A Parallel Error Diffusion Implementation on a GPU

Yao Zhang[a], John Ludd Recker[b], Robert Ulichney[c], Giordano B. Beretta[b], Ingeborg Tastl[b], I-Jong Lin[b], John D. Owens[a]

[a]University of California, Davis, One Shields Avenue, Davis, CA, USA;
[b]Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA, USA;
[c]Hewlett-Packard Co., 165 Dascomb Road, Andover, MA, USA

ABSTRACT

In this paper, we investigate the suitability of the GPU for a parallel implementation of the pinwheel error diffusion. We demonstrate a high-performance GPU implementation by efficiently parallelizing and unrolling the image processing algorithm. Our GPU implementation achieves a $10 - 30\times$ speedup over a two-threaded CPU error diffusion implementation with comparable image quality. We have conducted experiments to study the performance and quality tradeoffs for differences in image block sizes. We also present a performance analysis at assembly level to understand the performance bottlenecks.

Keywords: Halftoning, Error Diffusion, Parallel Processing, GPU Computing.

1. INTRODUCTION

With the ever-increasing printing resolution and page output rate, digital presses present high demands for the processing power of the imaging pipeline. On the other hand, today's massively parallel GPUs can potentially provide a high-performance and cost-effective solution to digital image processing.[1]

However, error diffusion, as a major stage in the printer imaging pipeline, is inherently serial in its original form.[2] Although there have been several efforts in adapting error diffusion for parallel processing,[3–8] most previous studies focused on the theoretical aspect of parallelization, and lack on actual implementation and performance analysis. In this paper, we investigate the suitability of the GPU for the pinwheel error diffusion algorithm developed by Li et al.[3] We demonstrate a high-performance GPU implementation by efficiently parallelizing and unrolling the image processing algorithm. We have further conducted experiments to study the tradeoffs between parallelism, performance, and image quality.

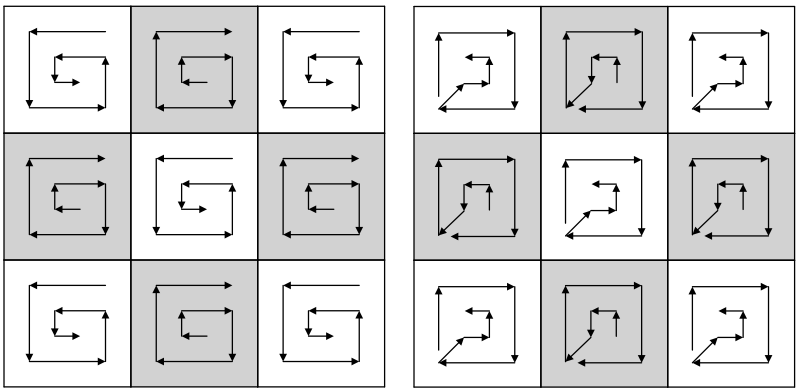
The rest of the paper is organized as follows. Section 2 reviews the pinwheel error diffusion algorithm. Section 3 describes the GPU implementation and performance considerations. Section 4 presents our results and analysis on performance and image quality. Section 5 concludes the paper and describes future work.

2. ALGORITHM REVIEW

Parallelizing error diffusion is a difficult task and usually comes with a compromise between performance and image quality. Metaxas[4] has attempted to parallelize the Floyd-Steinberg algorithm along the image diagonal in a wavefront fashion. As the amount of parallelism of this method is limited by the number of diagonal pixels, it is not well-suited for the massively-parallel GPU. Another problem of this method is that it requires global synchronization when processing diagonal pixels, which may hurt the performance. In contrast, the pinwheel algorithm by Li and Allebach[3] does not require global synchronization and is able to create far more parallelism by dividing an image into blocks and processing them in parallel. Given this consideration, we choose the pinwheel algorithm as a starting point for our GPU implementation.

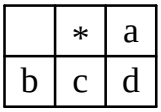
The pinwheel algorithm first divides an input image into two groups of blocks in a checker-board fashion as shown by the gray and white blocks in Figure 1, then respectively processes all blocks in each group in parallel with the gray group being processed first.

To process each block, we scan through all pixels of the block sequentially. The scan proceeds along a spiral or serpentine path as shown in Figure 1. The scan path goes outward for gray blocks, and inward for white blocks. Note that during the processing of gray blocks, we diffuse errors from the gray blocks to the white blocks

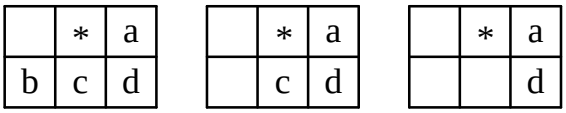


(a) Spiral. (b) Serpentine.

Figure 1. Block interlaced pinwheel error diffusion with spiral or serpentine scan path. The gray blocks are outward blocks. The white blocks are inward blocks.



(a) outward.



(b) inward.

Figure 2. Error-weighting matrices for outward and inward spiral blocks.

across block boundaries. To process each pixel, we first threshold the pixel value, then diffuse the quantizer error to its neighbors according to an error-weighting matrix.

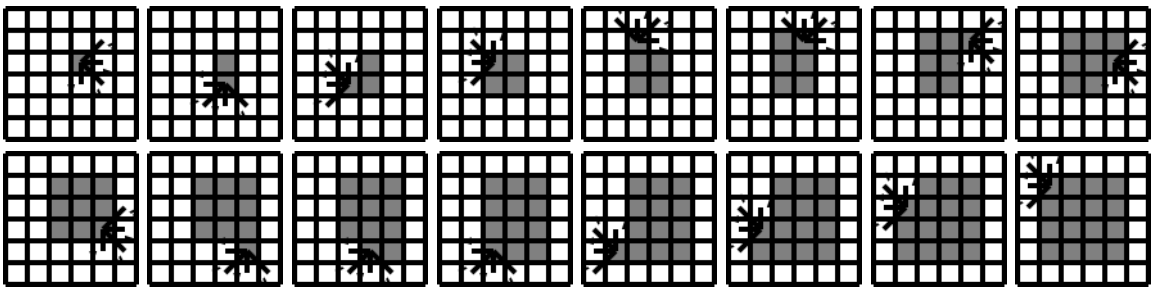
Inward spiral blocks use three types of error-weighting matrices (Figure 2(b)) respectively applied to pixels of different locations. Figure 3 shows where each error-weighting matrix is applied in the case of 4×4 image blocks. Outward spiral blocks share the 4-weight matrix of inward blocks for all pixels (Figure 2(a)) except the ones at corners, and use another set of 4-weight matrices for corner pixels. We use tone-dependent error-weighting matrices and quantizer thresholds for better image quality, which means we have 256 matrices for 256 gray levels respectively for each type of matrices.

3. GPU IMPLEMENTATION

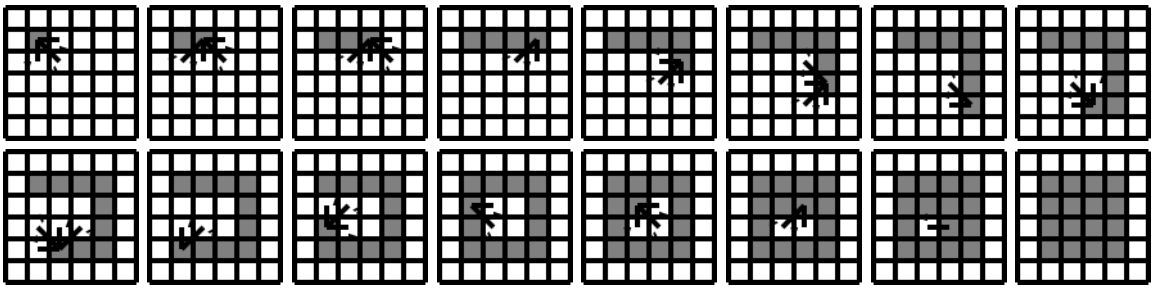
We have implemented the spiral pinwheel error diffusion on a NVIDIA GTX 460 GPU using the CUDA programming model.[9] The GPU is a hierarchical shared-memory vector architecture. GTX 460 has 7 multiprocessors, where each multiprocessor contains 48 thread processors and 64 KB of on-chip shared memory. CUDA virtualizes multiprocessors as blocks and processors as threads, which enables programmers to run thousands of threads and blocks across different generations of GPUs regardless of the number of physical processors.

We map each image block to one CUDA thread, and thus a number of image blocks to one CUDA block. We choose a CUDA block size of 64, which allows high thread occupancy on the GPU but still allows sufficient registers to be allocated to each thread.

Because we use both inward and outward spiral directions and three types of error-weighting matrices, the program control logic and memory addressing can be quite complex. One method to address neighboring pixels is to store their addresses in a look-up table. However, since the algorithm is already memory-intensive in thresholding pixel values and diffusing errors, we wanted to avoid extra address look-ups. To achieve this, we have written a program to unroll the scan loop automatically, so that all control logic and address look-ups are



(a) Outward.



(b) Inward.

Figure 3. Addressing pattern of each pixel for the outward and inward 4x4 image block.

eliminated. Figure 3 shows the automatically generated addressing pattern of each pixel for both the inward and outward image block.

We store all filter weights and quantizer thresholds in the 64 KB constant memory, which is cached in the 8 KB constant cache on the GPU. The total storage need for weights is [(4 + 3 + 2)(inward) + 4(outward) - 4] × 256 gray levels × 4 bytes = 9 KB. Because the sum of all weights is equal to 1, we can store one fewer weight for each weighting matrix (note that we subtract 4 for four weighting matrices). The total storage need for thresholds is 256 gray levels * 6 arrays * 4 bytes = 6 KB. We have 6 arrays of threshold values respectively for upper thresholds, lower thresholds, upper thresholds and lower thresholds at corners respectively for inward and outward blocks.

4. EXPERIMENTAL RESULTS

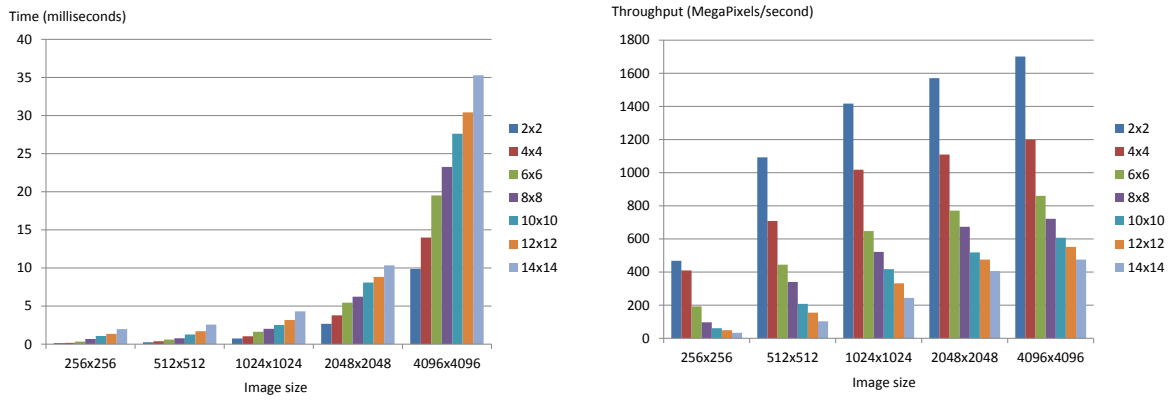
In this section, we present our results and analysis on performance and image quality.

4.1 Performance Results

Our test platform uses a 2.4 GHz Intel Core 2 6600 dual-core CPU, a GTX 460 graphics card with 1 GB video memory, CUDA 3.2 and the Windows 7 operating system. Figure 4 shows the GPU performance results in terms of timings and pixel processing throughput. For larger images, the processing time is nearly linear to the number of pixels (Figure 4(a)). For example, a 4096 × 4096 image takes almost 4 times longer to process than a 2048 × 2048 image. Figure 4(b) shows that processing throughput levels out for larger images, which better utilize the processing power of the GPU. We also note that bigger image block sizes lead to lower performance, but enjoy better image quality, as will be discussed in Section 4.3. For sufficiently large images (larger than or equal to 1024 × 1024), we achieved $10 - 30\times$ speedup compared to a two-threaded CPU implementation for relatively good image quality (using an image block size larger than or equal to 8 × 8) as shown in Figure 5.

4.2 Performance Analysis

To understand the program bottlenecks, we have measured and calculated the instruction throughput and effective bandwidth.



(a) Timings. The processing time is nearly linear to the number of pixels for larger images.

(b) Throughput. Processing throughput levels out for larger images.

Figure 4. GPU error diffusion performance for various image and block sizes.

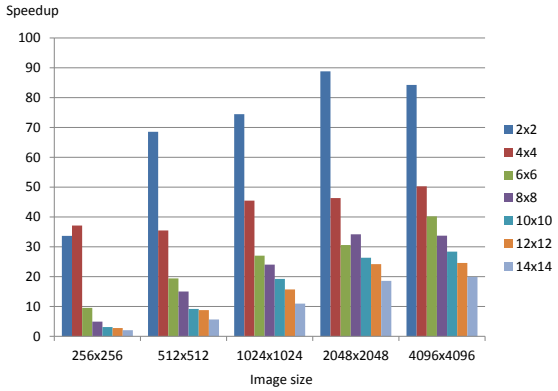


Figure 5. GPU error diffusion speedup compared to a two-threaded CPU implementation for various image and block sizes. For an image $\geq 1024 \times 1024$ and a block size $\geq 8 \times 8$, the speed-up is $10 - 30\times$.

Table 1 lists the instruction throughput for various image block sizes. We count the number of assembly instructions by using cuobjdump provided by the CUDA toolchain.[9] One observation is that the number of instructions per pixel remains relatively constant around 80 instructions/pixel. Combining this information with the measured pixel processing throughput, we obtain the instruction throughput. The theoretical peak throughput of the GTX 460 GPU is $numberThreadProcessors \cdot numberMultiprocessors \cdot processorFrequency = 48 \cdot 7 \cdot 1.3\,\text{GHz} = 436.8\,\text{GigaInstructions/sec}$. The measured instruction throughput for all image block sizes is far less than the peak value, which suggests that the program is not bound by instruction execution.

Table 1. Instruction throughput for various image block sizes. Image size: 4096×4096.

image block size	pixels/block	instructions	instructions/pixel	MegaPixels/sec	GigaInstructions/sec
2×2	4	325	81.3	1701.5	138.3
4×4	16	1360	85.0	1199.2	101.9
6×6	36	2818	78.3	859.9	67.3
8×8	64	5132	80.2	721.3	57.8
10×10	100	8275	82.8	607.4	50.3
12×12	144	12095	84.0	551.7	46.3
14×14	196	16091	82.1	475.5	39.0

Table 2. Memory reads/writes per pixel respectively for global memory, constant memory, and total count, and effective bandwidth. Image size: 4096×4096.

image block size	global memory	constant memory	total	Effective bandwidth (GB/s)
2×2	6.0	12.3	18.3	124.2
4×4	21.4	12.5	33.9	162.8
6×6	20.2	11.8	32.0	110.1
8×8	25.6	11.3	36.9	106.4
10×10	29.1	11.1	40.2	97.8
12×12	31.2	10.8	42.0	92.6
14×14	31.3	10.7	42.0	79.9

Table 2 lists the number of memory reads/writes per pixel and the measured effective bandwidth. The theoretical peak bandwidth of GTX 460 is $\frac{memoryFrequency \cdot busWidth}{8\,\text{bits/byte}} = \frac{3.4\,\text{GHz} \cdot 256\,\text{bits}}{8\,\text{bits/byte}} = 108.8\,\text{GB/s}$. The effective bandwidth is around the peak bandwidth, which suggests that the program is bound by memory bandwidth. Since the global memory access is cached, the effective bandwidth can be higher than the peak value. On the other hand, the GPU prefers to access entire continuous (coalesced) blocks of memory; otherwise the GPU does not fully utilize its peak memory bandwidth as only partial transfered data is used. Because the memory access to image blocks has a stride equal to the width of an image block, the effective bandwidth can be below the peak value for large image blocks, which leads to lower overall performance.

We also noticed that the CUDA compiler is not able to reuse the already allocated registers for each pixel, but keeps allocating new registers that spill to global memory. For larger image blocks, the compiler can exceed the spilling limit and thus produces an incorrect program, which is the reason why we have only been able to run an image block size of up to 14×14. Also, as shown in Table 2, because of the register spilling, the number of global memory reads/writes increases as we use larger image blocks. We see two possible solutions to this problem: (1) future compilers may fix this register allocation problem; or (2) replace the unrolled scan loop with the loop version of the program..

It is worth mentioning that we have also tried to utilize the on-chip shared memory to cache image blocks. However, the performance turns out to be 2x worse, which is not quite as expected. We see two reasons for the low performance: (1) because of the limited capacity of the shared memory, we can only fit 32 image blocks (32 threads, instead of the original 64 threads) to a multiprocessor, which leads to a lower thread occupancy and thus reduces the GPU's ability to hide memory latency; and (2) there is not enough data reuse. Since each pixel at most distributes its error to its four neighbors, the data reuse is at best only four times. However, the cost of using shared memory consists of a one-time data loading to shared memory and another one-time data offloading to global memory, which turns out to offset the benefit of already little data reuse.

4.3 Discussion on Image Quality

The input image used for our experimental results is a composite of a gray ramp, four constant patches, and a photo. The photo is one of the Kodak PhotoCD test images called motocross. Figure 6 shows the result of using our parallel algorithm on a GPU with a block size of 14×14 on a 512×512 pixel scaled version of our input image.

Figure 7 and Figure 8 detail cropped 512×64 pixel portions of the halftoned output using a 4096×4096 version of our composite input image for both the beginning of the gray ramp, and the photo. As expected, larger block sizes yield better quality results. Because our process is deterministic (non-random), a square block of size N can only possibly render $N^2 + 1$ gray levels. So our 8-bit source image requires a block of size 8×8 or larger to render all gray levels. Another consequence of using a deterministic process is that for fixed gray levels a repeating pattern will result with a period corresponding to the size of the block. While local variations in photo pixel values mask these periods, they are particularly apparent in the halftones of the ramp (Figure 7). We believe that using stochastic perturbations on the error filter weights[10] will significantly mitigate these periodic structures.

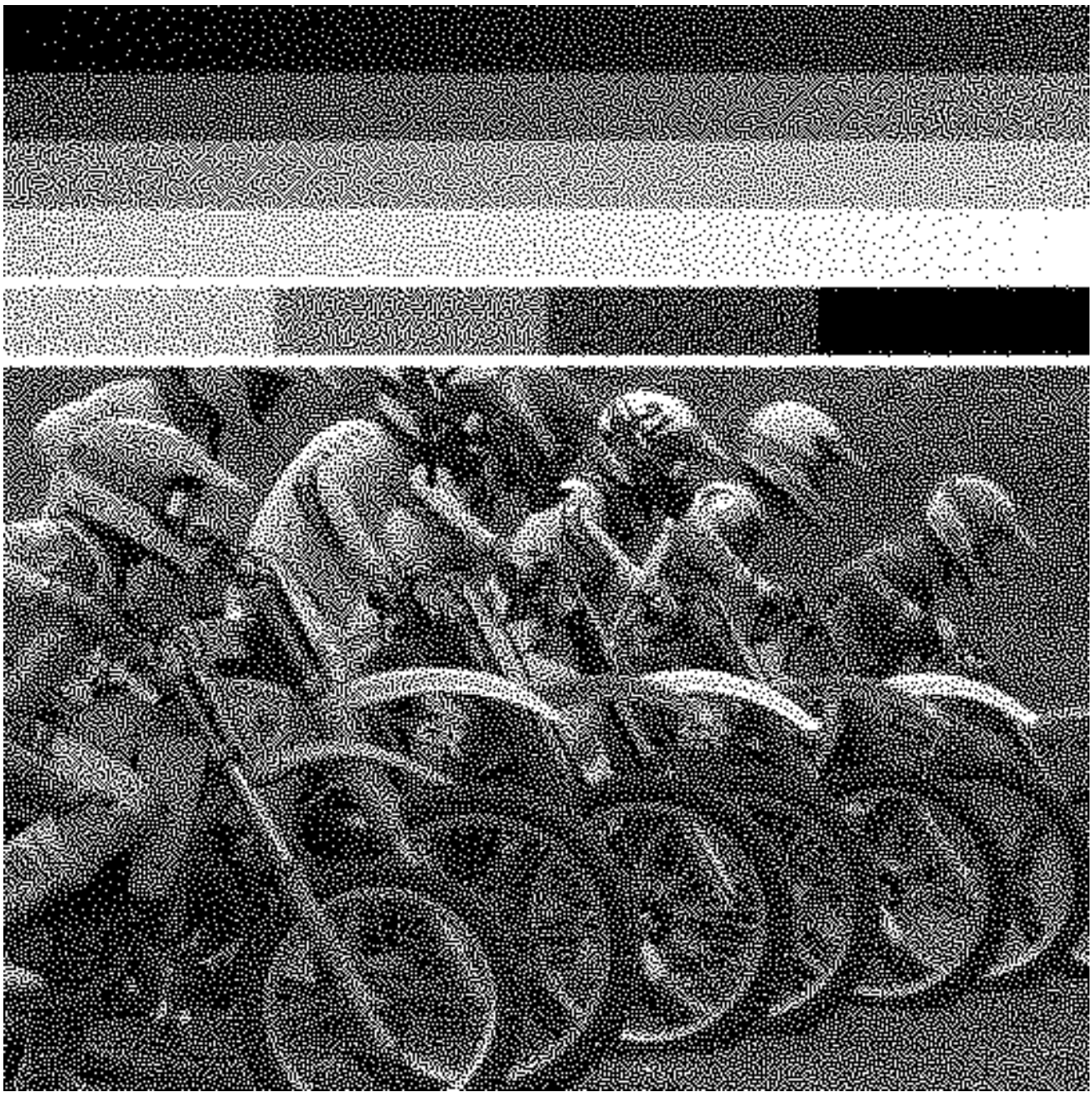


Figure 6. A 512×512 halftone image generated by our GPU pinwheel error diffusion implementation with an image block size of 14×14.

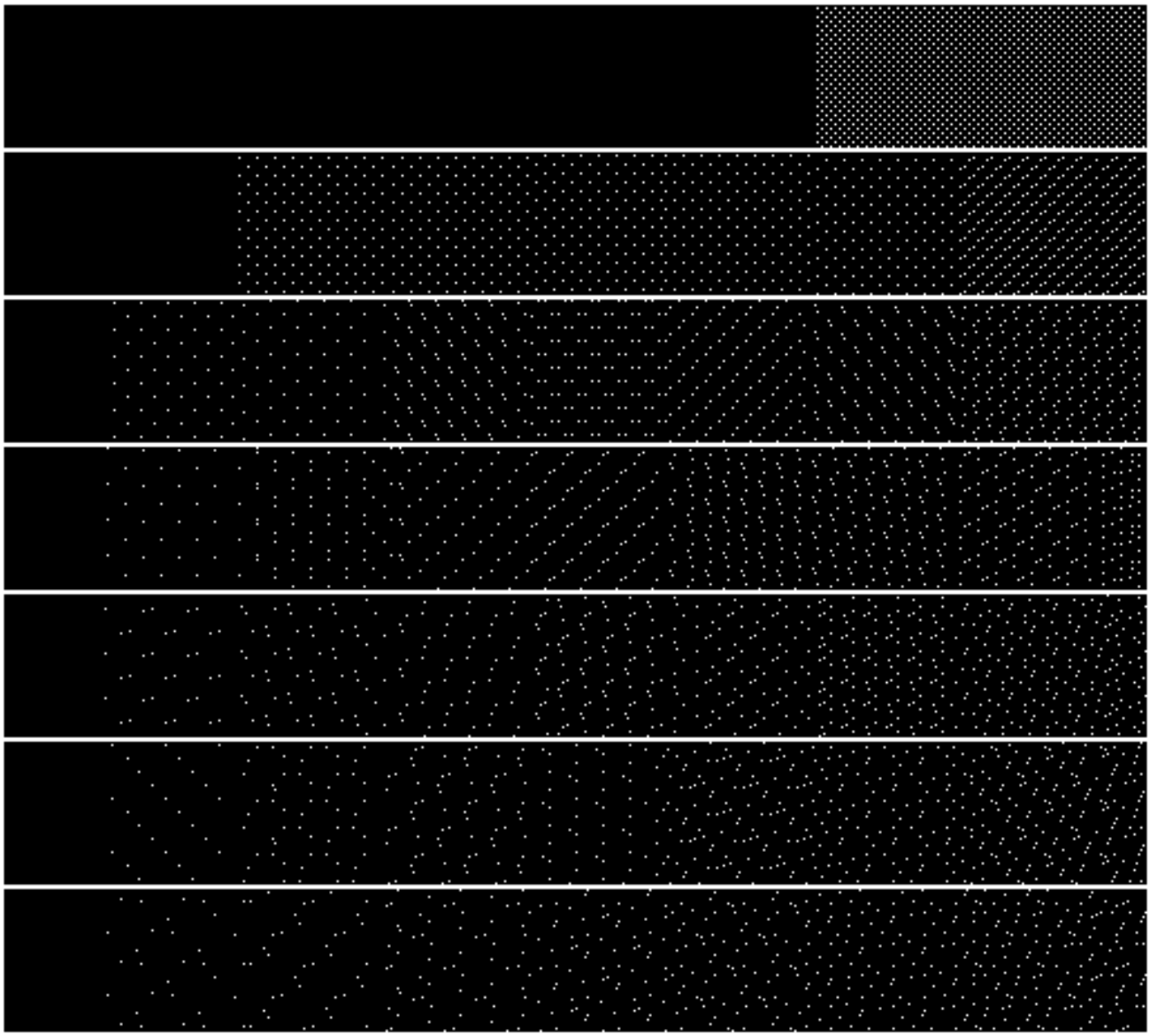


Figure 7. Comparison of image quality for various block sizes from 2×2 (top), 4×4, 6×6, 8×8, 10×10, 12×12, to 14×14 (bottom) for the ramp part of the image.

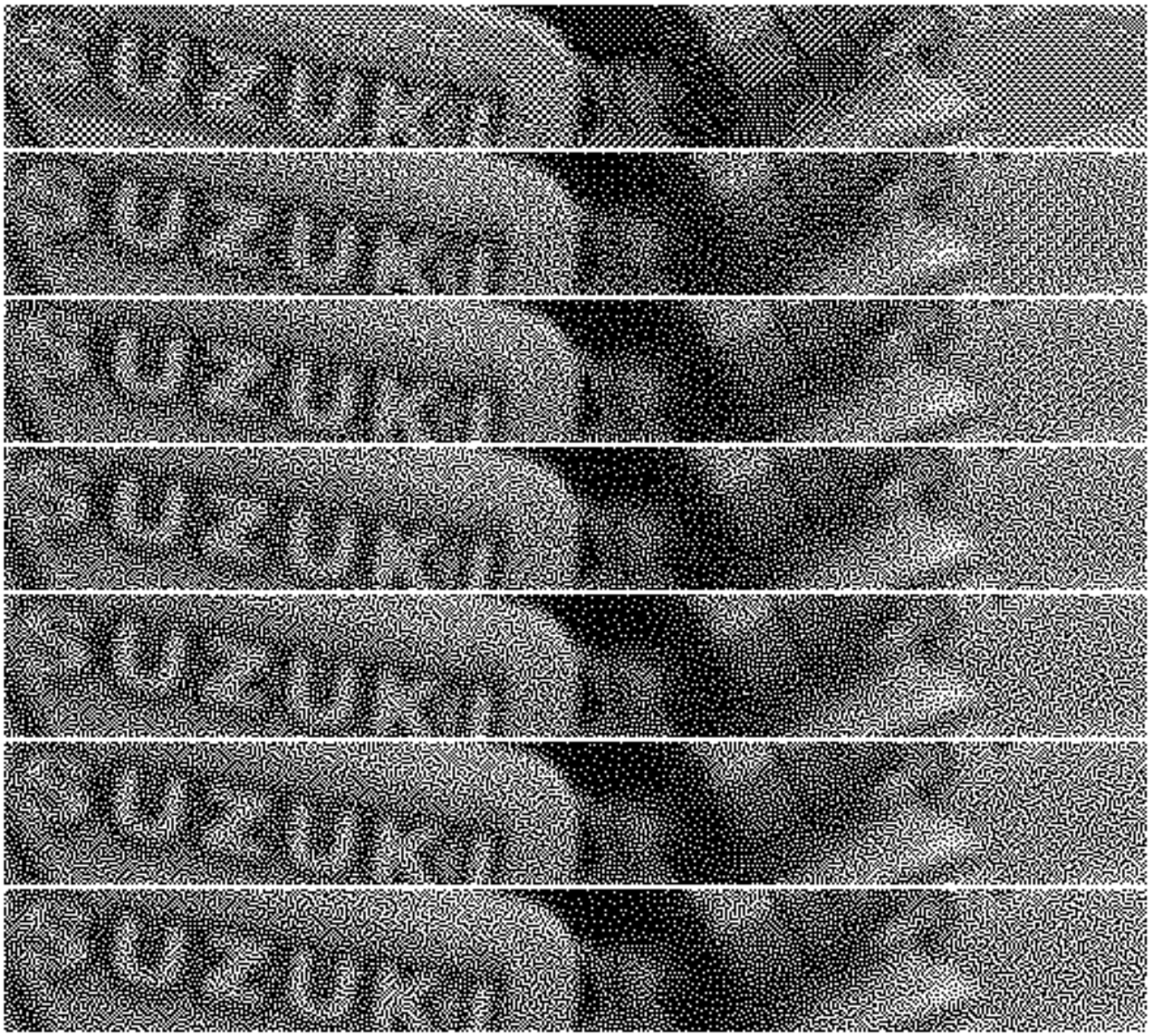


Figure 8. Comparison of image quality for various block sizes from 2×2 (top), 4×4, 6×6, 8×8, 10×10, 12×12, to 14×14 (bottom) for the photo part of the image.

5. CONCLUSION

We have demonstrated an efficient parallel implementation of pinwheel error diffusion on the GPU and showed good image quality. We should note that halftoning is just one stage in the printer imaging pipeline.[11] When this stage is performed on the CPU, the image must be written out from the GPU and then written back for the next stage. Therefore, while for the halftoning stage by itself we get a speedup of $10 - 30\times$, in reality for the entire pipeline the speedup is much more significant.

We see several directions for future work: (1) compare the performance between the unrolled and the loop implementation; (2) add randomization to improve image quality; (3) compare the image quality between using constant weights and using tone-dependent weights; and (4) generalize this work for color images.

REFERENCES

[1] "General-purpose computation using graphics hardware." `http://www.gpgpu.org/`.

[2] Floyd, R. and Steinberg, L., "An adaptive algorithm for spatial grey scale," in [*Digest of the Society of Information Display*], 36–37 (1976).

[3] Li, P. and Allebach, J. P., "Block interlaced pinwheel error diffusion," *Journal of Electronic Imaging* **14**(2), 1–13 (2005).

[4] Metaxas, P. T., "Optimal parallel error-diffusion dithering," in [*Proceedings of SPIE*], **3648**, 485–494 (1999).

[5] Zhang, Y. and Webber, R. E., "Space diffusion: An improved parallel halftoning technique using space-filling curves," in [*Proceedings of SIGGRAPH 93*], *Computer Graphics Proceedings, Annual Conference Series*, 305–312 (Aug. 1993).

[6] Zhang, Y., "Line diffusion: a parallel error diffusion algorithm for digital halftoning," *The Visual Computer* **12**(1), 40–46 (1996).

[7] Knuth, D. E., "Digital halftones by dot diffusion," *ACM Transactions on Graphics* **6**(4), 245–273 (1987).

[8] Takeuchi, Y. and Kunieda, H., "Space partitioning image processing technique for parallel recursive half toning," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* **E76-A**, 603–612 (Apr. 1993).

[9] NVIDIA Corporation, "NVIDIA CUDA C programming guide," (22 Oct. 2010). `http://developer.nvidia.com/cuda`.

[10] Ulichney, R., "Dithering with blue noise," *Proceedings of the IEEE* **76**(1), 56–79 (1998).

[11] Recker, J. L., Beretta, G. B., and Lin, I.-J., "Font rendering on a gpu-based raster image processor," *Color Imaging XV: Displaying, Processing, Hardcopy, and Applications* **7528**(1), 75280A, SPIE (2010).