A faster way to downscale during JPEG decoding to a fourth

written by written by Stefan Kuhr [1]

Introduction

The algorithm that is employed in the JPEGLib for downscaling to a fourth during decoding uses the following matrix-vector product, which is derived from the full IDCT from Loeffler, Ligtenberg and Moschytz:

$$\frac{1}{4}\begin{pmatrix} f(0)+f(1)+f(2)+f(3) \\ f(4)+f(5)+f(6)+f(7) \end{pmatrix} = \frac{1}{4\sqrt{2}} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & C_7 - C_5 & C_7 - C_3 & C_1 - C_5 & C_1 + C_3 \end{pmatrix} \cdot \begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(7) \\ \tilde{F}(3) \\ \tilde{F}(5) \\ \tilde{F}(1) \end{pmatrix} \quad (1)$$

In this matrix-vector product, $\tilde{F}(n)$ is the DCT coefficient at index n and $f(n)$ is the pixel value in the spatial domain at position n. The constants C_n that are used are defined as follows:

$$C_k = cos\frac{k\pi}{16}, \quad k = 0, \ldots, 7 \quad (2)$$

The structure of the algorithm can probably better be explained as a flowgraph as in figure 1. What can be found surprising in this structure is the fact, that all multiplicative constants scale the DCT coefficients, much like the scaling coefficients of the Arai-Agui-Nakajima DCT. The JPEGLib accounts for the Arai-Agui-Nakajima DCT's coefficients in that these are absorbed into the dequantization coefficients. This is what makes the Arai-Agui-Nakajima DCT so fast in comparison to other schemes. Unfortunately, the JPEGLib does not use this approach for the algorithm that scales to a fourth. If this were done

[1] The author is a software developer in Germany who had the opportunity to implement a JPEG decoder on an embedded system with a 16-bit microcontroller during his master thesis. The information in this article is published with the expressed consent of the author's company supervisor during the thesis. The author can be contacted under the email address list@craalse.de

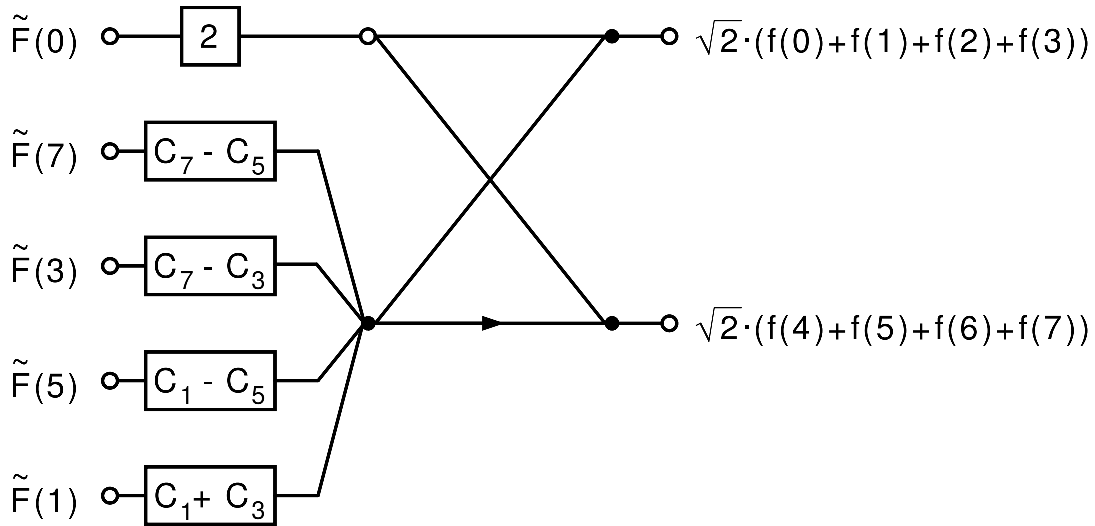

Figure 1: Flowgraph for the IDCT resizing to a fourth of the original size

in the JPEGLib, scaling to a fourth would in theory simply consist out of additions. The author made an implementation for this inside the JPEGLib and it actually turned out that decoding is much faster this way since now only additions and shift operations are involved. Unfortunately, it also turned out that there are dependencies on the bitness of the platform being used. The scaling constants over both dimensions when represented as real numbers look like the following:

```
4.000000, 3.624510, 2.000000, 1.272759, 2.000000, 0.850430, 2.000000, 0.720960,
3.624510, 3.284268, 1.812255, 1.153281, 1.812255, 0.770598, 1.812255, 0.653281,
2.000000, 1.812255, 1.000000, 0.636379, 1.000000, 0.425215, 1.000000, 0.360480,
1.272759, 1.153281, 0.636379, 0.404979, 0.636379, 0.270598, 0.636379, 0.229402,
2.000000, 1.812255, 1.000000, 0.636379, 1.000000, 0.425215, 1.000000, 0.360480,
0.850430, 0.770598, 0.425215, 0.270598, 0.425215, 0.180808, 0.425215, 0.153281,
2.000000, 1.812255, 1.000000, 0.636379, 1.000000, 0.425215, 1.000000, 0.360480,
0.720960, 0.653281, 0.360480, 0.229402, 0.360480, 0.153281, 0.360480, 0.129946
```

With this table, first all dequantization table values must be scaled and afterwards two passes over the dequantized DCT coefficients have to be made which in theory only consist out of additions: In the first pass we perform the algorithm over the columns 0, 1, 3, 5 and 7, each time doing the additions like in the flowgraph. This yields two values per each of these 5 columns and thus two rows with 5 columns of interest. The second pass performs only the additions like in the flowgraph, but this time over the two rows that hold the result of the first pass. In order to implement this table with fixed-point arithmetics, careful observation of potential overflow when scaling the quantization table and when adding and subtracting is required. This leads to six different tables in this new implementation that arise from dependencies on the platform's bitness, the size of a sample (8 bits or 12 bits)

and also allow a speed versus accuracy tradeoff. In order to implement the functionality, the new macro USE_FASTER_2x2_IDCT was introduced which simply needs to be defined in jconfig.h or jmorecfg.h like this:

```
#define USE_FASTER_2x2_IDCT
```

If this macro is not defined, the standard functionality from JPEGLib version 6b is used. Depending on this macro, additional code in jddctmgr.c multiplies the dequantization constants with a scaled variant of the table above and in jidctred.c an alternative implementation of jpeg_idct_2x2 gets compiled. The dependence on the bitness of the platform is automatically resolved by examining the value of the constant INT_MAX from limits.h. If INT_MAX evaluates to 32767, it is a 16-bit platform, otherwise a 32-bit platform or higher is expected.

In order to use the fastest possible mode (with the least accuracy), additionally the macro USE_INACCURATE_IDCT can be defined in jconfig.h or jmorecfg.h like this:

```
#define USE_INACCURATE_IDCT
```

Depending on the platform's bitness, this macro has different functionality:

- 16-bit: If USE_INACCURATE_IDCT is defined, all multiplications and additions yield results that are less than 2^{15}, so only the (16-bit) int data type is used. Nevertheless, in this "mode", a right shift operation is required after the first pass in order to avoid a potential overflow in the second pass.
- 32-bit: If USE_INACCURATE_IDCT is defined, the table is chosen in such a way, that no right shift operation after the first pass is required. For this "mode" the table that is used must not lead to an overflow in the second pass and is therefore less accurate than with USE_INACCURATE_IDCT undefined

If USE_INACCURATE_IDCT is undefined, the tables are chosen in such a way, that no overflow happens during the scaling of the dequantization tables and during the additions in both passes of the algorithm as will be shown later.

The tables and potential overflow

In this section we will look at the tables and show how they were chosen in order to avoid overflow. We will first examine the tables that were used for 32-bit platforms and then the corresponding tables for 16-bit platforms. In order to estimate the potential overflow we will first consider the case of 8-bit sample values. In this case the JPEG Standard states in Annex F.1.1.4: "The quantized DCT coefficient values are signed, twos complement integers with 11-bit precision for 8-bit input precision...". This simply means that the DCT coefficients after dequantization have a range of 10000000000. . . 11111111111, 00000000000, 00000000001, . . . 01111111111 or in decimals: -1024 . . . -1, 0, 1, . . . 1023. In the following we will use 1024 for worst case scenarios as the maximum input value and will simply disregard the sign of the constants that need to be absorbed into the dequantization table values, while keeping in mind that the dequantization table values have a range of 0 . . . 255.

The tables for 32-bit platforms

If USE_INACCURATE_IDCT is not defined on a 32-bit platform, the following fixed-point constants are used to implement the table of constants that are absorbed into the dequantization table:

```
524288, 475072, 262144, 166823, 262144, 111468, 262144, 94498,
475072, 430476, 237536, 151163, 237536, 101004, 237536, 85627,
262144, 237536, 131072, 83412, 131072, 55734, 131072, 47249,
166823, 151163, 83412, 53081, 83412, 35468, 83412, 30068,
262144, 237536, 131072, 83412, 131072, 55734, 131072, 47249,
111468, 101004, 55734, 35468, 55734, 23699, 55734, 20091,
262144, 237536, 131072, 83412, 131072, 55734, 131072, 47249,
94498, 85627, 47249, 30068, 47249, 20091, 47249, 17032
```

This table is simply the table with the real values presented before, but scaled with $131072 = 2^{17}$. In order to estimate the possibility of an overflow, we only have to observe the values in the first row (or column) since these are the highest values and thus are most likely to lead to an overflow. In the first row, only the values at odd indices and the value at index 0 are of any interest of us, because the algorithm makes no usage of all other values (therefore they are skipped during absorbing them into the dequantization constants).

The worst case during the first pass happens in the addition of the scaled value of $\tilde{F}(0)$ with the result of the addition of all other scaled values. This means that $524288+475072+166823+111468+94498$ times the maximum value of 1024 must be less than 2^{31}. In fact, $(524288 + 475072 + 166823 + 111468 + 94498) \times 1024 = 1405080576 < 2^{31} = 2147483648$. In order to avoid an overflow by adding 5 times this maximum value of 1405080576 in the second pass, it now is scaled by a right shift over 2 digits, which yields a maximum value after the first pass of $1405080576/4 = 351270144$. If this value is added five times in the second pass, the final result of the second result prior to the final downscaling operation is $5 \times 351270144 = 1756350720$ which is again less than 2^{31}. After the final downscaling over 20 digits the pixel values in the spatial domain are obtained.

If USE_INACCURATE_IDCT is defined on a 32-bit platform, the following fixed-point constants are used to implement the table of constants that are absorbed into the dequantization table:

```
131072, 118768, 65536, 41706, 65536, 27867, 65536, 23624,
118768, 107619, 59384, 37791, 59384, 25251, 59384, 21407,
65536, 59384, 32768, 20853, 32768, 13933, 32768, 11812,
41706, 37791, 20853, 13270, 20853, 8867, 20853, 7517,
65536, 59384, 32768, 20853, 32768, 13933, 32768, 11812,
27867, 25251, 13933, 8867, 13933, 5925, 13933, 5023,
65536, 59384, 32768, 20853, 32768, 13933, 32768, 11812,
23624, 21407, 11812, 7517, 11812, 5023, 11812, 4258
```

Again, this table is simply the table with the real values presented before, but this time scaled with $131072 = 2^{15}$. By choosing a table that is less accurate by two digits than the one before, we can omit the right shift over 2 digits after the first pass, resulting in a minimum of operations. This variant of the algorithm really requires only additions.

The tables for 16-bit platforms

For 16-bit platforms we have to make sure that any operation's result is less than 2^{15} if the int data type is used and 2^{31} if the INT32 data type is used for storing the result. If USE_INACCURATE_IDCT is not defined on a 16-bit platform, the intermediate results of the multiplications are kept in INT32 variables, but are scaled to int values after the first pass. Additionally, the following fixed-point constants are used to implement the table of constants that are absorbed into the dequantization table:

```
128, 116, 64, 41, 64, 27, 64, 23,
116, 105, 58, 37, 58, 25, 58, 21,
64, 58, 32, 20, 32, 14, 32, 12,
41, 37, 20, 13, 20, 9, 20, 7,
64, 58, 32, 20, 32, 14, 32, 12,
27, 25, 14, 9, 14, 6, 14, 5,
64, 58, 32, 20, 32, 14, 32, 12,
23, 21, 12, 7, 12, 5, 12, 4
```

The problem here is that the quantization tables are implemented as an int array, therefore the maximum value of this table (128) times the maximum value in a dequantization table (255) may not exceed $2^{15} - 1$. This can for maximum accuracy only be done with this table, since $128 \times 255 = 32640 < 2^{15} = 32768$. Therefore this table must be chosen which is simply the table with the real values presented before, but this time scaled with $32 = 2^5$. If as in this case INT32 variables are used for intermediate results in the first pass, no overflow will happen, since $(128 + 116 + 41 + 27 + 23) \times 1024 = 343040 < 2^{31}$. But in order to store this result in a (16-bit) int variable that can be added to itself 5 times during the second pass, this value needs to be scaled by a right shift over 6 digits (= division by $2^6 = 64$). This leads to a maximum value of $343040/64 = 5360$. In the second pass, if this value is added 5 times to itself, this yields a maximum of $5 \times 5360 = 26800 < 2^{15}$. Finally, the second pass scales the final result by a right shift over 4 digits to obtain the pixel values in the spatial domain.

If USE_INACCURATE_IDCT is defined on a 16-bit platform, the following fixed-point constants are used to implement the table of constants that are absorbed into the dequantization table:

```
8, 7, 4, 3, 4, 2, 4, 1,
7, 7, 4, 2, 4, 2, 4, 1,
4, 4, 2, 1, 2, 1, 2, 1,
3, 2, 1, 1, 1, 1, 1, 0,
4, 4, 2, 1, 2, 1, 2, 1,
2, 2, 1, 1, 1, 0, 1, 0,
4, 4, 2, 1, 2, 1, 2, 1,
1, 1, 1, 0, 1, 0, 1, 0
```

Again, this table is simply the table with the real values presented before, but this time scaled with $2 = 2^1$. This table allows all intermediate variables in both passes to be (16-bit) int variables: In the first pass, $(8 + 7 + 3 + 2 + 1) \times 1024 = 21504$ does not exceed $2^{15} - 1$. In order for the second pass not to lead to an overflow this value needs to be scaled by a right shift over 2 digits to be added to itself 5 times in the second pass: $21504/4 = 5376$ and $5 \times 5376 = 26880 < 2^{31}$. Finally, the second pass must end with a right shift over 4 digits to obtain the pixel values in the spatial domain.

The tables for 12-bit samples

As the documentation of JPEGLib version 6b states in the file install.doc "Currently, 12-bit support does not work on 16-bit-int machines". Therefore also with this new implementation no support for 12-bit samples on 16-bit platforms is provided. On 32-bit platforms, the tables from the 8-bit sample implementation described before are used, but downscaled by 4 digits. The final downscaling in the second pass is therefore only by 16 digits instead of 20 digits as is the case for 8-bit samples.

Accuracy

When downscale-decoding a JPEG image with the implementation explained in this document, visual differences between the current implementation of downscale-decoding can not be perceived, not even for 16-bit platforms with USE_INACCURATE_IDCT defined. Therefore the difference between the current implementation and these new variants have been measured for 8-bit sample data using the file testimg.jpg that comes with the JPEGLib. Table 1 shows the deviations for 32-bit and 16-bit platforms with USE_INACCURATE_IDCT undefined, table 2 shows the deviations for 32-bit and 16-bit platforms with USE_INACCURATE_IDCT defined.

	16-bit platform	**32-bit platform**
Peak Error:	1	1
Mean Square Error:	0.028778	0.009541
Mean Error:	-125	8
Different Pixel Components:	187	62

Table 1: Deviations of the new implementation from the standard implementation with USE_INACCURATE_IDCT undefined

	16-bit platform	**32-bit platform**
Peak Error:	4	1
Mean Square Error:	0.357956	0.009541
Mean Error:	-156	8
Different Pixel Components:	1587	62

Table 2: Deviations of the new implementation from the standard implementation with USE_INACCURATE_IDCT defined

What can be found surprising is the fact that at least in the case of downscale-decoding the file testimg.jpg for 32 bits, the result is exactly the same decoded image no matter whether USE_INACCURATE_IDCT is defined or not. However, it is unclear whether this is generally the case.

Performance

Performance was first tested on a 16-bit platform, namely MS-DOS version 6.22. This choice of operating system was done to ensure that no two or more concurrently running processes and scheduler strategies could have influence on the measured performance. Intentionally, the smartdrive hard disk cache was turned on in order to delay the write operations during decoding with djpeg until the end of the process. This way very constant conditions for measuring performance could be achieved. As the memory manager, jmemnobs.c was used. The compiler being used was Watcom version 10.6 and for both the current implementation of the JPEGLib and the new variants, the same compiler settings were used. The computer being used for measuring the numbers in the following was a rather old 486 computer with a Cyrix CPU running at 8 MHz of clock speed. In order to really measure the theoretically possible performance improvement, the adaptive IDCT that checks for rows and columns of zeros were turned off in both the old and the new implementations. The file that was downscale-decoded was a mid-quality JPEG image of size 88007 bytes and a resolution of 1024×768 pixels in 4:4:4 chroma subsampling.

Without USE_INACCURATE_IDCT defined, it took 88 seconds to decode this file, with USE_INACCURATE_IDCT defined, this took 72 seconds. The JPEGLib's current implementation needed 96 seconds to decode this file. In order to measure the improvement of the IDCT step alone, a variant of the JPEGLib was built that simply omits the IDCT step. This variant took 48 seconds to execute. This means, that the downscale-decoding step in the current implementation of the JPEGLib took $96 - 48 = 48$ seconds. The IDCT without USE_INACCURATE_IDCT defined took $88 - 48 = 40$ seconds, which is an improvement in performance of the IDCT of roughly 16 %. With USE_INACCURATE_IDCT defined, the IDCT took $72 - 48 = 24$ seconds, which is an improvement in performance of the IDCT of roughly 50 %. Note however that if USE_INACCURATE_IDCT is not defined, the new implementation's adaptive IDCT in the first pass is slightly less performant than the current implementation one's since it must use a temporary INT32 variable in order to store the result of the dequantization which is then scaled by a right shift over 6 digits[2]. Note also, that this is not an issue on a 32-bit platform which uses 32-bit integers throughout the whole algorithm.

The second test platform was the same hardware at the same clock speed running Windows NT 4.0 Workstation with only those processes running that are essential for the operating system to work. The compiler in use was Microsoft Visual C/C++ 6.0. The makefile and jconfig.h file were the ones that come with the JPEGLib package for usage with Visual C/C++. The same tests as above for this 32-bit binary took 53.7 seconds for the implementation that comes with the current release of the JPEGLib. Both new variants of the algorithm took 50.6 seconds and the variant that does not perform the IDCT step took 33.0 seconds to execute. This means that for the JPEGLib's current release the IDCT step takes roughly $53.7 - 33.0 = 20.7$ seconds whereas the new variants take $50.6 - 33.0 = 17.6$ seconds. This means that the IDCT step is around 15 % faster with the new implementations on this platform.

[2]Generally, for a 16-bit platform, image quality probably suffers with this right shift operation if the dequantized DCT coefficients have low values.

Summary

As a summary, the following steps need to be done to use the new functionality in a project that uses version 6b of the JPEGLib:

- Define the macro USE_FASTER_2x2_IDCT in jconfig.h.
- Optionally define the macro USE_INACCURATE_IDCT in jconfig.h.
- Download the modified source files from http://www.stefan-kuhr.de/jpeg/jpeg2x2code
- Replace the two files jddctmgr.c and jidctred.c of version 6b of the JPEGLib with the versions you just downloaded.
- Rebuild the library.

The solution was verified with the Microsoft Visual C/C++ compiler version 6.0 for Win32 and the Watcom version 10.6 compiler for MS-DOS as the target. The author would appreciate any feedback or comments from people who are using this code.