Multi-Core Programming

Increasing Performance through Software Multi-threading

Shameem Akhter
Jason Roberts

These pages were excerpted from Chapter 3 of *Multi-Core Programming* by Shameem Akhter and Jason Roberts.

Visit Intel Press on the web at www.intel.com/intelpress to learn more about this book.

This excerpt illustrates how a seemingly sequential problem, error diffusion, can be transformed into an efficient parallel implementation suitable for parallel processors.

Intel
PRESS

ISBN 0-9764832-4-6

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in professional services. If professional advice or other expert assistance is required, the services of a competent professional person should be sought.

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

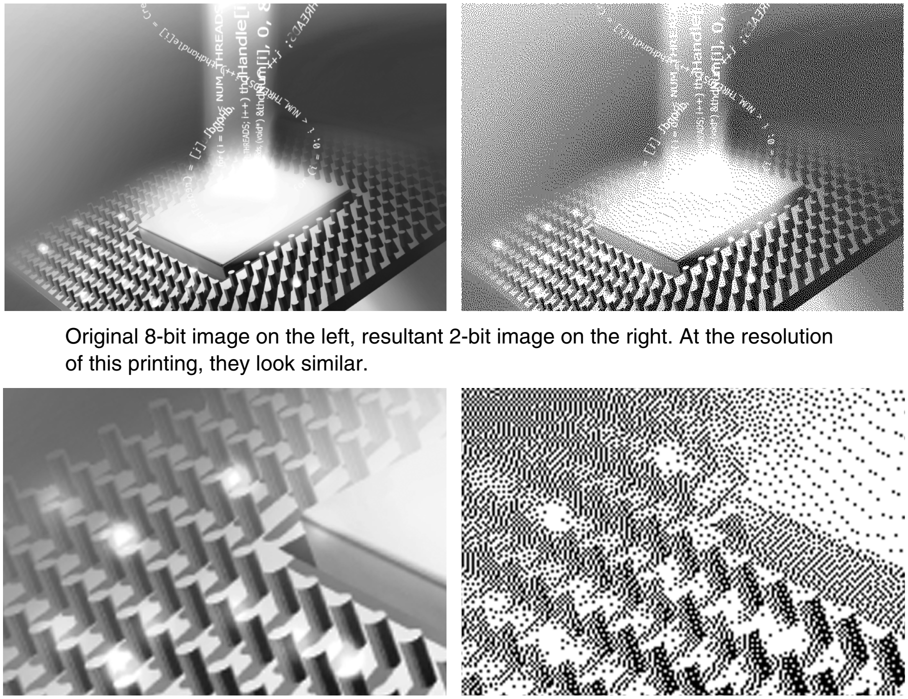
Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

Fictitious names of companies, products, people, characters, and/or data mentioned herein are not intended to represent any real individual, company, product, or event.

Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

A Motivating Problem: Error Diffusion

To see how you might apply the aforementioned methods to a practical computing problem, consider the error diffusion algorithm that is used in many computer graphics and image processing programs. Originally proposed by Floyd and Steinberg (Floyd 1975), *error diffusion* is a technique for displaying continuous-tone digital images on devices that have limited color (tone) range. Printing an 8-bit grayscale image to a black-and-white printer is problematic. The printer, being a bi-level device, cannot print the 8-bit image natively. It must simulate multiple shades of gray by using an approximation technique. An example of an image before and after the error diffusion process is shown in Figure 3.2. The original image, composed of 8-bit grayscale pixels, is shown on the left, and the result of the image that has been processed using the error diffusion algorithm is shown on the right. The output image is composed of pixels of only two colors: black and white.



Original 8-bit image on the left, resultant 2-bit image on the right. At the resolution of this printing, they look similar.

The same images as above but zoomed to 400 percent and cropped to 25 percent to show pixel detail. Now you can clearly see the 2-bit black-white rendering on the right and 8-bit gray-scale on the left.

Figure 3.2 Error Diffusion Algorithm Output

The basic error diffusion algorithm does its work in a simple three-step process:

1. Determine the output value given the input value of the current pixel. This step often uses quantization, or in the binary case, thresholding. For an 8-bit grayscale image that is displayed on a 1-bit output device, all input values in the range [0, 127] are to be displayed as a 0 and all input values between [128, 255] are to be displayed as a 1 on the output device.
2. Once the output value is determined, the code computes the error between what should be displayed on the output device and what is actually displayed. As an example, assume that the current input pixel value is 168. Given that it is greater than our threshold value (128), we determine that the output value will be a 1. This value is stored in the output array. To compute the error, the program must normalize output first, so it is in the same scale as the input value. That is, for the purposes of computing the display error, the output pixel must be 0 if the output pixel is 0 or 255 if the output pixel is 1. In this case, the display error is the difference between the actual value that should have been displayed (168) and the output value (255), which is –87.
3. Finally, the error value is distributed on a fractional basis to the neighboring pixels in the region, as shown in Figure 3.3.

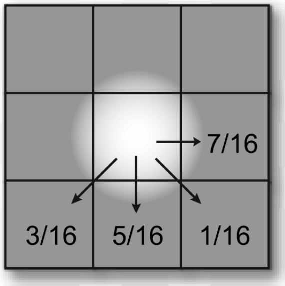


Figure 3.3 Distributing Error Values to Neighboring Pixels

This example uses the Floyd-Steinberg error weights to propagate errors to neighboring pixels. 7/16ths of the error is computed and added to the pixel to the right of the current pixel that is being processed. 5/16ths of the error is added to the pixel in the next row, directly below the current pixel. The remaining errors propagate in a similar fashion. While you can use other error weighting schemes, all error diffusion algorithms follow this general method.

The three-step process is applied to all pixels in the image. Listing 3.1 shows a simple C implementation of the error diffusion algorithm, using Floyd-Steinberg error weights.

```
/**************************************
* Initial implementation of the error diffusion algorithm.
***************************************/

void error_diffusion(unsigned int width,
                     unsigned int height,
                     unsigned short **InputImage,
                     unsigned short **OutputImage)
{
   for (unsigned int i = 0; i < height; i++)
   {
      for (unsigned int j = 0; j < width; j++)
      {
         /* 1. Compute the value of the output pixel*/
         if (InputImage[i][j] < 128)
            OutputImage[i][j] = 0;
         else
            OutputImage[i][j] = 1;

         /* 2. Compute the error value */
         int err = InputImage[i][j] - 255*OutputImage[i][j];

         /* 3. Distribute the error */
         InputImage[i][j+1]   += err * 7/16;
         InputImage[i+1][j-1] += err * 3/16;
         InputImage[i+1][j]   += err * 5/16;
         InputImage[i+1][j+1] += err * 1/16;
      }
   }
}
```

Listing 3.1 C-language Implementation of the Error Diffusion Algorithm

Analysis of the Error Diffusion Algorithm

At first glance, one might think that the error diffusion algorithm is an inherently serial process. The conventional approach distributes errors to neighboring pixels as they are computed. As a result, the previous pixel's error must be known in order to compute the value of the next pixel. This interdependency implies that the code can only process one pixel at a time. It's not that difficult, however, to approach this problem in a way that is more suitable to a multithreaded approach.

An Alternate Approach: Parallel Error Diffusion

To transform the conventional error diffusion algorithm into an approach that is more conducive to a parallel solution, consider the different decomposition that were covered previously in this chapter. Which would be appropriate in this case? As a hint, consider Figure 3.4, which revisits the error distribution illustrated in Figure 3.3, from a slightly different perspective.

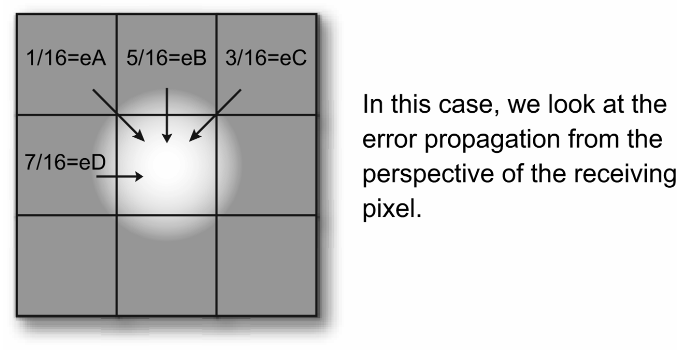
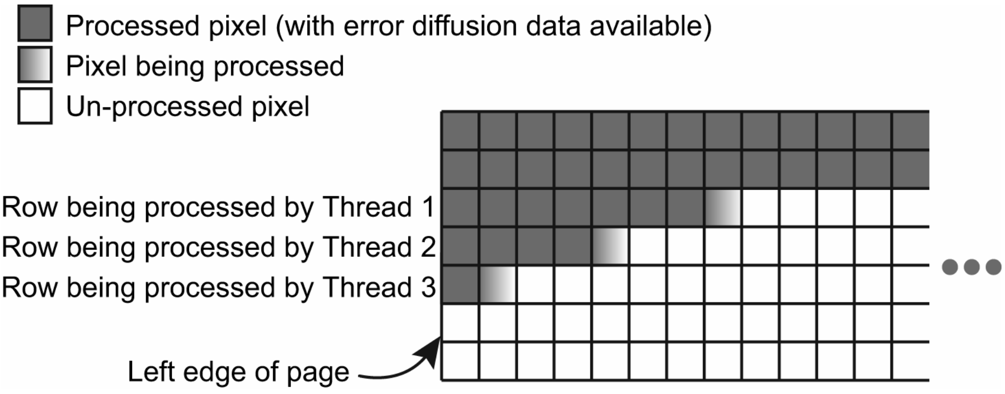

1/16=eA
5/16=eB
3/16=eC
7/16=eD
In this case, we look at the error propagation from the perspective of the receiving pixel.

Figure 3.4 Error-Diffusion Error Computation from the Receiving Pixel's Perspective

Given that a pixel may not be processed until its spatial predecessors have been processed, the problem appears to lend itself to an approach where we have a producer—or in this case, multiple producers—producing data (error values) which a consumer (the current pixel) will use to compute the proper output pixel. The flow of error data to the current pixel is critical. Therefore, the problem seems to break down into a data-flow decomposition.

Now that we identified the approach, the next step is to determine the best pattern that can be applied to this particular problem. Each independent thread of execution should process an equal amount of work (load balancing). How should the work be partitioned? One way, based on the algorithm presented in the previous section, would be to have a thread that processed the even pixels in a given row, and another thread that processed the odd pixels in the same row. This approach is ineffective however; each thread will be blocked waiting for the other to complete, and the performance could be worse than in the sequential case.

To effectively subdivide the work among threads, we need a way to reduce (or ideally eliminate) the dependency between pixels. Figure 3.4 illustrates an important point that's not obvious in Figure 3.3—that in order for a pixel to be able to be processed, it must have three error values (labeled eA, eB, and eC[1] in Figure 3.3) from the previous row, and one error value from the pixel immediately to the left on the current row. Thus, once these pixels are processed, the current pixel may complete its processing. This ordering suggests an implementation where each thread processes a row of data. Once a row has completed processing of the first few pixels, the thread responsible for the next row may begin its processing. Figure 3.5 shows this sequence.


Processed pixel (with error diffusion data available)
Pixel being processed
Un-processed pixel
Row being processed by Thread 1
Row being processed by Thread 2
Row being processed by Thread 3
Left edge of page

Multiple threads are able to process multiple rows simultaneously.

Figure 3.5 Parallel Error Diffusion for Multi-thread, Multi-row Situation

[1] We assume eA = eD = 0 at the left edge of the page (for pixels in column 0); and that eC = 0 at the right edge of the page (for pixels in column W-1, where W = the number of pixels in the image).

Notice that a small latency occurs at the start of each row. This latency is due to the fact that the previous row's error data must be calculated before the current row can be processed. These types of latency are generally unavoidable in producer-consumer implementations; however, you can minimize the impact of the latency as illustrated here. The trick is to derive the proper workload partitioning so that each thread of execution works as efficiently as possible. In this case, you incur a two-pixel latency before processing of the next thread can begin. An 8.5" X 11" page, assuming 1,200 dots per inch (dpi), would have 10,200 pixels per row. The two-pixel latency is insignificant here.

The sequence in Figure 3.5 illustrates the data flow common to the wavefront pattern.