ConsumerBench™1.1

software benchmark data book

www.eembc.org

An Industry-Standard Benchmark Consortium

Table of Contents

An Industry-Standard Benchmark Consortium

ConsumerBench™ Version 1.1 Benchmark Name: High Pass Grey-Scale Filter

Highlights

- **Benchmarks performance for digital image processing used in digital still camera and other digital image products**
- **Explores 2-D data array access and multiply / accumulate capability.**
- **This benchmark has potential for Full-Fury benchmark optimization, especially by SIMD and VLIW architectures.**

Application

A high pass grey-scale filter is used in the front end processing of DSCs (Digital Still Camera). RGB data from either CCD or CMOS sensors is pre-processed by this filter to deliver image enhancement, and then passed to the JPEG image compression processing. This filter takes a "blurry" image and sharpens it with a 2-dimensional spatial filter.

DSCs implement this filter either in software or hardware, with software giving the flexibility to add customization for picture quality. The number of filter taps can vary from 3(H) x 3(V) to more than 5(H) x 5(V).

This benchmark is one of the most frequently used algorithms in image processing and represents a good measure of the CPU performance in digital imaging products.

Benchmark Description

This benchmark explores the target CPU's capability to perform two dimensional data array access and multiply/accumulate calculation.

For each pixel in the image, the filter calculates the output result from the 9 pixels (including the center pixel) multiplied by filter coefficients, accumulated and then shifted left by 8-bits. The 2-dimensional coefficients used here are:

$$\begin{pmatrix} F11 & F21 & F31 \\ F12 & F22 & F32 \\ F13 & F23 & F33 \end{pmatrix} = \begin{pmatrix} -28 & -28 & -28 \\ -28 & 255 & -28 \\ -28 & -28 & -28 \end{pmatrix}$$

Each pixel is computed according to the following equation:.

```
PelValue = (Short)(  F11*P(c-w-1) +F21*P(c-w)  +F31*P(c-w+1)
                     +F12*P(c-1)   +F22*P(c)    +F32*P(c+1)
                     +F13*P(c+w-1)+F23*P(c+w)  +F33*P(c+w+1) )

Out = (Byte)(PelValue >>8);
```

Here, P(i) is the pixel intensity, c is the center location of the filter window, w is the width of the input image. The data type of P(i) is Byte, and the two dimensional data is arranged in a linear way. Therefore addition or

subtraction of the horizontal image width "w" and offset of "-1" or "+1" are required to retrieve the 2-dimension window data. The accumulation is performed as a 16-bit data and the final output data is converted to a Byte data after a shift right by 8-bits. The top/left and right/left borders are black out by assigning "BLACK" value of 0.

The input data size is 320-pixels in the horizontal direction and 240-pixels in the vertical direction. This is a monochrome or gray-scale calculation. It is not an RGB calculation where the same process is performed three times. Usually the enhancement is performed just in the luminance signal Y, which is the gray-scale signal.

If the benchmark score is extrapolated for a larger image, the processing time will be almost linearly proportional to the pixel count (e.g. For a 640 x 480 image, it will be x4 times). The iteration/sec score will be the inverse e.g. for a 640 x 480 image, iteration/sec it will be x1/4.

Analysis of Computing Resources

Out of the Box Benchmark: A 'for loop' calculates the filter output one pixel at a time. For one pixel calculation, the center pixel itself and the eight neighbor pixel data should be loaded. This is a time consuming process, considering the offset/width index calculation, and the time spent for the memory or cache access. Higher performance would be expected from a microprocessor with a single-cycle MAC unit.

Full-Fury Benchmark: Because of the simple structure of the multiplication and accumulation, a VLIW or SIMD architecture with multiple MAC units are able to offer a simple acceleration. Another possible optimization is loading multiple Bytes at a time, although a SIMD architecture may show some overhead for the rearranging the data to feed the SIMD engine.

Regarding the memory architecture, the image data is repeatedly used for the consecutive window and can benefit from a Data Cache. The code size is trivial and will easily fit in to a small L1 Instruction Cache.

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.

ConsumerBench™ Version 1.1

Benchmark Name: RGB to CMYK Conversion

Highlights

- **Benchmarks digital image processing performance in printers and other digital imaging products.**
- **Explores basic arithmetic and minimum value detection capability.**
- **This benchmark provides opportunities for Full-Fury benchmark optimization. Conditional move and multi-Byte processing SIMD or VLIW architectures are effective for example.**

Application

RGB to CMYK conversion is widely used in color printers. RGB inputs from PC data is converted to CMYK color signals for printing.

Benchmark Description

This benchmark explores the target CPU capability for basic arithmetic and minimum value detection.

R, G, B 8-bit pixel color image input is fed to the following equation:

```
/* calculate complementary colors */
c = 255 – R;
m = 255 – G;
y = 255 – B;

/* find the black level k */
K = minimum (c,m,y)

/* correct complementary color lever based on k */
C = c – K
M = m – K
Y = y - K
```

RGB values are in the range of [0:255]..

CMYK values are in the range of [0:255]..

The input and output data size is 320-pixels in the horizontal direction and 240-pixels in the vertical direction. The 320x240 data for RGB and CMYK is stored sequentially as.

R[0], G[0], B[0], R[1], G[1], B[1],……R[76799], G[76799], B[76799]

C[0], M[0], Y[0], K[0], C[1], M[1], Y[1],K[1]……C[76799], M[76799], Y[76799], K[76799]

The pointers are just incremented by one to access R, G, B or C, M, Y, K data is this order.

If the benchmark score is extrapolated for a larger image, the processing time will be almost linearly proportional to the pixel count (e.g. For a 640 x 480 image, it will be x4 times.) The iteration/sec score will be the inverse e.g. for a 640 x 480 image, iteration/sec it will be x1/4. There is data dependency in the cycle counts for the minimum value K search, due to branch taken or not taken. If this operation is handled by conditional move, the cycle will constant.

Analysis of Computing Resources

Out of the Box Benchmark: A 'for loop' calculates the conversion of a set of RGB inputs and CMYK outputs at a time. A set of R, G, B input data is read from the memory by incrementing a read pointer. A set of output C, M, Y, K output data is written back to the memory by incrementing a write pointer. There is no complex 2-dimensonal access like the high pass grey-scale filter benchmark.

The complementary color calculation and correction are simple subtract calculations without any MAC operation.

The minimum value search has two branches for processing each pixel.

```
If (c<m) {
        K = (Byte)(c<y ? c:y);
      }
  else {
        K = (Byte)(m<y ? m:y);
      }
```

This can be a very expensive routine because of the branch penalty.

Full-Fury Benchmark: By using compare and conditional moves, the branch penalty can be avoided. VLIW and SIMD can process multiple Byte of data at a time. A SIMD architecture which can handle multiple of Byte data at a time, is especially suited to this benchmark e.g. A 4-way SIMD microprocessor can handle 4 x 8-bit data every cycle.

Special Notes

Regarding the memory architecture, the image data is used just once and there is no benefit from a big Data Cache, unless the microprocessor has a cache prefetch feature. A small Data Cache will work to fetch consecutive data and avoid external memory access overhead. The code size is trivial and easily fits in to a small L1 Instruction Cache.

ConsumerBench™ Version 1.1

Benchmark Name: RGB to YIQ Conversion

Highlights

- **Benchmarks performance for digital video processing.**
- **Explores multiply / accumulate capability.**
- **This benchmark has opportunities for Full-Fury benchmark optimizations, especially by SIMD and VLIW architectures.**

Application

RGB to YIQ conversion is used in the NTSC encoder where the RGB inputs from the camera are converted to a luminance (Y) and two chrominance information (I,Q). In the NTSC encoder, these I,Q signals are modulated by a subcarrier and added to the Y signal.

Historically, when color TVs appeared in the market, they had to coexist with the existing monochrome TVs and this was made possible with the NTSC signal structure. The chrominance signals are averaged out as a fine mesh of invisible signals in the monochrome TV sets.

YUV used in the European PAL standard and YCbCr used in the JPEG standard have different codings. All three standards share the same luminance signal Y but the chrominance calculations are different. The matrix calculation scheme used in the RGB to YIQ can be applied to these standards too.

In the actual products, this trivial calculation is usually performed in dedicated hardware, especially in digital video products. For cost saving and flexibility, this algorithm can be implemented in software if the CPU is powerful enough and where the digital image is a still picture.

Benchmark Description

This benchmark explores the capability of the CPU to perform a straightforward matrix multiply/accumulate calculation.

The R, G, B 8-bit pixel color image input is processed as follows:

$$Y = 0.299*R + 0.587*G + 0.114*B$$
$$I = 0.596*R - 0.275*G - 0.321*B$$
$$Q = 0.212*R - 0.523*G + 0.311*B$$

RGB values are in the range of [0:255]. The conversion coefficients are 16-bits. The multiply/accumulate results are shifted right by 16-bits. Before the shift, 1 is added to a bit location right to the LSB of the shifted result for rounding to the nearest integer.

The output is 8-bit data. Y is in the range of [0,255] and I,Q in the range of [-127, 127].

The input and output data size is 320-pixels in the horizontal direction and

240-pixels in the vertical direction. The 320x240 data for RGB and YIQ are stored sequentially as.

R[0], G[0], B[0], R[1], G[1], B[1],……R[76799], G[76799], B[76799]

Y[0], I[0], Q[0], Y[1], I[1], Q[1],……Y[76799], I[76799], Q[76799]

The pointers are just incremented by one to access R, G, B or Y, I, Q data is this order.

Analysis of Computing Resources

Out of the Box Benchmark: A 'for loop' calculates the conversion of a set of RGB inputs and YIQ outputs at a time. A set of R, G, B input data is read from the memory by incrementing a read pointer. A set of output Y, I, Q output data is written back to the memory by incrementing a write pointer. There is no complex 2-dimensonal access such as that in the high pass grey-scale filter benchmark.

The calculation is a straightforward multiplication and accumulation that a microprocessor with a single-cycle MAC unit will benefit from.

The code size is trivial and easily fits in to a small L1 Instruction Cache.

Full-Fury Benchmark: Because of the simple structure of the multiplication and accumulation, a VLIW or SIMD architecture with multiple of MAC units can be used to accelerate performance. A further optimization is the loading of multiple Bytes at a time. Software pipelining could be used to pass the loaded data efficiently to the MAC unit for calculation.