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.