An Industry-Standard Benchmark Consortium

OABench™ Version 2.0 — Benchmark Name: Dithering

Highlights

- Benchmarks potential performance of a printer application
- Uses Floyd-Steinberg Error Diffusion Dithering Algorithm (1975)
- Converts 8bpp grayscale image to 1bpp monochrome.
- Largely integer math with shifts and logical compares
- A component of the EEMBC OAV2mark™
- Implements 11 new data files compared with OABench Version 1.1
- Implements cyclical redundancy checksum (CRC) for self-checking as well as the ability to view processed output files (new in Version 2)
- Jarvis Grayscale Dithering included for debugging purposes

History, Application, and Restrictions

The dithering benchmark is representative of color and monochrome printer applications. The algorithm converts a grayscale image into a form ready for printing using the Floyd-Steinberg Error Diffusion dithering algorithm. This algorithm propagates an error quantity from image row to image row, effectively diffusing errors from the rendering calculations and preventing unwanted printing artifacts, such as banding.

References: Robert Ulichney (1987); *Digital Halftoning,* The MIT Press, Cambridge, Massachusetts; pp. 239-242

Benchmark Description

The benchmark changes a grayscale 8bpp image to a 1bpp monochrome image, using a Floyd-Steinberg Error Diffusion dithering algorithm. It uses two image buffers (one for the source image and a second for the generated output) and two line buffers to hold error data. Two “error” arrays are used — one for saving the errors from the current row (used to dither the next row) and one from the previous row, used to diffuse the errors from that row to the current pixel. This array must be zeroed out before the first row, to ensure that no spurious data is left there.

The error array is created such that there is one extra value at either end. This eliminates special processing at the start and end of each row (but requires zeroing the additional columns).

Each pixel of the input image file is processed as follows:

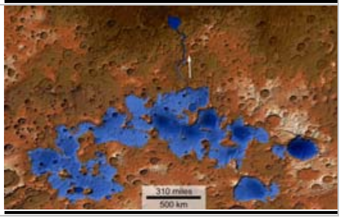
1. Calculate an “error” value using the history buffer (weighted values of surrounding pixels).
2. Calculate a monochrome output pixel value and store.
3. Store “error” value to next line history buffer.

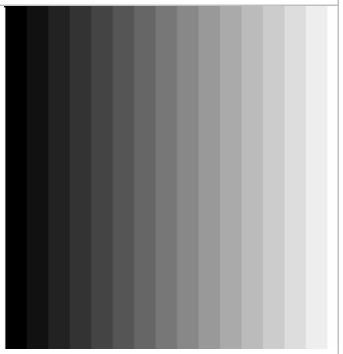
Benchmark Description (continued)

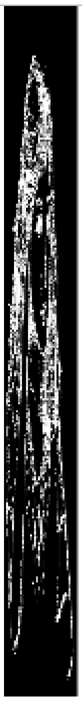
For Version 2, datasets are now taken from external data files (the same .pgm files as found in DENbench Version 1.0), and data is output to files as well to aid in verification. The input data files are:

Data Name	Data File	Attributes	Picture
Data 1	DavidAndDogs	564x230, 256 shades of gray. The image has 215 unique colors.	
Data 2	DragonFly	606x896, 16 million colors. The image has 162,331 unique colors. Highlights, wide range of contrasts.	
Data 3	EEMBCGroup Shot-Miami	EEMBCGroupShotMiami: 640x480, 16 million colors. The image has 181,872 unique colors. Large number of fleshtones, highest number of unique colors in data set.	

Data 4	Galileo	290x415, 16 million colors. The image has 36,557 unique colors, and also contains "real black" for over 30% of the picture, which is interesting from an optimization perspective.	
Data 5	Goose	320x240, 256 colors. The image has 22921 unique colors.	

Data 6	Mandrake	320x240, 16 million colors. The image has 71,482 unique colors.	
Data 7	MarsFormer Lakes	800x482, 16 million colors. The image has 91,152 unique colors.	

Data 8	Rose256	227x149, 256 colors. The image contains 256 unique colors.	
Data 9	Dragon	370 x 384, 256 colors, 88 unique colors.	
Data 10	Gradient	A grayscale gradient shading test pattern. 256 x 256, 256 colors.	

Data 11	Medium	A long, thin, black and white picture having 37 x 345 pixels, 256 colors, and 255 unique colors.	
200 iterations are the default, 2 for CRC verification runs.			

Analysis of Computing Resources

The benchmark effectively stresses four areas of the target CPU:

- Indirect references used for managing internal buffers
- Manipulation of large data sets, since large images will stress the cache
- Ability to manipulate packed-byte quantities, used to hold grayscale pixel information
- Ability to perform four byte-wide multiply-accumulate operations per pixel

The instruction mix for this benchmark is very architecture and compiler dependent, since the main part of the inner loop can be implemented with add/sub/shift, or multiplies, or MAC instructions depending on hardware characteristics.

Analysis of Computing Resources (continued)

The C library function memset() is called twice per iteration (for the output buffer and for the error buffers). No floating-point calculations are used. The code size is small and the data size is large. By using multiple data sets (and private EEMBC data for certification), data-focused optimization is eliminated.

Optimizations Allowed	**Out of the Box / Standard C** **Full Fury / Optimized**

- The C code must not be changed for Out-of-the-Box unless it must be modified to get it to compile. All changes must be documented, authorized by the certification authority, and must not have a performance impact.
- For Out-of-the-Box, additional hardware can be used if it does not require code changes.
- All optimized libraries must be part of the standard compiler package, and/or available to all customers.
- Test harness changes may be made for portability reasons if they do not impact performance.
- For Optimized, the basic algorithm may not be changed, but the code may be rewritten in assembler. Rewriting the code to take advantage of parallelism is allowed so long as the correct answers are achieved using any arbitrary keys (not just those supplied in the benchmark code).
- For Optimized, optimized libraries can be used if they are publicly available.
- For Optimized, in lining is allowed.
- Additional data files may be used during certification to ensure the correctness of the optimized benchmark. You should *not* assume data patterns during optimization.
- Profile directed optimization is allowed using train data set 1, DavidAndDogs.pgm.