ConsumerBench™ Version 1.1 Benchmark Name: JPEG

Highlights

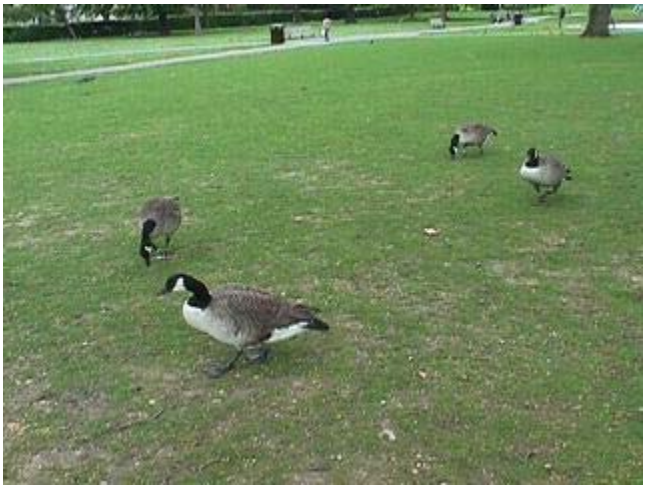
- **Benchmarks Potential Performance for still picture image coding (e.g. still camera)**
- **Compression and decompression benchmarks**
- **Integer math, with diverse operand types, accessing large image memories**

Application

The JPEG compression benchmark takes an image and encodes it to produce a compressed representation. The JPEG image compression standard provides for a wide range of options in the way that images are compressed. The benchmark uses the baseline subset of image compression "tools" with parameters that would generally be regarded as typical.

The JPEG decompression benchmark essentially reverses the process of the compression benchmark. Since the compressed image that is used by the benchmark is that produced by the compression benchmark (above) it uses the same set of image coding tools and parameters.

This is the particular image processed during these benchmarks:



The benchmark provides an indication of the potential performance of a microprocessor in an application requiring still-image compression and decompression (for example a still picture camera).

Benchmark Description

The JPEG compression benchmark takes an image and encodes it. The image used in the benchmark is of relatively low resolution (320 pixels by 240 lines) represented in the RGB (Red-Green-Blue) color space, with each component being represented by 8-bit data.

The benchmark first performs a number of preprocessing steps on the image data:

- The image is color-space converted to a YCrCb color space that uses a luminance, Y, component (a black-and-white image) together with two color-difference components, Cr and Cb.
- The two color difference components are scaled so as to have one half the number of pixels and one half the number of lines as the luminance component.

The benchmark code then produces the JPEG header information which includes data about the size and nature of the image as well as the detailed quantization matrices and Huffman code tables that are being used. (These are required in order for the JPEG decompression to decode the resulting bitstream.)

The JPEG algorithm then segments the image to be coded into a series of MCUs (Minimum Coded Unit) consisting of four 8x8 pixel blocks of the luminance component and the corresponding 8x8 pixel blocks for each of the two color difference components. Each of these 8x8 pixel blocks is then processed as follows:

- 2-D DCT
  A two-dimensional transform is performed on the data resulting in an 8x8 array of frequency-domain coefficients for the block. A "fast" algorithm analogous to the FFT (Fast-Fourier-Transform) is used. The particular decomposition is such that 16 one-dimensional 8-point transforms are performed each requiring 12 multiples and 32 adds.

- Quantization
  Each of the frequency-domain coefficients is divided by a scale factor unique to that particular spatial frequency to yield an integer "code". This will subsequently be used in a decoder (by multiplying by the scale factor) to derive an approximation to the original frequency-domain coefficient. JPEG is "lossy" in the sense that the decoded images are an approximation to the original images and it is at this stage that information is lost.

- Zig-Zag scan
  The quantized coefficients are scanned in a "zig-zag" fashion to produce a 1-D sequence of 64 coefficients. A large number of these coefficients that are zero. Each non-zero coefficient is represented as a "SIZE" value. The number of zero coefficients preceding the non-zero coefficient is referred to as the "RUN".

- Huffman encode
  Each possible combination of RUN and SIZE is allocated a unique Huffman code word such that statistically likely RUN-SIZE combinations have short code words while RUN-SIZE combinations that occur infrequently have long code words. The appropriate Huffman code word is looked up (a table lookup operation) inserted into the bitstream and followed by "SIZE" binary digits to specify the value of the quantized coefficient.

(This description glosses over many details but gives a general feel for the operations performed.)

The JPEG decompression benchmark essentially performs the same series of steps, but in reverse. Huffman decoding is a somewhat more complex operation to Huffman encoding (which is just a simple table lookup). The Inverse Quantization stage involves multiplication rather than division and may therefore be less demanding on many processors. However in broad terms the computational burden of decoding JPEG is similar to that of encoding.

Analysis of Computing Resources

The JPEG benchmarks use a wide range of types of operations:

- Operations on 8-bit data for the scaling and color space pre- and post-processing stages.
- Extensive arithmetic on 16-bit data in the transform (DCT) and quantization stages with various intermediate values requiring more than 16-bits.
- Table lookup and low-level bit manipulation operations for Huffman coding and decoding and assembling and unpacking the coded bitstream.

The image used in the benchmark is relatively small (320 pixels by 240 lines) with three bytes per pixel. (A total of 225 Kbytes.) JPEG is reasonably scalable and engineers might broadly expect the time required to process an image to scale proportionally with the number of pixels. However, the computational demands of JPEG are dependent on image content, particularly in the entropy (Huffman) coding section, and since the statistical content of typical images does vary with image resolution caution should be exercised in scaling performance over a wide range.
Users of these benchmark results should also be aware that it is NOT designed to indicate worst-case performance characteristics. The computational demands of JPEG are dependent on the specific image being coded (or decoded) and the choice of coding "tools" and parameters that are chosen.

Though the image and encoded data buffer sizes in memory are large, the algorithm proceeds block-by-block and the data for a block will generally be in cache.

The high spatial and temporal locality of reference inherent in the JPEG algorithm allows processors to make good use of caches. Even small data caches work well, with miss rates decreasing as cache size increases, there is no "knee" in the curve where performance increases markedly because the data fits within the cache. The small size of the algorithm kernel will fit in even very small instruction caches (for example 4Kbytes), however the total code size is significantly larger so that larger instruction caches do provide additional performance benefit.