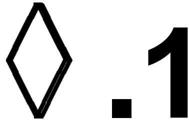

FAST CONVOLUTION WITH PACKED LOOKUP TABLES

George Wolberg
Dept. of Computer Science
City College of New York / CUNY
New York, NY 10031

Henry Massalin
Microunity Corporation
Sunnyvale, CA 94089

◊ Abstract ◊

Convolution plays a central role in many image processing applications, including image resizing, blurring, and sharpening. In all such cases, each output sample is computed to be a weighted sum of several input pixels. This is a computationally expensive operation that is subject to optimization. In this gem, we describe a novel algorithm to accelerate convolution for those applications that require the same set of filter kernel values to be applied throughout the image. The algorithm exploits some nice properties of the convolution summation for this special, but common, case to minimize the number of pixel fetches and multiply/add operations. Computational savings are realized by precomputing and packing all necessary products into lookup table fields that are then subjected to simple integer (fixed-point) shift/add operations.

◊ Introduction ◊

Discrete convolution is expressed as the following convolution summation

$$f(x) = \sum_{k=0}^{N-1} f(x_k)h(x - x_k)$$

where h is the convolution kernel weighted by N input samples $f(x_k)$. In practice, h is nearly always a symmetric kernel, i.e., $h(-x) = h(x)$. We shall assume this to be true in the discussion that follows.

The computation of one output point is illustrated in Fig. 1, where a convolution kernel is shown centered at x among the input samples. The value of that point is equal to the sum of the values of the discrete input scaled by the corresponding values of the convolution kernel. This example is appropriate for image resizing, where integer output addresses map back into real-valued input locations. For instance, output locations 0, 1, 2, 3, ... correspond to input locations 0, .5, 1, 1.5, ... upon two-fold magnification.

Copyright © 1993 by Academic Press, Inc.
All rights of reproduction in any form reserved.
ISBN 0-12-XXXXXX-X

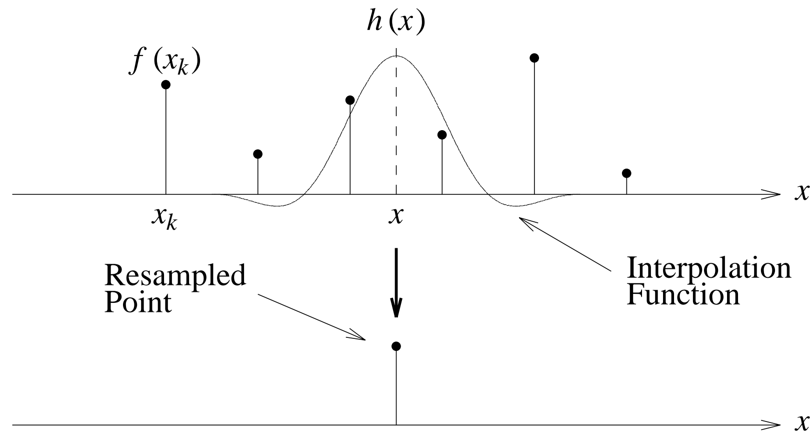

Figure 1. Convolution

In other applications, such as blurring, the image dimensions remain the same after convolution and the convolution kernel is always centered on an input sample.

There are two time-consuming phases in convolution: computing the convolution kernel weights $h(x - x_k)$ to be used, and the actual multiply/add core operations. The first problem is apparent if we consider what happens when the kernel in Fig. 1 is moved slightly. A new set of kernel values must now be applied to the input. Several researchers have looked at ways of speeding up this computation. In (Ward and Cok 1989, Wolberg 1990), a technique using coefficient bins is described that places constraints on where the kernel may be centered. By limiting the kernel to be placed at any one of, say, 64 subpixel positions, the kernel weights may be precomputed and stored in a table before the actual multiply/add operations begin. As the kernel makes its way across the input image, it must be recentered to the closest subpixel position. In (Schumacher 1992), a scaling algorithm is given which stores the computed kernel values after processing an input row, and reuses those weights for all subsequent scanlines.

This gem describes an efficient means for implementing fast convolution using lookup table operations. It assumes one important constraint: the same set of kernel values are applied throughout the image. This is appropriate for low-pass filtering (blurring), for instance, where the kernel is always centered directly on an input sample, and the same set of weights are applied to the neighbors. It is also appropriate for two-fold magnification or minification where, again, only a single set of kernel values is needed (Wolberg and Massalin 1993).

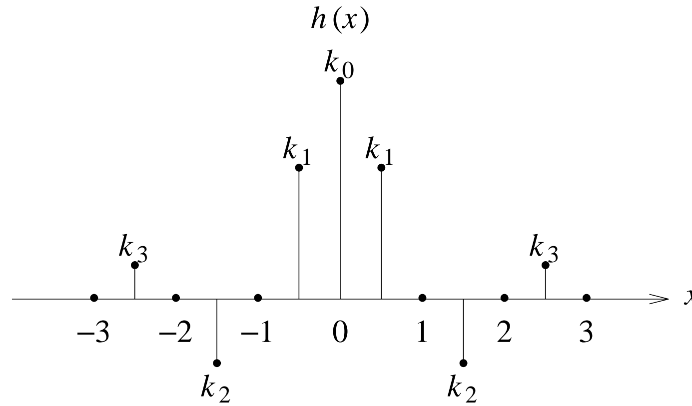

Figure 2. 6-point kernel samples

◇ A Fast Convolver for Two-fold Magnification ◇

In order to place this presentation on firm ground, we will first describe this approach in the context of two-fold image magnification. We will later demonstrate how these results are generalized to fast convolution with filter kernels of arbitrary length. This is relevant to any linear filtering operation, such as image blurring, sharpening, and edge detection. It is important to note that the algorithm is developed for the 1D case, where only rows or columns may be processed. In 2D, the image is processed separably. That is, each input row is filtered to produce an intermediate image I. Image I is then convolved along its columns to produce the final output image. For convenience, our discussion will assume that we are convolving 8-bit data with a 6-point kernel for the purpose of image magnification.

Due to symmetry, a 6-point kernel has only three unique kernel values: k_1, k_2, and k_3 (see Fig. 2). The seventh kernel value, k_0, in Fig. 2 is unused here since it sits between the input samples. It will be necessary later when we consider general linear filtering.

Since each kernel value k_i can be applied to an integer in the range [0, 255], we may precompute their products and store them in three lookup tables tab_i, for $1 \leq i \leq 3$. The product of data sample s with weight k_i now reduces to a simple lookup table access, e.g., $tab_i[s]$. This makes it possible to implement a 6-point convolver without multiplication; only lookup table and addition operations are necessary. In order to retain numerical accuracy during partial evaluations, we designate each 256-entry lookup table to be 10-bits wide. This accommodates 8-bit unsigned integers with 2-bit fractions.

The use of lookup tables to eliminate multiplication becomes unfeasible when a large number of distinct kernel values are required in the convolution summation. This is particularly true of general convolution. Fortunately, many filtering applications, including two-fold magnification, require only a few distinct kernel values. The memory

demands to support the corresponding lookup tables are very modest, i.e., $256(N/2)$ 10-bit entries for an N-point kernel, where N is even.

Further computational savings are possible by exploiting some nice properties of the convolution summation for our special two-fold rescaling problem. These properties are best understood by considering the output expressions after a 6-point kernel is applied to input samples A through H. The expanded expressions for convolution output CD, DE, and EF are given below. Note that CD refers to the output sample lying halfway between pixels C and D. The same notation applies to DE and EF.

$$\begin{aligned} CD &= k_3A + k_2B + k_1C + k_1D + k_2E + k_3F \\ DE &= k_3B + k_2C + k_1D + k_1E + k_2F + k_3G \\ EF &= k_3C + k_2D + k_1E + k_1F + k_2G + k_3H \end{aligned}$$

These results demonstrate a pattern: each input sample s is weighted by all k_i values during the course of advancing the kernel across the data. This is apparent for samples C and F in all three expressions. Rather than individually accessing each of the three tables with sample s, all three tables may be packed side-by-side into one wide table having 30 bits in width. This permits one index to access three packed products at once. The largest number of tables that may be packed together is limited only by the precision with which we store the products and the width of the longest integer, e.g., 32 bits on most computers.

Figure 3 shows table entries for input samples A through H. Three 10-bit fields are used to pack three fixed point products. Each field is shown to be involved in some convolution summation, as denoted by the arrows, to compute output CD, DE (shown shaded), and EF. The organization of the data in this manner not only reduces the number of table accesses, but it also lends itself to a fast convolution algorithm requiring only shift and add operations. The downward (upward) arrows denote a sequence of right-shift (left-shift) and addition operations, beginning with the table entry for A (D). Let *fwd* and *rev* be two integers that store both sets of results. The first few shift-add operations produce *fwd* and *rev* with the fields shown in Table 1.

Notice that the low-order 10-bit fields of *fwd* contain half of the convolution summa-

Table 1. 10-bit Fields in *fwd* and *rev*

fwd			*rev*		
bits 20-29	bits 10-19	bits 0-9	bits 20-29	bits 10-19	bits 0-9
k_3B	$k_3A + k_2B$	$k_2A + k_1B$	$k_3E + k_2D$	$k_2E + k_1D$	k_1E
k_3C	$k_3B + k_2C$	$k_3A + k_2B + k_1C$	$k_3F + k_2E + k_1D$	$k_2F + k_1E$	k_1F
k_3D	$k_3C + k_2D$	$k_3B + k_2C + k_1D$	$k_3G + k_2F + k_1E$	$k_2G + k_1F$	k_1G
k_3E	$k_3D + k_2E$	$k_3C + k_2D + k_1E$	$k_3H + k_2G + k_1F$	$k_2H + k_1G$	k_1H

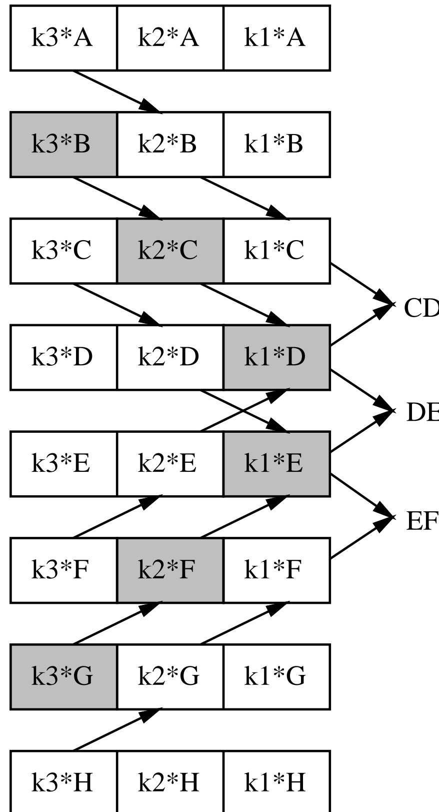

Figure 3. A fast convolver

tion necessary to compute the output. The other half is contained in the high-order 10-bit fields of *rev*. Simply adding both fields together generates the output values.

This scheme is hampered by one complication: addition may cause one field to spill into the next, thereby corrupting its value. This will happen if a field value exceeds the range $[0, 2^8 - 1]$. Note that although we use 10-bit fields, the integer part is 8 bits wide. We now consider the two range limits on field value $v = k_3A + k_2B + k_1C$.

We may guard against negative values by simply adding a bias to the field of all negative kernel values k_i. After rescaling the bias to a 10-bit quantity, it is simply $1024k_i$. Simultaneously, we must guarantee that the sum of the three kernel values and the biases is less than unity to ensure that the upper limit is satisfied. Note that the bias is removed from the computation when we add the low-order 10-bit field of *fwd* to the high-order 10-bit field of *rev*.

The following fragment of C code demonstrates the initialization of the packed lookup table *lut*, consisting of 256 32-bit integers.

```
#define MASK 0x3FF
#define ROUND 1
#define PACK(A,B,C) (((A)<<20) + ((B)<<10) + (C))
#define INT(A) ((int) ((A)*262144+32768) >> 16)

b1 = b2 = b3 = 0;
if(k1 < 0) b1 = -k1 * 1024;
if(k2 < 0) b2 = -k2 * 1024;
if(k3 < 0) b3 = -k3 * 1024;
bias = 2 * (b1 + b2 + b3);

for(i=0; i<256; i++)
lut[i] = PACK( INT(i*k3)+b3, INT(i*k2)+b2+ROUND, INT(i*k1)+b1 );
```

The *INT* macro converts the real-valued kernel samples into 10-bit fixed point quantities. Notice that since the macro argument A has an 8-bit magnitude, we form an intermediate 26-bit result by multiplying A by 1<<18. Roundoff is achieved by adding 1<<15 (or .5) before right-shifting by 16 bits to obtain the final 10-bit number. A bias is added to each field to prevent negative numbers and the undesirable sign extension that would corrupt its neighbors. *ROUND* is necessary to avoid roundoff error when adding the *fwd* and *rev* terms together to compute the output. Adding *ROUND* directly in *lut* spares us from having to explicitly add it at every output computation.

Once *lut* is initialized, it is used in the following code to realize fast convolution. The variable *len* refers to the number of input samples, and *ip* and *op* are input and output pointers that reference the padded working buffer *buf*. We assume that the input samples have already been copied into the even addresses of *buf* in order to trivially compute half of the two-fold magnification output. Since we are now using a 6-point kernel, the left padding occupies positions 0 and 2, the first input sample lies in position 4, and the first output sample will lie in position 5.

```
/* clamp definition: clamp A into the range [L,H] */
#define CLAMP(A,L,H) ((A)<=(L) ? (L) : (A)<=(H) ? (A) : (H))

/* initialize input and output pointers, ip and op, respectively */
ip = &buf[0];
op = &buf[5];

fwd = (lut[ip[0]] >> 10) + lut[ip[2]];
rev = (lut[ip[6]] << 10) + lut[ip[8]];
ip += 4;

while(len--) {
        fwd = (fwd >> 10) + lut[ip[0]];
```

```
        rev = (rev << 10) + lut[ip[6]];
        val = ((fwd & MASK) + ((rev >> 20) & MASK) - bias) >> 2;
        *op = CLAMP(val, 0, 255);

        /* input and output strides are 2 */
        ip += 2;
        op += 2;
}
```

The bias terms find their way into the low-order and high-order fields of *fwd* and *rev* through the sequence of shift-add operations. Since these two fields are used to compute val, we must subtract twice the sum of the bias terms from val to restore its proper value. Recall that $bias$ was defined in the previous code fragment. We then discard the fractional part of val by a two-bit right shift, leaving us with an 8-bit integer. Since val may now be negative, it is necessary to clamp it into the range [0, 255].

Operating with symmetric kernels has already been shown to reduce the number of arithmetic operations: $N/2$ multiplies and $N-1$ for an N-point kernel, where N is even. This algorithm, however, does far better. It requires no multiplication (other than that needed to initialize lut), and a total of four adds per output sample, for a 6-point kernel. Furthermore, no distinction is made between 2-, 4-, and 6-point kernels because they are all packed into the same integer. That is, a 4-point kernel is actually implemented as a 6-point kernel with $k_3 = 0$. Since there is no additional penalty for using a 6-point kernel, we are induced to use a superior (6-point) filter at low cost. Larger kernels can be assembled by cascading additional 6-point kernel stages together (see the supplied code).

◊ General Convolution ◊

The C code provided with this gem demonstrates the use of the fast convolver for general linear filtering. This is essentially the same technique as for the two-fold magnifier shown earlier. There is one important difference though: the N-point kernels are centered on input samples and so N must be odd. The extra kernel value corresponds to the center pixel, that must now be weighted by k_0. Although this can be explicitly added to the weighted sum of the neighboring six pixels, the number of addition operations to compute an output pixel would rise by one. For instance, the following C statement could be used:

```
val = ((fwd & MASK) + ((rev >> 20) & MASK) - bias + lut0[*ip]) >> 2;
```

where ip points to the center pixel that is used to index $lut0$, a lookup table storing the product of the pixel with kernel sample k_0. Note that a total of 5 additions are needed to compute a 7-point kernel: 1 each for *fwd* and *rev*, and 3 for val. We will refer to this

Table 2. Comparison of Operation Counts

N	Method 1	Method 2
3	5	4
5	5	4
7	5	8
9	9	8
11	9	8
13	9	12
15	13	12

approach as Method 1. In order to reduce one addition, we can embed that weighting directly in the packed lookup tables by halving k_0 and permitting it to be applied on the center pixel in both *fwd* and *rev* for the purpose of adding the contribution of that pixel (Method 2). This, however, reduces the extent of the kernel by one. Table 2 compares the number of additions needed to compute an output value using the two methods for various values of N in an N-point kernel.

It is important to note that multiple instances of *fwd* and *rev* are needed when they are cascaded to realize wider kernels. This explains why the number of additions above rises by increments of four: two to compute a new pair of *fwd* and *rev* terms, and two to add them to *val*. Method 2 is generally more efficient than Method 1, except in instances when the overhead cost of adding an additional *fwd* and *rev* pair sets in. The supplied code implements Method 2.

◊ Summary and Conclusions ◊

In summary, this gem has focused on optimizing the evaluation of the convolution summation. We achieve large performance gains by packing all weighted instances of an input sample into one 32-bit integer and then using these integers in a series of shift-add operations to compute the output. The algorithm benefits from a technique well known in the folklore of assembler programmers and microcoders: multiple integers can be added in parallel in a single word if their bit fields do not overlap. This alone, however, is not the basis of the algorithm. Rather, the novelty of the algorithm lies in identifying a particularly efficient structure for the fast convolver that can exploit this straightforward technique for parallel addition and apply it to kernels of arbitrary length. An additional feature of the fast convolver is that it requires each pixel to be fetched only once, eventhough it is used in the computation of several output pixels.

The sequence of shift-add operations essentially mimics a pipelined vector processor on a general 32-bit computer. This approach will likely find increased use with the forthcoming generation of 64-bit computers. The additional bits will permit us to handle wider kernels at finer precision.

◇ Acknowledgements ◇

This work was supported in part by the NSF (PYI Award IRI-9157260), PSC-CUNY (RF-664314), and the Xerox Foundation.

◇ C Code ◇

The following C code implements the fast convolution algorithm. The program takes three arguments: filenames for the input image, kernel, and output image. Utility functions are provided to read and write 8-bit grayscale images. The image format used is simple: two integers specifying the width and height followed by a stream of 8-bit unsigned pixel data. The kernel is stored in an ASCII file containing one kernel value per row beginning with k_0. For instance, the following kernel file contains a 7-point low-pass filter.

```
.3333333333333333333
.2395833333333333333
.0833333333333333333
.0104166666666666666
```

The program handles up to 17-point kernels. Extending this to handle larger kernels is a simple matter of accommodating additional stages in the *lutS* data structure and *fastconv*() function.

Execution time on a SUN 4/50 (IPX) workstation was measured on the repeated calls to *fastconv*() in function *convolve*(). Convolution of a 256×256 image with a 7-point and 17-point kernel took .75 seconds and .85 seconds, respectively. The same convolution took 1.75 seconds and 3.96 seconds, respectively, when implemented with standard multiply/add operations. Due to the separable implementation, execution time grows linearly with filter width.

```
/* ======================================================================
 *
 *      Fast Convolution With Packed Lookup Tables
 *
 *         by George Wolberg and Henry Massalin
 *
 *      Compile: cc convolve.c -o convolve
 *      Execute: convolve in.bw kernel out.bw
 * ======================================================================
 */

#include <stdio.h>
#include <stdlib.h>

typedef unsigned char   uchar;
```

```
typedef struct {                /* image data structure  */
        int width;              /* image width  (# cols) */
        int height;             /* image height (# rows) */
        uchar *image;           /* pointer to image data */
} imageS, *imageP;

typedef struct {                /* packed lut structure     */
        int lut0[256];          /* stage 0 for  5-pt kernel */
        int lut1[256];          /* stage 1 for 11-pt kernel */
        int lut2[256];          /* stage 2 for 17-pt kernel */
        int bias;               /* accumulated stage biases */
        int stages;             /* # of stages used: 1,2,3  */
} lutS, *lutP;

/* definitions */
#define MASK            0x3FF
#define ROUNDD          1
#define PACK(A,B,C)     (((A)<<20) + ((B)<<10) + (C))
#define INT(A)          ((int) ((A)*262144+32768) >> 16)
#define CLAMP(A,L,H)    ((A)<=(L) ? (L) : (A)<=(H) ? (A) : (H))
#define ABS(A)          ((A) >= 0 ? (A) : -(A))

/* declarations for convolution functions */
void    convolve();
void    initPackedLuts();
void    fastconv();

/* declarations for image utility functions */
imageP  allocImage();
imageP  readImage();
int     saveImage();
void    freeImage();

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * main:
 *
 * Main function to collect input image and kernel values.
 * Pass them to convolve() and save result in output file.
 */
main(argc, argv)
int       argc;
char    **argv;
{
        int     n;
        imageP  I1, I2;
        float   kernel[9];
        char    buf[80];
        FILE    *fp;

        /* make sure the user invokes this program properly */
        if(argc != 4) {
                fprintf(stderr, "Usage: convolve in.bw kernel out.bw\n");
                exit(1);
```

```
        }

        /* read input image */
        if((I1=readImage(argv[1])) == NULL) {
                fprintf(stderr, "Can't read input file %s\n", argv[1]);
                exit(1);
        }

        /* read kernel: n lines in file specify a (2n-1)-point kernel
         * Don't exceed 9 kernel values (17-point symmetric kernel is limit)
         */
        if((fp=fopen(argv[2], "r")) == NULL) {
                fprintf(stderr, "Can't read kernel file %s\n", argv[2]);
                exit(1);
        }
        for(n=0; n<9 && fgets(buf, 80, fp); n++) kernel[n] = atof(buf);

        /* convolve input I1 with fast convolver */
        I2 = allocImage(I1->width, I1->height);
        convolve(I1, kernel, n, I2);

        /* save output to a file */
        if(saveImage(I2, argv[3]) == NULL) {
                fprintf(stderr, "Can't save output file %s\n", argv[3]);
                exit(1);
        }
}



/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * convolve:
 *
 * Convolve input image I1 with kernel, a (2n-1)-point symmetric filter
 * kernel containing n entries: h[i] = kernel[ |i| ] for -n < i < n.
 * Output is stored in I2.
 */
void
convolve(I1, kernel, n, I2)
imageP   I1, I2;
float   *kernel;
int      n;
{
        int     x, y, w, h;
        uchar   *src, *dst;
        imageP  II;
        lutS    luts;

        w = I1->width;                          /* image width          */
        h = I1->height;                         /* image height         */

        II = allocImage(w, h);                  /* reserve tmp image    */
        initPackedLuts(kernel, n, &luts);       /* init packed luts     */
```

```
        for(y=0; y<h; y++) {                    /* process all rows     */
                src = I1->image + y*w;          /* ptr to input  row    */
                dst = II->image + y*w;          /* ptr to output row    */
                fastconv(src, w, 1, &luts, dst);/* w pixels; stride=1   */
        }

        for(x=0; x<w; x++) {                    /* process all columns  */
                src = II->image + x;            /* ptr to input  column */
                dst = I2->image + x;            /* ptr to output column */
                fastconv(src, h, w, &luts, dst);/* h pixels; stride=w   */
        }

        freeImage(II);                          /* free temporary image */
}


/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * initPackedLuts:
 *
 * Initialize scaled and packed lookup tables in lut.
 * Permit up to 3 cascaded stages for the following kernel sizes:
 *      stage 0:  5-point kernel
 *      stage 1: 11-point kernel
 *      stage 2: 17-point kernel
 * lut->lut0 <== packed entries (i*k2, i*k1, .5*i*k0), for i in [0, 255]
 * lut->lut1 <== packed entries (i*k5, i*k4,    i*k3), for i in [0, 255]
 * lut->lut2 <== packed entries (i*k8, i*k7,    i*k6), for i in [0, 255]
 * where k0,...k8 are taken in sequence from kernel[].
 *
 * Note that in lut0, k0 is halved since it corresponds to the center
 * pixel's kernel value and it appears in both fwd0 and rev0 (see gem).
 */
static void
initPackedLuts(kernel, n, luts)
float   *kernel;
int      n;
lutP     luts;
{
        int     i, k, s, *lut;
        int     b1, b2, b3;
        float   k1, k2, k3;
        float   sum;

        /* enforce flat-field response constraint: sum of kernel values = 1 */
        sum = kernel[0];
        for(i=1; i<n; i++) sum += 2*kernel[i];  /* account for symmetry */
        if(ABS(sum - 1) > .001)
                fprintf(stderr, "Warning: filter sum != 1 (=%f)\n", sum);

        /* init bias added to fields to avoid negative numbers (underflow) */
        luts->bias = 0;
```

```
	/* set up lut stages, 3 kernel values at a time */
	for(k=s=0; k<n; s++) {                  /* init lut (stage s)   */
		k1 = (k < n) ? kernel[k++] : 0;
		k2 = (k < n) ? kernel[k++] : 0;
		k3 = (k < n) ? kernel[k++] : 0;
		if(k <= 3) k1 *= .5;            /* kernel[0]: halve k0  */

		/* select proper array in lut structure based on stage s */
		switch(s) {
		case 0: lut = luts->lut0;       break;
		case 1: lut = luts->lut1;       break;
		case 2: lut = luts->lut2;       break;
		}

		/* check k1,k2,k3 to avoid overflow in 10-bit fields */
		if(ABS(k1) + ABS(k2) + ABS(k3) > 1) {
			fprintf(stderr, "|%f|+|%f|+|%f| > 1\n", k1, k2, k3);
			exit(1);
		}

		/* compute bias for each field to avoid underflow */
		b1 = b2 = b3 = 0;
		if(k1 < 0) b1 = -k1 * 1024;
		if(k2 < 0) b2 = -k2 * 1024;
		if(k3 < 0) b3 = -k3 * 1024;

		/* luts->bias will be subtracted in convolve() after adding
		 * stages; multiply by 2 because of combined effect of fwd
		 * and rev terms
		 */
		luts->bias += 2*(b1 + b2 + b3);

		/* scale and pack kernel values in lut */
		for(i=0; i<256; i++) {
			/*
			 * INT(A) forms fixed point field:
			 * (A*(1<<18)+(1<<15)) >> 16
			 */
			lut[i] = PACK(  INT(i*k3) + b3,
					INT(i*k2) + b2 + ROUNDD,
					INT(i*k1) + b1 );
		}
	}
	luts->stages = s;
}


/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * fastconv:
 *
 * Fast 1D convolver.
```

```
 * Convolve len input samples in src with a symmetric kernel packed in luts,
 * a lookup table that is created by initPackedLuts() from kernel values.
 * The output goes into dst.
 */
static void
fastconv(src, len, offst, luts, dst)
int     len, offst;
uchar   *src, *dst;
lutP    luts;
{
        int      x, padlen, val, bias;
        int      fwd0, fwd1, fwd2;
        int      rev0, rev1, rev2;
        int     *lut0, *lut1, *lut2;
        uchar   *p1, *p2, *ip, *op;
        uchar    buf[1024];

        /* copy and pad src into buf with padlen elements on each end */
        padlen = 3*(luts->stages) - 1;
        p1 = src;               /* pointer to row (or column) of input  */
        p2 = buf;               /* pointer to row of padded buffer      */
        for(x=0; x<padlen; x++) /* pad left side: replicate first pixel */
                *p2++ = *p1;
        for(x=0; x<len; x++) {  /* copy input row (or column)           */
                *p2++ = *p1;
                 p1  +=  offst;
        }
        p1 -= offst;            /* point to last valid input pixel      */
        for(x=0; x<padlen; x++) /* pad right side: replicate last pixel */
                *p2++ = *p1;

        /* initialize input and output pointers, ip and op, respectively */
        ip = buf;
        op = dst;

        /* bias was added to lut entries to deal with negative kernel values */
        bias = luts->bias;

        switch(luts->stages) {
        case 1:         /* 5-pt kernel */
                lut0 = luts->lut0;

                ip  += 2;       /* ip[0] is center pixel */
                fwd0 = (lut0[ip[-2]] >> 10) + lut0[ip[-1]];
                rev0 = (lut0[ip[ 0]] << 10) + lut0[ip[ 1]];

                while(len--) {
                        fwd0 = (fwd0 >> 10) + lut0[ip[0]];
                        rev0 = (rev0 << 10) + lut0[ip[2]];
                        val = ((fwd0 & MASK) + ((rev0 >> 20) & MASK) - bias)
                                >> 2;
                        *op = CLAMP(val, 0, 255);
```

```
                ip++;
                op += offst;
        }
        break;
case 2:         /* 11-pt kernel */
        lut0 = luts->lut0;
        lut1 = luts->lut1;

        ip  += 5;       /* ip[0] is center pixel */
        fwd0 = (lut0[ip[-2]] >> 10) + lut0[ip[-1]];
        rev0 = (lut0[ip[ 0]] << 10) + lut0[ip[ 1]];

        fwd1 = (lut1[ip[-5]] >> 10) + lut1[ip[-4]];
        rev1 = (lut1[ip[ 3]] << 10) + lut1[ip[ 4]];

        while(len--) {
                fwd0 = (fwd0 >> 10) + lut0[ip[0]];
                rev0 = (rev0 << 10) + lut0[ip[2]];

                fwd1 = (fwd1 >> 10) + lut1[ip[-3]];
                rev1 = (rev1 << 10) + lut1[ip[ 5]];

                val  =  ((fwd0 & MASK) + ((rev0 >> 20) & MASK)
                        +(fwd1 & MASK) + ((rev1 >> 20) & MASK) - bias)
                        >> 2;
                *op = CLAMP(val, 0, 255);

                ip++;
                op += offst;
        }
        break;
case 3:         /* 17-pt kernel */
        lut0 = luts->lut0;
        lut1 = luts->lut1;
        lut2 = luts->lut2;

        ip  += 8;       /* ip[0] is center pixel */
        fwd0 = (lut0[ip[-2]] >> 10) + lut0[ip[-1]];
        rev0 = (lut0[ip[ 0]] << 10) + lut0[ip[ 1]];

        fwd1 = (lut1[ip[-5]] >> 10) + lut1[ip[-4]];
        rev1 = (lut1[ip[ 3]] << 10) + lut1[ip[ 4]];

        fwd2 = (lut2[ip[-8]] >> 10) + lut2[ip[-7]];
        rev2 = (lut2[ip[ 6]] << 10) + lut2[ip[ 7]];

        while(len--) {
                fwd0 = (fwd0 >> 10) + lut0[ip[0]];
                rev0 = (rev0 << 10) + lut0[ip[2]];

                fwd1 = (fwd1 >> 10) + lut1[ip[-3]];
                rev1 = (rev1 << 10) + lut1[ip[ 5]];
```

```
                        fwd2 = (fwd2 >> 10) + lut2[ip[-6]];
                        rev2 = (rev2 << 10) + lut2[ip[ 8]];

                        val  =  ((fwd0 & MASK) + ((rev0 >> 20) & MASK)
                                +(fwd1 & MASK) + ((rev1 >> 20) & MASK)
                                +(fwd2 & MASK) + ((rev2 >> 20) & MASK) - bias)
                                >> 2;
                        *op = CLAMP(val, 0, 255);

                        ip++;
                        op += offst;
                }
                break;
        }
}


/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * readImage:
 *
 * Read an image from file.
 * Format: two integers to specify width and height, followed by uchar data.
 * Return image structure pointer.
 */
imageP
readImage(file)
char    *file;
{
        int      sz[2];
        FILE    *fp;
        imageP   I = NULL;

        /* open file for reading */
        if((fp = fopen(file, "r")) != NULL) {   /* open file for read   */
                fread(sz, sizeof(int), 2, fp);  /* read image dimensions*/
                I = allocImage( sz[0],sz[1]);   /* init image structure */
                fread(I->image, sz[0],sz[1],fp);/* read data into I     */
                fclose(fp);                     /* close image file     */
        }
        return(I);                              /* return image pointer */
}


/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * saveImage:
 *
 * Save image I into file.
 * Return NULL for failure, 1 for success.
 */
int
saveImage(I, file)
```

```
imageP   I;
char    *file;
{
        int      sz[2], status = NULL;
        FILE    *fp;

        if((fp = fopen(file, "w")) != NULL) {   /* open file for save   */
                sz[0] = I->width;
                sz[1] = I->height;
                fwrite(sz, sizeof(int), 2, fp); /* write dimensions     */
                fwrite(I->image,sz[0],sz[1],fp);/* write image data     */
                fclose(fp);                     /* close image file     */
                status = 1;                     /* register success     */
        }
        return(status);
}


/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * allocImage:
 *
 * Allocate space for an uchar image of width w and height h.
 * Return image structure pointer.
 */
imageP
allocImage(w, h)
int w, h;
{
        imageP   I;

        /* allocate memory for image data structure */
        if((I = (imageP) malloc(sizeof(imageS))) != NULL) {
                I->width  = w;                  /* init width           */
                I->height = h;                  /* init height          */
                I->image  =(uchar*) malloc(w*h);/* init data pointer    */
        }
        return(I);                              /* return image pointer */
}


/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * freeImage:
 *
 * Free image memory.
 */
void
freeImage(I)
imageP I;
{
        free((char *) I->image);
        free((char *) I);
```

}

Bibliography

(Schumacher 1992) Dale Schumacher. *Graphics Gems III*, chapter General Filtered Image Rescaling. Academic Press, Boston, 1992.

(Ward and Cok 1989) Joseph Ward and David R. Cok. Resampling algorithms for image resizing and rotation. In *Proc. SPIE Digital Image Processing Applications*, volume 1075, pages 260–269, 1989.

(Wolberg and Massalin 1993) George Wolberg and Henry Massalin. A fast algorithm for digital image scaling. In *Proc. Computer Graphics International*, 1993.

(Wolberg 1990) George Wolberg. *Digital Image Warping.* IEEE Computer Society Press, Los Alamitos, CA, 1990.

Copyright © 1993 by Academic Press, Inc.
All rights of reproduction in any form reserved.
ISBN 0-12-XXXXXX-X