit per pixel) that solves these problems is described.

We present our solution to these problems in the more general greylevel case: the Table–Lookup Haar (TLHaar) method, a reversible n–bit to n–bit transform. TLHaar uses two lookup tables (LUTs) called AB2HL and HL2AB, each of size 2^{2n}. Given A and B, $(H, L) = AB2HL(A, B)$. Thus, during the transform process only table lookups and bit shifts are performed when operating on data values.

To evaluate TLHaar we assembled a suite of 8–bit image sets, with each set containing images of a particular type (bilevel, shaded line art, photographs, etc.). We then implemented and optimized the TLHaar and IHaar transforms. We recorded their execution times and compressed the coefficients they produced with several coders. Our tests indicate that TLHaar is faster than IHaar, particularly when transforming data in large chunks. For data that have sharp edges, such as bilevel, line art, and computer generated images, coefficients generated by TLHaar compress better than IHaar. For other types of data, such as MRI, coefficients compress from 0.013% to 1.78% worse, depending on the compression method used.

2 Table–Lookup Haar

Table–Lookup Haar (TLHaar) performs a Haar–like wavelet transform, replacing the averaging and differencing steps with a single table lookup. TLHaar uses a set of two 2D lookup tables, called AB2HL and HL2AB. Each table contains 2^{2n} entries, with each entry being $2n$ bits wide, the upper and lower n bits each containing a value. AB2HL is used when performing a forward transform. It takes two n–bit data values A and B as indices and produces a $2n$ bit value, where the upper n bits are the high–pass value and the lower n bits are the low–pass value. HL2AB is used when reversing a transform, similarly converting (H, L) to (A, B). The transform process is illustrated in figure 1.

TLHaar is related to the IHaar transform. The IHaar transform process is described by the equations at the left of figure 2. These can be interpreted as a lookup table, as shown at the right of the same figure, where the sign of $\hat{H}$ can be thought of as an indicator as to whether the table entry is above or below the diagonal. Given $\hat{L}$ and the magnitude of $\hat{H}$, it is easy to determine what pair of values should be assigned to A and B, but the ordering of the pair is ambiguous. The sign bit is what disambiguates this.

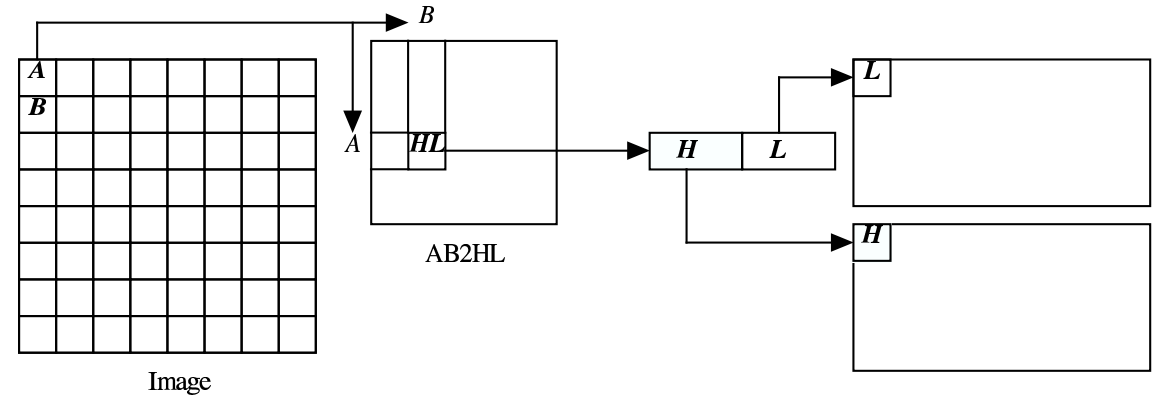


Figure 1: TLHaar transform process.

$$\hat{L} = \left\lfloor \frac{A+B}{2.0} \right\rfloor$$

$$\hat{H} = A - B$$

A \ B	0	1	2	3	...
0	(0,0)	(-1,0)	(-2,1)	(-3,1)	
1	(1,0)	(0,1)	(-1,1)	(-2,2)	
2	(2,1)	(1,1)	(0,2)	(-1,2)	
3	(3,1)	(2,2)	(1,2)	(0,3)	
...					...

Figure 2: The IHaar equations, and the equivalent transform table.

2.1 Transform LUTs

The LUTs used in TLHaar have the following properties:

$$|\tilde{H}_i| \leq |\tilde{H}_j| \iff |H_i| \leq |H_j| \quad (1)$$

$$|\tilde{L}_i| \leq |\tilde{L}_j| \iff |L_i| \leq |L_j|. \quad (2)$$

That is, for any given two pairs of data values their high– and low–pass values as created by TLHaar will have the same relationships as those created by Haar. For the transform table to be reversible there must be a 1:1 mapping between entries in the two tables. We therefore initialize each with an identity transform: AB2HL$[i, j]$ $= (i, j)$, HL2AB$[i, j] = (i, j)$. We then rearrange the entries in HL2AB and AB2HL so properties 1 and 2 hold. We accomplish this via a sort of the LUTs according to the following pseudocode:

```
do {
    For each L Column in HL2AB
        Sort based on |(A − B)/2.0|
    For each H Row in HL2AB
        Sort based on (A + B)/2.0
} while (there was a swap)
```

During the sort process whenever a swap occurs in HL2AB the corresponding entries in AB2HL are also swapped.

It was not clear beforehand that this sort would converge. At this time we do not have a general proof that the sort will always converge, however we tested this process of creating tables for values $2 \leq n \leq 12$, and our tests indicate that in all cases the sort converges. We are unable to test further since when $n > 12$ the tables become so large they are impractical. For $n = 13$ a single LUT will contain over 67 million entries and be 256 megabytes in size.

Figure 3 shows AB2HL and HL2AB for $n = 3$ before and after sorting. Comparing with the table in figure 2, we see that for the TLHaar $(A, B) \rightarrow H$ transform the entries on the diagonal are equivalent to IHaar. For example, (3,3) produces an H of 0 and an L of 3. But further from the diagonal the entries do not approximate IHaar as well. Because TLHaar does not use sign bits, instead using an identity transform to obtain a 1:1 mapping, the LUTs must contain (H,L) and (A,B) pairs that do not exist in the IHaar transform table.

AB2HL **Before Sorting** HL2AB

A \ B	0	1	2	3	4	5	6	7
0	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7	7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7

H \ L	0	1	2	3	4	5	6	7
0	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7	7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7

After Sorting

A \ B	0	1	2	3	4	5	6	7
0	0,0	1,1	3,0	4,0	6,0	7,0	7,2	7,4
1	1,0	0,1	1,2	3,3	5,1	5,2	6,3	6,5
2	2,0	2,1	0,2	1,3	3,4	5,5	6,6	7,7
3	3,1	3,2	2,2	0,3	1,4	3,5	4,4	5,6
4	5,0	4,1	4,2	2,3	0,4	1,5	3,6	4,7
5	6,1	5,3	5,4	4,3	2,4	0,5	1,6	2,6
6	7,1	6,2	6,4	4,5	4,6	2,5	0,6	1,7
7	7,3	7,5	7,6	6,7	5,7	3,7	2,7	0,7

H \ L	0	1	2	3	4	5	6	7
0	0,0	1,1	2,2	3,3	4,4	5,5	6,6	7,7
1	1,0	0,1	1,2	2,3	3,4	4,5	5,6	6,7
2	2,0	2,1	3,2	4,3	5,4	6,5	5,7	7,6
3	0,2	3,0	3,1	1,3	2,4	3,5	4,6	7,5
4	0,3	4,1	4,2	5,3	3,6	6,3	6,4	4,7
5	4,0	1,4	1,5	5,1	5,2	2,5	3,7	7,4
6	0,4	5,0	6,1	1,6	6,2	1,7	2,6	7,3
7	0,5	6,0	0,6	7,0	0,7	7,1	7,2	2,7

Figure 3: The $n = 3$ LUTs before and after sorting.

2.2 Sorted LUTs Are Not Unique

In section 2.1 we demonstrated the method used for creating the TLHaar LUTs, using a sort on an identity transform. If the LUTs are initialized with a 1:1 mapping different from our original method, will the sort still result in the same transform LUTs? The answer to this question is no, implying that LUTs satisfying relationships 1 and 2 are not unique. As a test we permuted one of the initialized LUTs before sorting, by randomly swapping entries. The corresponding entries in the other LUT were also swapped, maintaining a 1:1 mapping. We then performed our sort on the permuted tables. The resulting transform tables were different from those of our original method. By varying the number of swaps and the seed to the pseudorandom number generator we produced tables that were similar to each other, but not identical.

3 TLHaar Implementation Optimizations

Because TLHaar operates on and produces n–bit data, when n is both a power of 2 and an integer size common in modern computer architectures (8–bit byte, 16–bit short integer, etc.) it is possible to store the low–pass and high–pass values in arrays of that integer type. This allows us to implement and take better advantage of some special optimizations. Here we describe optimizations made for an implementation that operates on 8–bit images.

We first altered how we perform table lookups in a row transform. Since input values A and B are adjacent in memory, instead of reading A and B separately and indexing the AB2HL LUT with both (i.e. $HL = AB2HL[A][B]$) we cast the input array of bytes into an array of 16–bit short integers, and read A and B together as a single short AB. This allows us to perform a complete table lookup using fewer operations: $HL = AB2HL[AB]$.

We would like to use the above optimization when performing a transform in the column direction. The standard row transform operates on an image one row at a time, writing out the resulting low–pass values such that they are contiguous in the row direction. Thus a given image column is not contiguous in memory. To solve this when performing a row transform we transform two rows at a time. Given the k-th pair of pixels from rows i and $i+1$ we transform A_iB_i and $A_{i+1}B_{i+1}$, and place L_i and L_{i+1} adjacent to each other in preparation for the column transform. The column transform can then proceed down columns in image space, but along adjacent memory locations. The idea behind the optimization is shown in figure 4. The right side shows the low–pass values ordered in both memory and image space, the bold lines indicating adjacent memory values.

4 Results

To obtain the results in this paper we assembled a suite of 8–bit images. The images fall into the following categories:

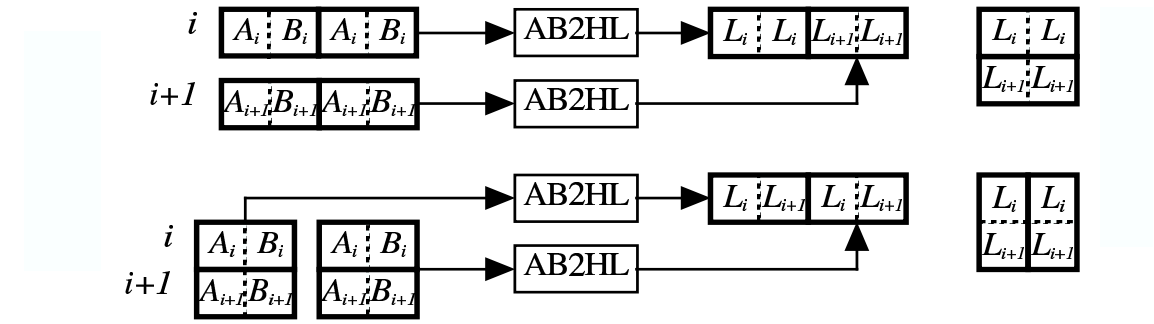


Figure 4: The TLHaar data reordering optimization. The top shows the normal transform procedure, and the bottom the reordering procedure.

- BW_Lines. A collection of 106 bilevel (black–and–white) line figures. Each figure is small, an example size being 108 by 110 pixels [1].
- LineArt. A set of 18 line art images. Each image has shading, gradient fills, and the like.
- ObjectBank. A set of 122 computer–rendered images of everyday objects [1].
- MRI. A set of 185 MRI scans. Each image is 256 by 256 pixels [2].
- ccitt. Eight of the standard bilevel (black–and–white) ccitt FAX test images [3]. Each is 1728 by 2376 pixels.
- DB1_B. A set of 80 fingerprint scans[4]. Each is 300 by 300 pixels.
- DB2_B. A set of 80 fingerprint scans. These are the same fingerprints as in DB1_B, but each image is 256 by 364 pixels, and the images have been processed to bring out details.
- Photos. A set of 29 photographic images [5, 6]. These include standard test images (such as "Lena") and personal photographs from the lead author.
- r2_slices. A set of 221 randomly selected images extracted from the data produced by a Richtmeyer–Meshkov mixing simulation, described in [8].
- Power2. This is a selection of images from Photos, where each image is square and has an edge length that is a power of 2. These are used to evaluate the reordering method described in section 3.

In each table of results the column heading "% Gain" indicates gains obtained using TLHaar over IHaar (positive percentage meaning TLHaar is better).

4.1 Execution Time

To compare execution times we implemented classes to perform IHaar and TLHaar transforms on 8–bit grayscale images, and optimized each separately. We also implemented versions of TLHaar and IHaar that perform data reordering and operate on square images with edge lengths that are a power of 2. Timings were taken on a 550 MHz PowerBook G4 running MacOS 10.1.5. The time given is an average over 10 transform runs, where a run transforms all images in a particular category, and includes only the time taken to transform the image. Execution times are given in table 1.

	Transform Time (sec)		
Category	TLHaar	IHaar	% Gain
BW_Lines	0.06368	0.06540	2.63
LineArt	0.15289	0.27621	44.65
ObjectBank	0.90947	1.08220	15.96
MRI	0.45442	0.55188	17.66
MRI (reord)	0.38121	0.51650	26.19
ccitt	1.31669	1.96779	33.09
DB1_B	0.25860	0.36670	29.48
DB2_B	0.31501	0.35577	11.46
Photos	1.02017	1.34319	24.05
r2_slices	0.31479	0.39773	20.85
Power2	0.23322	0.31506	25.98
Power2 (reord)	0.20557	0.29665	30.70

Table 1: Our test image categories and their transform times. (reord) indicates execution time using the reordering method of section 3.

4.2 Compressibility of Coefficients

There is no absolute way to determine how well a set of wavelet coefficients compresses, as the amount of compression obtained is determined by the methods used to find and exploit any redundancy present. We are unable to review the many methods that are possible. To demonstrate how coefficients produced by TLHaar compress compared to those produced by IHaar we transformed the test images and then compressed the results using three freely available compression programs: gzip[1], bzip[2], and an arithmetic coder available from Alistair Moffat[3]. We used binary and byte arithmetic encoding.

To gauge the effect of sign bits on the compressibility of IHaar coefficients we compressed them in two ways. In the first method the coefficient magnitudes were written as a stream of bytes and compressed, and the sign bit for each nonzero

[1] `http://www.gzip.org`

[2] `http://sources.redhat.com/bzip2/`

[3] `http://www.cs.mu.oz.au/~alistair/arith_coder/`

magnitude was appended uncompressed. In the second method coefficient magnitudes were written as a stream of bytes, and then a binary stream consisting of the sign bits of all nonzero magnitudes was appended. The combined stream was then compressed.

Due to lack of space we present in table 2 results only for the former method, as it presents IHaar more favorably and the comparison to TLHaar is more fair. Generally when the latter method is used IHaar's coefficients compress worse, to the extent that when byte arithmetic encoding is used TLHaar is always superior. Some example results in this case are TLHaar being 4.45% better in the Photos category and 7.23% better in the MRI category.

Results for TLHaar in table 2 are when using unpermuted tables, as described in section 2.1. We do not include the size of the transform LUTs in the TLHaar coefficient sizes. The tables are a static part of the transform process and are therefore known ahead of time, so in a coding application the tables do not need to be sent as part of the encoded data.

Image	TLHaar	IHaar	% Gain	TLHaar	IHaar	% Gain
	gzip			bzip		
BW_Lines	360972	466731	22.66	391737	500173	21.68
LineArt	419235	528964	20.74	445387	515182	13.55
ObjectBank	3131056	3341803	6.31	3115625	3258474	4.38
MRI	5477859	5381934	-1.78	5281869	5190000	-1.77
ccitt	1157989	1557889	25.67	1013614	1454085	30.29
DB1_B	5016509	5086675	1.38	4973728	4853424	-2.48
DB2_B	5820744	6165921	5.60	6175512	6353409	2.80
Photos	15029496	14912472	-0.78	14963474	14651157	-2.13
r2_slices	777975	801756	2.97	785564	783836	-0.22
	Binary Arithmetic			Byte Arithmetic		
BW_Lines	391472	494289	20.80	377422	496202	23.94
LineArt	509478	627262	18.78	535573	677915	21.00
ObjectBank	3290576	3474223	5.29	3620160	3771556	4.01
MRI	5691930	5627898	-1.14	5408487	5407783	-0.013
ccitt	1199500	1652339	27.41	1288297	1969556	34.59
DB1_B	5673653	5346789	-6.11	4796648	4703212	-1.99
DB2_B	6415074	6803730	5.71	5679509	5946472	4.49
Photos	15234370	14873308	-2.43	14285797	14090810	-1.38
r2_slices	828160	850696	2.65	823509	834297	1.29

Table 2: Compressed category sizes (in bytes).

In section 2.2 we described an alternative method of creating LUTs for use in TLHaar, by permuting the identity transform before sorting. We found that in some cases a permuted table may result in a better coding rate. For example, a permuted table used to transform the Photos category resulted in a compressed size of 15,011,101 bytes using binary arithmetic encoding—an improvement of 1.47% over the unpermuted table. Likewise the MRI category compressed 15% further, to 4,818,612 bytes. However, this same table when used with byte arithmetic coding compressed

the Photos to 19,912,143 bytes, a 39% degradation. MRI likewise degraded 12%.

5 Conclusions and Future Research

From our timing tests it is clear that TLHaar is consistently faster—usually much faster—than IHaar, particularly when data reordering is implemented. The only case in which TLHaar's speed is close to that of IHaar is when the images are rather small, as is the case with the BW_Lines image set. If a coding process will be using a simple wavelet transform and speed is the primary concern then TLHaar should be chosen over IHaar.

The compression results of TLHaar are not as clear as they are for execution time. It is obvious that if a wavelet transform is to be used and the data being processed is bilevel, or contains lines or hard edges (as are in the BW_Lines, LineArt, ObjectBank, ccitt, and DB2_B sets), then TLHaar is superior to IHaar. For other classes of images the results are mixed, and vary depending on the coding method used. For example, when using gzip as the compressor, the TLHaar coefficients produced for the Photos category compress 0.78% worse than the IHaar coefficients. The gap widens to 2.43% worse when binary arithmetic coding is used. Conversely for the MRI data set gzip compresses TLHaar coefficients 1.78% worse than IHaar coefficients, but using byte arithmetic encoding this gap narrows to only 0.013% worse.

We find this encouraging, as it indicates that with other coding techniques we may be able to get TLHaar coefficients to compress even more efficiently, to an amount that is better, or only slightly worse, than IHaar coefficients. For example, if the coefficients are reordered in a systematic way using a Zerotree [10], this may expose more redundancy. Our future research will involve exploring various ways of manipulating the coefficients to improve the coding rate.

Other research will center around studying the LUTs in more detail. First, we would like a better understanding of the behavior of the permuted LUTs. In some cases permuted LUTs are better than unpermuted ones, and we want a clearer understanding of the circumstances in which this is the case. Second, during the transform process each image type only touches a small percentage of the total entries in the LUT. These entries are often in clusters. It may therefore be possible to create LUTs for specific data types by optimizing only those parts of the LUT that the image touches.

6 Acknowledgements

Images in our test suite were provided by a variety of sources. In particular, the BW_Lines and ObjectBank images were obtained from Michael J. Tarr at Brown University [1]. The LineArt images are clip art downloaded through AppleWorks, from Apple Computer, Inc. Some images in the Photos category were obtained from the Signal and Image Processing Institute at the University of Southern California [5], and others from the Waterloo BragZone maintained by John Kominek [6]. MRI

data was provided by the Department of Radiology at UCSD Medical Center, and the Vision List Imagery archive [2]. Fingerprint images were obtained from the web site of the Fingerprint Verification Competition 2000 [4]. We are grateful to all those who made data available.

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Joshua Senecal's work was supported in part by a United States Department of Education Government Assistance in Areas of National Need (DOE-GAANN) grant #P200A980307.

References

[1] `http://www.cog.brown.edu/~tarr/stimuli.html`.

[2] `ftp://ftp.vislist.com/IMAGERY/MED_3D_SLICES`.

[3] `ftp://ftp.funet.fi/pub/graphics/misc/test-images/`.

[4] `http://bias.csr.unibo.it/fvc2000/download.asp`.

[5] `http://sipi.usc.edu/services/database/`.

[6] `http://links.uwaterloo.ca/bragzone.base.html`.

[7] Calderbank, Daubechies, Sweldens, and Yeo. Lossless image compression using integer to integer wavelet transforms. In *International Conference on Image Processing*, pages 596–599. IEEE Computer Society, 1997.

[8] Mirin, Cohen, Curtis, Dannevik, Dimits, Duchaineau, Eliason, Schikore, Anserson, Porter, Woodward, Shieh, and White. Very high resolution simulation of compressible turbulence on the ibm–sp system. Technical Report UCRL–JC–134237, Lawrence Livermore National Laboratory, 1999.

[9] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, second edition, 2000.

[10] Jerome Shapiro. Smart compression using the embedded zerotree wavelet (ezw) algorithm. In *Conference Record of The Twenty-Seventh Asilomar Conference on Signals, Systems and Computers*, pages 486–490. IEEE Computer Society, 1993.

[11] Mitchell Swanson and Ahmed Tewfik. A binary wavelet decomposition of binary images. *IEEE Transactions on Information Processing*, 3(12):1637–1650, Dec 1996.