4-14-2003

High-Performance Architecture for Color Error Diffusion

Christopher R. Brown
Microwave Data Systems Inc.

Andreas Savakis
Rochester Institute of Technology

Follow this and additional works at: <http://scholarworks.rit.edu/article>

Recommended Citation

Christopher R. Brown, Andreas E. Savakis, "High-performance architecture for color error diffusion", Proc. SPIE 5012, Real-Time Imaging VII, (14 April 2003); doi: 10.1117/12.477498; <https://doi.org/10.1117/12.477498>

This Article is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Articles by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

HIGH-PERFORMANCE ARCHITECTURE FOR COLOR ERROR DIFFUSION

Christopher Brown¹ and Andreas Savakis²

¹ Microwave Data Systems
Rochester, New York 14620
cbrown@microwavedata.com

² Department of Computer Engineering
Rochester Institute of Technology
Rochester, New York 14623
savakis@ieee.org

ABSTRACT

Error diffusion is one of the most widely used algorithms for halftoning gray scale and color images. It works by distributing the thresholding error of each pixel to unprocessed neighboring pixels, while maintaining the average value of the image. Error diffusion results in inter-pixel data dependencies that prohibit a straightforward data pipelining processing approach and increase the memory requirements of the system. In this paper, we present a multiprocessing approach to overcome these difficulties, which results in a novel architecture for high performance hardware implementation of error diffusion algorithms. The proposed architecture is scalable, flexible, cost effective, and may be adopted for processing gray scale or color images. The key idea in this approach is to simultaneously process pixels in separate rows and columns in a diagonal arrangement, so that data dependencies across processing elements are avoided. The processor was realized using an FPGA implementation and may be used for real-time image rendering in high-speed scanning or printing. The entire system runs at the input clock rate, allowing the performance to scale linearly with the clock rate. Higher data rate applications required by future applications will automatically be supported using more advanced high-speed FPGA technologies.

1. INTRODUCTION

Error diffusion is one of the most important and popular algorithms for halftoning gray scale and color images [1-3]. The development of flexible hardware solutions that may be used for real-time image rendering via error diffusion can be very useful in high-speed scanning applications. In this paper, we present a novel architecture for a high-performance color error diffusion image processor. The proposed design may be used in production scanning to process 100-120 pages per minute at a resolution up to 600 dpi. This results in a maximum image size of (5100×6600) and imposes a rough processing requirement of 75 megapixels per second. It is possible to use field programmable gate arrays (FPGAs), media processors, or custom designs to meet these performance requirements. While fully customized designs are common and provide the highest performance, they can be quite expensive when compared to total system cost. An important design goal is to achieve the speed requirement at minimum system cost.

There are many advantages of using an FPGA implementation for a design of this nature [4]. An FPGA solution offers a reasonable tradeoff between performance and cost, and provides immense flexibility in the nature of image processing algorithms that can be handled [5]. A large number of algorithms can be resident in nonvolatile system memory, limited only by the system memory size. The desired algorithm for a given application can be downloaded to the FPGA before processing the images under consideration.

2. MULTIPROCESSING APPROACH TO ERROR DIFFUSION

The well-known Floyd-Steinberg color error diffusion technique provides a good compromise between output image quality and required processing power [1,2]. Each 24-bit input pixel is transformed to a printable 3-bit color output value using any of a number of techniques. The error between the output value and the input value is then diffused to four unprocessed (future) pixels in the neighborhood using the weights shown in Figure 1. The diffusion process adds considerable complexity to the design of hardware because it imposes data dependency between immediately neighboring pixels. The result from any given pixel must be completely determined before processing any of the four future pixels within the 8-neighborhood. Due to data dependency, pipelining alone cannot achieve the desired performance. In addition, the interaction between pixels in different rows suggests that a large amount of memory is required, so that the modified image data and the error can be stored, which makes a naïve design approach expensive.

In this paper, a multiprocessing architecture is proposed that can handle the input data at an average of one clock cycle per pixel while keeping the necessary data storage to a manageable level. The design approach taken is based on the observation that although the error diffusion technique provides no inherent parallelism within any given row, it still provides parallelism among different data columns on consecutive rows. A similar approach was discussed in [6] for a media processor implementation. Figure 2 is used to illustrate this point. The letters A through H denote individual pixels within an image. When pixel C is processed, the error is diffused to surrounding pixels D, F, G, and H. Since the error value for pixel E has already been fully calculated before processing C or any other pixel following C on the same row, pixel E can be processed as soon as its code value is acquired.

The multiprocessing approach adopted for this design utilizes this type of parallelism by processing pixels along the diagonals. A number of processing elements are used to process multiple pixels at a time, each working on a different row and column of the image. Since each processing element takes multiple clock cycles to process a pixel and one result per clock cycle is desired, the number of processing elements must be chosen to equal the number of clock cycles per processing element. For an N element system and an image width W , each individual processor can sequentially handle W/N pixels during the time it takes for a full image row to be clocked into the image processor. In other words, a single processing element can process $1/N$ times the input data rate.

Figure 3 illustrates the process with the first four rows of an image using four processing elements. Note that each lettered block in the figure represents a block of pixels within a single row and the four columns represent the entire image width. All of the pixels in the four rows are available to the processor for this snapshot. The grayed out boxes show the regions of the image which have fully processed results and the unshaded boxes show the regions of the image that have partially processed results or no error results calculated. At this point, the first processing element has completed the first row, the second element has completed E, F, and G, the third element has completed I and J, and the fourth element has completed M. Since the first row is complete, its processing element is free to take on the next available data row. As the next image row is acquired, the first element handles the new data, the second processor completes H, the third processor completes K, and the fourth processor completes N.

There are various design impacts that must be considered when choosing the number of processing elements, or in effect, the number of allowed clock cycles for processing each pixel. More processing elements result in higher storage space requirements as well as interconnect logic, but would also allow for more computationally intensive algorithms to be implemented within the elements. Four clock cycles provide a reasonable tradeoff among required storage, architectural complexity, and allowed algorithm processing time for this design. Four clock cycles at an 80 MHz clock rate allow each processing element 50ns to complete each computation. This is sufficient to allow for a very wide range of image processing algorithms within the Floyd-Steinberg error diffusion family.

3. HIGH-LEVEL ARCHITECTURE

The high-level architecture of the system is shown in Figure 4, and is composed of three main components: the processors, the process controller, and the output controller. The processors perform the actual image processing computations. The process controller is responsible for delivering data to each of the processors and routing the output data to the output controller. The output controller is merely responsible for organizing and delivering the output data to the user. The interesting aspect of this high level architecture is that the processors take advantage of the fact that they can always deliver their error data to the next processor in line and always receive their error from the previous processor in line. This is due to the fact that the processors always begin new rows of data in the same order. Processor #1 handles rows 1, 5, 9, 13, etc, processor #2 handles rows 2, 6, 10, 14, etc.

The processors are designed to operate on a single image row at a time, distributing their error to the next processor in line. As an image row is accumulated in the processor cache, the processing engine operates on the data it already has available. Every four clocks will produce a single result and four error terms. One of the error terms stays within the processor, but the other three terms need to be accessed by the next processor in line. The processors will accumulate error terms regardless of whether or not the processor has received any input pixels. Of course, this case only applies in the first few rows of an input image.

The process controller has the very simple task of routing I/O signals to and from the internal processor units. Result values are simply routed to the next higher level unit, but the input pixels must be sent to the correct processor. This is achieved by keeping track of row and column information on the incoming image data. The row is essentially a state variable indicating which of four processors the current input data needs to be sent to. The column is needed to change the processor to which the input data is being routed.

The final piece of the architecture shown in Figure 4 is the output controller. It is necessary to produce a steady output stream in the same order as the input data. Four rows of output data storage at three bits per data word are used to perform this task. No output data is produced until the third image row has been completely acquired. Starting early in the fourth row when all processors are running, one result per clock cycle will be output. This is necessary in order to produce output data in the same order as the input data. Extra clocks after the image has been acquired are necessary to flush all results from the system. When the output data has been fully sent, the output unit will simply stop outputting data, indicating that the image processing is complete.

4. PROCESSOR ARCHITECTURE

The processor encompasses the majority of the system functionality. It receives a data input stream, stores the data until it is ready to be used, processes the data, and returns the result and error values. There are four main components to this unit, as shown in the block diagram of Figure 5: an I/O block, an error control block, a cache block, and a computation block. A data stream oriented approach was taken for this design. The clock and reset signals are routed to each of the sub-units, but otherwise there are no high-level processing control signals. The system is self-regulating based on the input clock and reset signals. All processing by each of the components is based only on received data. The units start processing when they receive data and stop processing when all received data has been consumed.

All data coming to and from the processor goes through the I/O block. The I/O block is simply a data router. The clock and reset signals are distributed to all other processor units, error values are sent to the

error control block, which then assembles and forwards the values to cache, input pixels are sent directly to the cache, and output results are received from the computation block and sent to the next higher level unit. The I/O block also distributes the image width to the cache and engine units.

The error control block consolidates the error terms received from another processor. Three terms are received from every computation, and represent the distributed error values for each of three different image locations. With the exception of the first and last image row locations, each location will have three error terms associated with it; one for each of the three locations directly above it in the image. The three terms will arrive at consecutive inputs, making the addition of the values necessary before they can be used to process the location. This control block simply accumulates the terms into three separate error registers that are a moving window of accumulated error values for the three image locations. By accumulating these values before sending them to the cache unit, one complete value representing the total error from the previous image row for an image location can be sent to the cache for storage and processing.

The cache block does a considerable amount of work for the processor. The data stream organization is shown in Figure 6. There are two distinct components to the stream handler: the input side and the output side. Each has its own controller to determine keep track of high level information such as the current position of the data within the image row in order to properly store and retrieve data from the memory device. The image width is used for this purpose. Addresses are rolled back to zero after their values have reached the width of the image being processed. In software terms, the cache is a circular buffer with data entering and exiting the buffer simultaneously at different points. It is important to note that the pixel and error input values are two separate data streams for this architecture and each utilizes its own memory device. The block shown in Figure 6 is essentially doubled due to the second data stream.

On the input side, the cache unit takes the error and pixel values and stores them in memory. Separate input address registers are kept for each of these data streams and are incremented for each new input value that comes in. The register addresses always begin at zero since the processor always receives data that starts at the beginning of an image row. On the output side, the cache sends the stored pixel and error values to the engine for processing. The control signals for sending the output data are all generated internally within this unit. At the beginning of each row of data, the error terms will always precede the input pixels. This is due to the fact that the processor taking care of the previous image row is ahead column-wise of the current processor by 25% of the image width. This means that any given processor will have 25% of its error values for a given image row stored before any input pixels are received. As the input pixels arrive, they arrive at a rate of one per clock cycle while the error terms that are generated by the previous processor are only generated at a rate of one for every four clock cycles. As a result, the addresses of the input values overtake the error values halfway through the input row. The output side of the cache unit works very simply. When the input pixel stream begins to arrive at the cache, we know that the error terms have already been received, except of course in the case of the first row of data where the error terms simply do not exist. When the input pixel stream arrives, we can immediately begin sending the pixel and error values to the processing engine at a rate of one for every four cycles. When the last 25% of the row is being processed, the previous processor is now working on the first 25% of a new data row and storing those error values to this processors cache. It is important to note that as soon as the error input address register reaches the image width, the value is immediately reset back to zero and will continue accepting error values at the start of the error memory. This works because that part of the error memory has already been processed and the values that are stored there are no longer meaningful.

The processing engine adheres to the data stream processing approach. Like the cache, it is given the width of the image so that it can internally keep track of the number of input terms to expect in each image row and it simply waits for the first input data term to arrive. Once the first arrives, the rest of the input terms for the entire image are to arrive at a rate of one for every four clocks. Every four cycles, the engine flags its output result and error term to be valid and grabs the next set of inputs to be processed.

The error term destined for the next pixel within the same image row is stored in an internal register to be carried between computations. When the processor counts to the image width a new row of data is being processed, so the engine discards the next pixel error register in this special case. If a valid input is received at this time, counting is begun for a new image row, otherwise the unit has completed its work for the current image.

5. ALTERNATE CACHE CONFIGURATION

The design works very well in its present form, as described above. However, there is one relatively minor change that can be made in order to significantly improve it. The cache unit can be replaced with a new version that uses less memory by combining the pixel and error terms into a single memory device. The above method uses two separate memory devices of 24 bits per pixel each, while the alternate method uses a single memory device of 27 bits per pixel due to the fact that the error terms will be combined with the pixel values and will now require 9 bits for each color plane. The operation of the cache changes significantly since incoming pixel and error values must be written to the same memory device, creating a read-modify-write memory bandwidth problem.

A state machine controls the input values coming into the cache. The error and pixel terms must be written to the same memory device and it is not desirable to keep track of which value is written to each memory location first, so it is simpler to keep all unused memory locations cleared. When a value needs to be written, the location is first read, the values are added, and the sum is written back to memory. When output values are read from memory, it is then necessary to clear the memory location in order to make it ready for the next input value to be stored at that particular location.

This method greatly increases the necessary number of read/write accesses to memory, so it becomes necessary to widen the memory word size to alleviate this bottleneck. For our purposes, a word length of four pixels works well. Input values are stored in temporary registers until four of them are available for writing to memory. Likewise for the output values, a full word at a time is grabbed from memory and the values put in registers until all four values have been sent to the computation unit.

6. RESULTS AND DISCUSSION

The system architecture was designed in VHDL and simulated using the Aldec-HDL tool for simulation. System performance was checked using Synplicity for synthesis and Xilinx ICE for a place and route tool. The design works effectively, limited only by memory capacity of the target FPGA. With the original design, the memory required for an image that is 5100 pixels wide is 1,040,400 bits. This value is computed using eight rows of pixels stored within the cache devices at 24 bits per pixel and four rows of results stored within the output unit at 3 bits per pixel. Using the alternate cache design, the required memory for the same width image is reduced to 612,000 bits.

We have proposed a novel architecture for high-performance image processing that can be adopted for real-time error diffusion processing of gray scale and color images. There are many advantages to this image processor design and its implementation using an FPGA. The processor achieves the performance required in extremely fast image scanning systems. The target FPGA hardware allows for considerable scalability since any number of processing algorithms can be resident in the system, and new algorithms are implemented with a simple software download. The processing elements in this error diffusion architecture are modular can be easily replaced, which means that improved algorithms within the error diffusion family can be implemented with a relatively small software change. Additionally, an easy

migration path exists to newer and faster programmable logic devices when they become available in the future.

7. REFERENCES

- [1] R. Ulichney, "Digital halftoning", MIT Press, Cambridge, Massachusetts, 1987.
- [2] H. Kang, *Digital Color Halftoning*, SPIE press, 1999.
- [3] D. Lau and G. Arce, *Modern Digital Halftoning*, Marcel Dekker, New York NY, USA 2001.
- [4] A. DeHon, "The Density Advantage of Configurable Computing," *IEEE Computer Magazine*, pp. 41-49, Apr. 2000.
- [5] S. Hauck, "The role of FPGAs in Reprogrammable Systems," *Proc. IEEE*, pp. 615-639, Apr. 1998.
- [6] J.W. Ahn and W. Sung , "Multimedia Processor-Based Implementation of an Error-Diffusion Halftoning Algorithm Exploiting Subword Parallelism," *IEEE Trans. Circuits and Systems for Video Technology*, pp. 129-138, 2001.

Figure 1. Floyd-Steinberg Error Diffusion

Figure 2. Error Diffusion Processing Snapshot

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Figure 3. Processing Snapshot After Four Input Rows

Figure 4: Image Processor Architecture

Figure 5: Processing Unit Architecture

Figure 6: Cache I/O Stream Architecture