原文 2

PARALLEL IMPLEMENTATION OF AN ERROR DIFFUSION HALFTONING ALGORITHM WITH A GENERAL PURPOSE GRAPHICS PROCESSING UNIT

Becksang Seong, Jaewoo Ahn and Wonyong Sung
School of Electrical Engineering Seoul National University

Abstract: General purpose graphics processing units (GPGPUs) contain many execution units, thus they are very attractive for high speed image processing. However, the error diffusion halftoning algorithm can hardly exploit the benefit of massively parallel processing architecture because this algorithm uses feedback of the output error as well as the results of neighboring pixels. In this study, pixels that can be processed without dependency are found by examining the dependency graph. Also, a parallel processing method requiring less synchronization overhead is developed by considering the characteristics of GPGPUs.

Index Terms: Halftoning, error diffusion, general purpose graphics processing unit, parallel image processing

1 INTRODUCTION

General purpose graphics processing units, such as NVIDIA GTX 285, can accelerate many image processing kernels employing thousands of threads. However, the error diffusion algorithm [1], which is a digital halftoning technology that converts a multi-level image to a binary version, employs feedback, and as a result, it is difficult to parallelize this algorithm. Fig. 1 shows the block diagram of the well-known error diffusion process. There are previous researches on the parallel implementation of the error diffusion algorithm that include SIMD (Single Instruction Multiple Data) architecture based design, where three parallel processing approaches were compared [2], and MasPar architecture based one [3]. However, GPGPUs need parallelism that is greater than a few hundreds, while conventional parallel architectures such as SIMD, VLIW (Very Long Instruction Word) or superscalar usually support parallelism no greater than 16 [4]. Parallel implementation of the error diffusion algorithm also demands synchronization due to strict processing order of pixels. Synchronization between different streaming multiprocessors (SMs) in GPGPUs is difficult and slow, thus it is very needed to reduce the number of synchronization operations as much as possible.

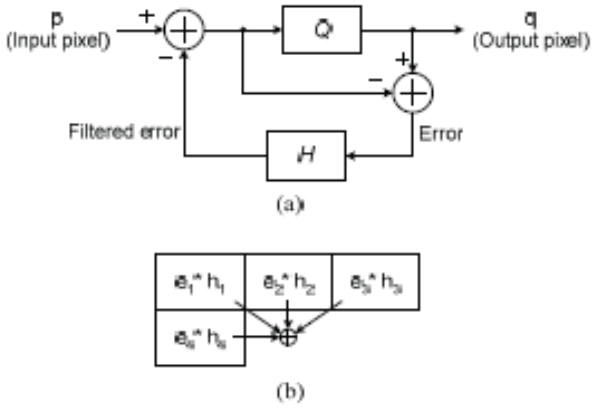

Fig. 1: (a) Block diagram of the error diffusion algorithm. (b) Floyd-Steinberg filter kernel, H

We have used the NVIDIA GTX 285 hardware and CUDA (Compute Unified Device Architecture) [5]. The hardware contains 30 SMs, and each SM contains eight processing elements. Assigning at least

32 threads, which is called a warp, for each SM is needed to fully utilize all the pipelined processing elements. It is also necessary to assign multiple warps to each SM to absorb the latency of the global memory access by task switching. The threads that are executed at the same SM are called the thread-block. Since a thread-block can accommodate multiple warps and each warp consists of 32 threads, a thread-block usually contains a multiple of 32 threads. The GPU supports two levels of synchronization. The local synchronization among the threads in the same thread-block can be fast because of the specialized hardware support, however that among thread-blocks is slow since it needs to be conducted through the global memory (DRAM). There are four types of on-chip memory in each SM: registers, shared memory, texture cache and constant cache. Data in the shared memory can be accessed by the threads in the same thread-block. The access of on-chip shared memory is also much faster than that of the global memory. A program manages the device memory (DRAM) space that includes global, constant and texture memory space through calls to the CUDA at the runtime.

In this study, the error diffusion algorithm shown in Fig. 1 is implemented utilizing thousands of threads in the GPGPU. Although there are several other error diffusion algorithms that are friendly for parallel processing hardware [6]–[8], the error diffusion algorithm shown in Fig. 1 is used because of its wide acceptance in digital printing industry. In order to remove the dependency among the threads, the skewed scan-line (SSL) technique [2] that exploits the independence of skewed pixels in adjacent lines is used. We reduce the synchronization overhead by conducting the slow inter-thread-block synchronization only at the beginning of a thread-block operation. The load imbalance problem of the skewed scan-line technique is also addressed.

Section 2 shows the GPU based parallelization and implementation techniques of the error diffusion algorithm. The implementation results are shown in Section 3, and concluding remarks are followed in Section 4.

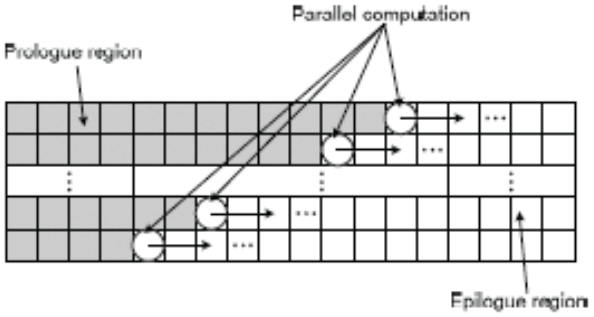

Fig. 2: A strip of an image assigned to a thread-block and independent pixels in different scan-lines of the strip

2 PARALLEL IMPLEMENTATION WITH GPGPU

2.1 Parallel algorithms

Since a GPU can support a large degree of concurrency, it is not appropriate to try to find out parallelism while processing a single pixel, where only a few arithmetic operations for the error diffusion filter kernel, H in Fig. 1 (b), can be conducted simultaneously. It is very needed to process multiple pixels at a time. The speculative quantization (SQ) method [2] that processes multiple pixels in the same scan-line can be applicable to GPGPU architecture. Although this method has the advantage of perfect load balance, the time for the compensation step increases as the degree of parallelism goes up. This approach also demands much synchronization between thread-blocks.

We employ the modified version of the skewed scan-line method [2]. In the error diffusion algorithm, it seems that the pixels in line $i+1$ can only be processed after completing the line i. However, if we carefully look at the algorithm, when the pixel j in line i is processed, the pixel '$j-2$' in line $i+1$ can be processed concurrently. This is because the pixel '$j-2$' in line $i+1$ needs the error value of pixels in $j-3$,$j-2$, and $j-1$ of the line i. Thus, it is possible to process the pixels in (i, j),$(i+1, j-2)$,$(i+2, j-4)$,

$\cdots$, concurrently, where (i, j) denotes the j-th pixel in the i-th line. When the image width is W, this method can accommodate as large as $W/2$ threads. Considering that the image width of most printed material is more than a few thousands pixels, the degree of parallelism that this method can provide can be considered enough. However, this method has two weak points; one is the load imbalance caused by the skewed processing and the other is large synchronization overhead. In the prologue region shown in Fig. 2, not all the threads can be utilized. At the same reason, this method needs the epilogue region where the threads that have started early should wait for other threads. Considering that $W/2$ threads are used for the image width of W, the efficiency due to load imbalance is 50%, which is however the low bound of the efficiency. A more serious thing is the synchronization problem. Since the inter-thread-block synchronization is slow, we need to reduce its amount as much as possible. For this purpose, the image is divided into multiple strips, and each strip is assigned to each thread-block. Thus, the height of a strip corresponds to the number of threads in a thread-block, which is 32 or higher in this implementation. When processing pixels inside of each strip, synchronization is conducted through the hardware support. However, processing the first line of a strip needs synchronization with a thread of the other thread-block as illustrated in Fig. 3. Note that the inter-thread-block synchronization needs global memory access and is slow.

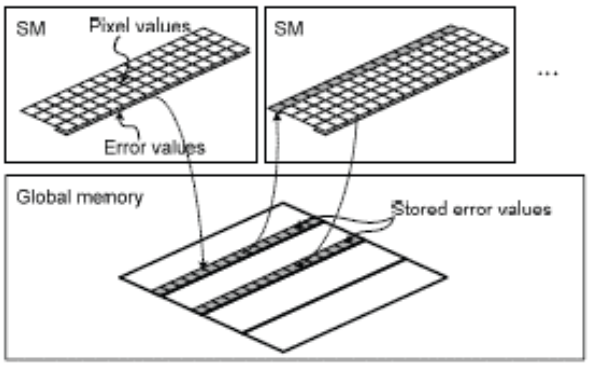

Fig. 3: Exchange of the error values between thread-blocks

A strip of an image assigned to a thread-block is shown in Fig. 2. The gray boxes represent already processed pixels, and the circled boxes are pixels currently being processed. One thread in a thread-block manages one scan-line of the strip. As being synchronized with the other threads, a thread sequentially processes pixels in a scan-line. Note that the synchronization between threads within a thread-block is compulsory. If any pixel (except for the uppermost pixel) of a skewed group is not synchronized with the others, it cannot obtain the correct error value. Unlike the synchronization between thread-blocks, the cost of thread synchronization within the same thread-block is not expensive, consuming only 4 cycles [5]. However, inter-thread-block synchronization operations are also needed in this method because the pixels in the next strip should start as soon as possible when the needed error values in the current strip are computed. This inter-thread-block synchronization is conducted through message passing using global memory access. The number of inter-thread-block synchronization is minimized by conducting the inter-thread-block synchronization quite seldom, for example, at the beginning pixel of the first line of a slice. Instead, some time margin, which is determined by experiment, is given between the processing of inter-block threads so that the thread for the next strip should not violate the dependency.

There is a trade-off among the load imbalance, synchronization overheads, and the number of global memory access operations. If we increase the height of a strip, which means that more threads are used for each thread-block, the load imbalance becomes larger. However, this reduces the number of global memory access operations and also the overhead of inter-thread-block synchronization.

2.2 Implementation

In the GPU based implementation, a thread loads its own input pixels from the global memory, and computes the error value of the pixels using other error values stored in the shared or global memory. Then it computes the value of the output pixels, and finally stores them to the global memory. It is very important to minimize the number of the global memory accesses in the process because it takes at least 400 to 600 clock cycles to read a data from the global memory [5]. To share error values between thread-blocks, they have to be stored in the global memory. However, not all error values, but only the error values of the last line of a thread-block need to be stored in the global memory so that the next thread-block can use them (Fig. 3).

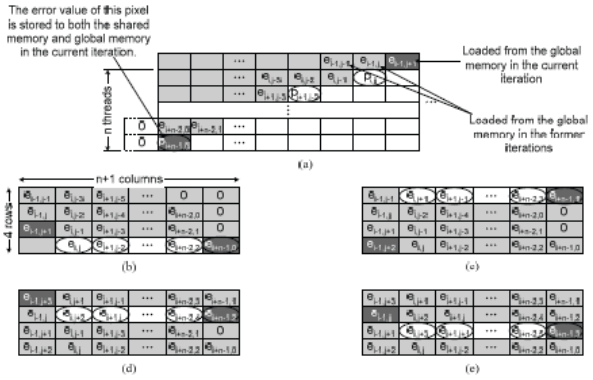

Fig. 4: Pixels and computed error values in a thread-block and contents of the array in the shared memory

Fig. 4 represents the usage of shared memory for the GPU based implementation of the error diffusion algorithm. Fig. 4 (a) shows one slice of an image that is being processed by a thread-block. The light gray boxes are processed pixels that have the error values, while the circled boxes, $p_{i,j}$,$p_{i+1,j-2}$,$\cdots$,$p_{i+n-1,0}$, are the pixels being quantized in the current step. In other words, these pixels form a skewed line that is processed simultaneously. Processing $p_{i,j}$ needs the error values $e_{i-1,j-1}$,$e_{i-1,j}$,$e_{i-1,j+1}$ and $e_{i,j-1}$, where $e_{i-1,j+1}$ needs to be newly read from the global memory while the other error values are stored in the shared memory. Fig. 4 (b) (e) show the error matrix stored in the shared memory, whose size is four rows by $(n+1)$ columns, where n is the number of threads in a thread-block. As shown in Fig. 4 (b), the newly produced error values in this iteration are stored at the fourth row, which are $e_{i,j}$,$e_{i+1,j-2}$,$\cdots$,$e_{i+n-1,0}$. For producing $e_{i,j}$ that is stored in the fourth row of the second column, the error values in the first column, $e_{i-1,j-2}$,$e_{i-1,j}$,$e_{i-1,j+1}$ and the one just above, $e_{i,j-1}$, are used. As shown in this figure, this L-shaped access pattern is equally applied to the other pixels in the same skewed line. There is no bank conflict because it is a linear addressing with the stride of one 4-byte word. After quantizing the current pixels and also obtaining the error values, the next skewed line, which consists of $p_{i,j+1}$,$p_{i+1,j-1}$, $\cdots$,$p_{i+n-1,1}$ are processed. Then, the error values for the next skewed line is stored at the first row of the error matrix in the shared memory as illustrated in Fig. 4 (c). In other words, the error values are stored in a cyclic buffer scheme where the row next to the last row is the first row. Since the values in the first row in Fig. 4 (b) will not be reused any more, as illustrated in Fig. 4 (c), the new error values that are $e_{i,j+1}e_{i+1,j-1}$,$\cdots$,$e_{i+n-2,3}$ and $e_{i+n-1,1}$ can be stored to that position. In further iterations similar process continues, updating the contents of the shared memory (Fig. 4 (d) and (e)). After the computation of error values corresponding to the circled boxes in Fig. 4 (e), the relative position of the error values in the array is equal to that of Fig. 4 (b), creating one cycle.

When all the thread-blocks finish their computation, we can finally obtain a halftoned image. However, as mentioned earlier, a thread-block has the dependency on the previous thread-block and thus, it is required to synchronize them. Fig. 5 represents a whole image assigned to N/n thread-blocks, where N is the height of an image and n is the number of threads in a thread-block. During the process, the next thread-block waits until the previous thread-block reaches a specific pixel location of the image, m. When the previous thread-block reaches there, it stores a specific value to the global memory space. The

next thread-block loads the value continuously from the beginning of the kernel, and when it confirms that the value was changed by the previous thread-block, it starts its process. This synchronization continues until the last thread-block starts. When the image width is small or the margin, m, is large enough, only a single synchronization operation at the starting-point is needed for each thread-block. Since the number of synchronization operations is small, it does not incur much overhead.

3 IMPLEMENTATION RESULTS

For the performance measurement of our implementation, we compared the execution time of two implementations: the conventional C program running on a CPU and our CUDA program for a GPU. The platforms we use are Intel Core 2 Quad and NVIDIA GTX 285 that has 240 available cores. The clock speed of the Core 2 Quad CPU is 2.83 GHz, and that of the GTX 285 GPU is 1.48 GHz. The operating system is Linux Fedora 10, and we use GCC for compiling the conventional C code and NVCC for the CUDA code.

In the experiment, we have a few parameters that would have influence on the execution time. The first parameter is the size of a thread-block that would affect the load balance, number of global memory accesses, and inter-thread-block synchronization overhead as explained earlier. The second parameter is m, which is the margin to avoid frequent synchronization. This parameter depends on the size of the thread-block and the image width, which was obtained by experiment. When the image size is 1K by 1K pixels and the number of threads is 32, m is chosen as 40. It is evident that we can obtain a higher speed-up as m is getting smaller. But it is risky to set m as too small a number, since there is a possibility that the processing speed of a current thread-block is faster than that of its previous thread-block. For prudent operation with small m, we may check if the error values read from the previous slice are within the predetermined ranges.

Fig. 5: Synchronization between inter-thread-block

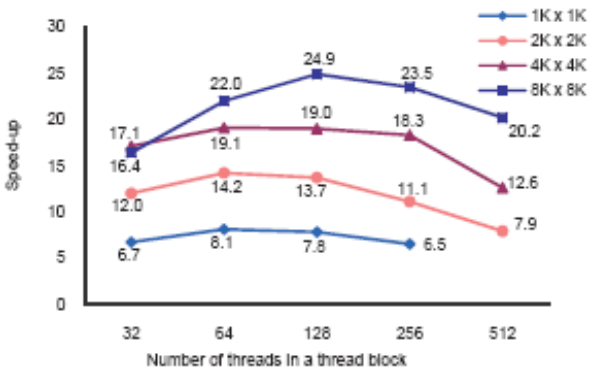

Fig. 6: Performance according to the thread-block size and image size

Considering those factors, we measured the speed-up with various sized images. Fig. 6 represents the performance according to the thread-block size and image size. As shown in this figure, the maximum

speed-up is obtained when the thread-block size is, in general, between 64 and 128, where the trade-off among the load balance, number of the global memory accesses, and inter-thread-block overhead is optimized. One more thing that we should consider is SM occupancy. The main reason of the performance degradation when the thread-block size is very large is the limitation of the on-chip memory resource because the memory size for each thread decreases as the number of threads increases. We can find that the speed-up is increased as the image size becomes larger. This is because we can utilize a GPU more efficiently by increasing the number of active threads in the case of a larger image.

When the data transfer time between the host and device are included, the speed-up is lower, since its proportion is quite large (Table I). For the image size of 8K by 8K pixels, the speed-up obtained excluding the host and GPU data transfer time is as high as 24.9, but is decreased to 9.8 when the data transfer time is included. However, this overhead is not significant when many applications are executed as a group of kernels in the GPU.

Table 1: Comparisons of the kernel execution time (sec) and data transfer time (sec) according to the image size (pixels)

Image size	**Kernel Exec. Time (a)**	**Data transfer time (b)**	**CPU time (c)**	**Speed-up c/a c/(a+b)**
1024×1024	0.004865	0.002212	0.039457	8.1 5.6
2048×2048	0.010066	0.007886	0.143056	14.2 8.0
4096×4096	0.026404	0.030981	0.505470	19.1 8.8
8192×8192	0.073270	0.112392	1.824052	24.9 9.8

4 CONCLUDING REMARKS

The error diffusion halftoning algorithm is implemented using thousands of threads on a GPGPU. The skewed scan-line method is used for finding independent pixels. The overhead is minimized by reducing the number of inter-thread-block synchronization and using the shared memory efficiently. The result shows that the developed technique can lead to a speed-up of about 24.9 when compared with the conventional C program on a PC.