外 文 翻 译

毕业设计题目：基于深度学习的高效半色调处理技术研究

原文 1：**4-Row Serpentine Tone Dependent Fast Error Diffusion**

译文 1：四行蛇形色调自适应快速误差扩散算法

原文 2：**Parallel implementation of an error diffusion halftoning algorithm with a general purpose graphics processing unit**

译文 2：基于通用图形处理器的误差扩散半色调算法并行实现

原文 1

4-ROW SERPENTINE TONE DEPENDENT FAST ERROR DIFFUSION

Y. Mao, L. Abello, U. Sarkar, R. Ulichney and J. Allebach
School of Electrical and Computer Engineering, Purdue University

Abstract: Error diffusion is a popular technique widely used in desktop printers, especially inkjet printers. However, since the conventional error diffusion is computed in raster order, it produces worm artifacts in the highlights and shadows. In addition, as a serial algorithm, it limits the efficiency and flexibility of hardware implementations. To address these two issues, we propose a novel serpentine based error diffusion algorithm that uses tone dependent error weights and thresholds. We also propose an expanded error weight location matrix to improve the halftone quality in the extreme tones. With this new algorithm, we achieve better halftones comparing to the original tone dependent fast error diffusion, especially in the quarter tones.
Index Terms: halftoning, modified tone dependent fast error diffusion, parallel implementation

1 Introduction

Digital halftoning is a method of creating the illusion of continuous-tone output through the use of a series of dots arranged differently in size or in spacing. Halftoning allows one to simulate various shades of color with one or two colors, so it is an widely used technique in rendering devices that are only capable of producing limited number of tone levels, for example printers and some displays.

According to the level of computational complexity, halftoning algorithms can be classified into three general categories: screening, error diffusion, and search-based methods. Since error diffusion renders better detail than screening while maintaining lower cost than search-based methods, it is the most popular algorithm for marking engine technologies that can stably render isolated dots, such as inkjet. In this paper, we will focus on error diffusion.

As originally proposed by Floyd and Steinberg [1], error diffusion is a neighborhood operation that moves through the input image in a raster order, quantizing each pixel in the scan line, and feeding the error ahead to the neighboring pixels that have not yet been binarized. Despite excellent detail rendition, error diffusion sometimes creates worm-like patterns and visible structures. To solve these problems, a number of derivations and modifications of error diffusion were developed in previous research, including use of alternative scan paths [2]-[4], threshold modulation [5]-[9], variable weights [4], [10]-[12], and tone dependent parameters [13], [14]. Reference [15] is of particular note, as it provides an excellent summary of recent methods based on directly training the weights to match a desired blue noise spectral characteristic, and proposes an improvement to this approach. It also incorporates the training of the thresholds to eliminate edge sharpening.

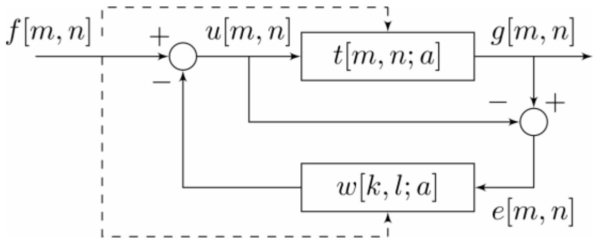


Fig. 1: Tone dependent fast error diffusion system.

Aside from generating visually unpleasant textures, another disadvantage of error diffusion is lack of locality. This means that when either a conventional raster or serpentine scan path [4] is used, we are not allowed to process a pixel until all pixels in its preceding scan line have been quantized. As a consequence, hardware must store the information associated with the states of the pixels that are spatially far away, which is inefficient. With Peano scan [2], a pioneer of parallel scan path, however, the output quality of error diffusion is not satisfactory. In fact, customers in the printing market base their judgment on both print quality and implementation efficiency. Therefore, it is extremely beneficial to enable the hardware to decide the binary output locally without losing quality. In this regard, we design a novel 4-row serpentine scan path. The novelty of our approach lies in using a compound of conventional raster and serpentine scan patterns, which greatly enhances hardware efficiency. Other prior work that incorporated the concept of a serpentine raster with novel error diffusion architecture include [16], [17].

In this paper, we apply the 4-row serpentine scan path with the tone dependent fast error diffusion (TDFED) [18] algorithm. To reduce worm-like textures, a tone-dependent 4-weight location matrix that diffuses errors further back along the next line is designed. To further refine the halftone outputs, the weights and thresholds values for each gray level are optimized in an offline training process based on a visual cost function developed in [19]. However, the primary focus of this paper is the development of a new error diffusion architecture that is suitable for efficient hardware implementation, rather than methods and cost functions for optimization of the weights and thresholds, which is the primary focus of [15]. In fact, the concepts introduced in this paper could also be deployed with weights and thresholds optimized according to the methods introduced in [15]. The rest of the paper is organized as follows: Section 2 demonstrates the 4-row serpentine TDFED algorithm. Section 3 discusses the experimental results. Section 4 draws the conclusions.

2 4-Row Serpentine Tone Dependent Fast Error Diffusion

We will start by providing an overview of TDFED in Section 2.1 and then present the details of the scan path. Section 2.3 will illustrate the expanded error location matrix. Lastly, Sec. 2.4 will discuss the training process.

2.1 Overview of Tone Dependent Fast Error Diffusion

Since we are presenting an error diffusion algorithm that is designed for monochrome printing devices, the pixel value is represented in units of absorptance $0 \leq a \leq 1$, where 0 corresponds to white, and 1 corresponds to black.

Figure 1 illustrates the block diagram of the TDFED system. In this figure, $f[m,n] = a$ is the pixel absorptance of the continuous-tone image, $u[m,n]$ is the updated pixel value, and $g[m,n]$ is the binary output. Unlike Floyd Steinberg error diffusion, the thresholds $t[m,n;a]$ and error weights $w[k,l;a]$ of TDFED depend on the input absorptance, where k,l are the relative position indices indicating the location of the neighboring pixels of $f[m,n]$. The neighboring pixels of $f[m,n]$ are defined by Floyd and Steinberg to be on its right, lower right, below, and lower left, assuming the image is scanned from left to right in a raster order. Since TDFED moves through the input image in a serpentine raster order, a mirror image of the weight location is adopted when scanning from right to left.

The binary output of the system is determined by thresh-olding the updated pixel value:

$$g[m,\ n] = \begin{cases} 1, & \text{if } u[m,\ n] \geq t[m,\ n;a], \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

The updated continuous-tone pixel value $u[m,n]$ is computed as:

$$u[m+k,\ n+l] \leftarrow u[m+k,\ n+l] - w[k,\ l;a] \cdot e[m,\ n], \tag{2}$$

where $e[m,n]$ is the quantization error. It is computed as:

$$e[m,\ n] = g[m,\ n] - u[m,\ n], \tag{3}$$

and the weights $w[k,l;a]$ satisfy $\sum_{k,l} w[k,l;a] = 1$ to preserve the average local tone. To eliminate the checkboard patterns in the midtone, the threshold matrix $t[m,n;a]$ of TDFED is defined based on a halftone pattern $p[m,n;0.5]$ generated by DBS with period 128×128 for absorptance 0.5 as:

$$t[m,\ n;a] = \begin{cases} t_u(a), & \text{if}\, p[m,\ n;\ 0.5] = 0, \\ t_l(a), & \text{otherwise.} \end{cases} \tag{4}$$

where $t_u(a)$ and $t_l(a)$ are tone dependent parameters that serve as upper and lower thresholds satisfying $t_l(a) \leq t_u(a)$. Substituting (5) into (1) yields:

$$g[m,\ n] = \begin{cases} 1, & \text{if}\, u[m,\ n] \geq t_u(a), \\ 0, & \text{if}\, u[m,\ n] \leq t_l(a), \\ p[m,\ n;0.5], & \text{otherwise.} \end{cases} \tag{5}$$

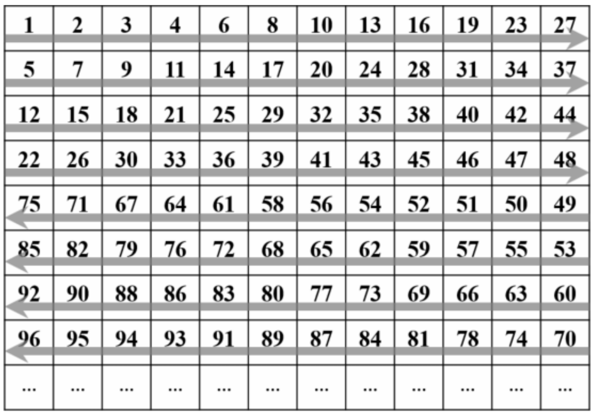
In order to reduce the computational complexity, Li and Allebach chose the optimal filters for tone levels higher than $127/256$ according to $t_u(a) = t_u(1-a)$ and $w[k,l;a] = w[k,l;1-a]$. The same strategy is used for all experiments in this paper.

2.2 4-Row Serpentine Scan Path

It has been shown in [4] that implementing error diffusion in serpentine order effectively reduces the worm artifacts in the extreme gray levels. However, serpentine error diffusion is intrinsically a serial process, we must finish an entire scan line before moving on to the next one. This limits the parallelism and locality required by hardware implementations. Thus, it is necessary to mix the serpentine and raster scan together to solve the problem.

In 4-row serpentine TDFED, every 4 rows of an input image are grouped as a swath. As depicted by the arrows in Fig. 2, four scan lines in the same swath are processed in one direction and the consecutive four lines are processed in the opposite direction. By doing so, the serpentine property is preserved between adjacent swaths.

To be more specific, inside each swath, error diffusion starts from the first pixel of the first row, and then it advances rightward along the first row in raster scan order until the next row is activated. The activation condition is that the binarization of d pixels in the preceding row has been finished, where d is defined as the delay between the processing of sequential lines in the 4-row swath. It can be adapted according to the need of the actual hardware architecture. Error diffusion travels back and forth across the activated scan lines. After the last pixel of the fourth row has been visited, the scan path then traverses leftward to process the next swath in the same manner described above. Figure 2 is an instance of the 4-row serpentine scan pattern in which d equals 3 pixels. Each entry of the matrix represents a pixel of the continuous-tone input image, and the number indicates the order in which the pixel at that location is processed.



1	2	3	4	6	8	10	13	16	19	23	27
5	7	9	11	14	17	20	24	28	31	34	37
12	15	18	21	25	29	32	35	38	40	42	44
22	26	30	33	36	39	41	43	45	46	47	48
75	71	67	64	61	58	56	54	52	51	50	49
85	82	79	76	72	68	65	62	59	57	55	53
92	90	88	86	83	80	77	73	69	66	63	60
96	95	94	93	91	89	87	84	81	78	74	70
...	...	...	...	...	...	...	...	...	...	...	...

Fig. 2: Scan path of 4-row serpentine scan with 3 pixel delay

2.3 Error Weight Location Matrix

To reduce worm-like patterns with a conventional raster scan error diffusion, randomized weights [4] and a low frequency modulated threshold matrix [6] have been proposed. Unfortunately, these approaches introduce noise to the halftone image. Jarvis et al and Stucki [20] proposed a larger set of error weights with 24 terms, which requires heavy computation. Shiau and Fan [21] moved the $1/16$ term in Floyd Steinberg weights from location $(1, 1)$ to $(-2, 1)$. Li and Allebach [18] developed a set of wider matrices that diffuse the errors further back. Our approach is based on [18].

As it can be seen from Fig. 4(a), the 4-row serpentine scan pattern does not produce long diagonal structures but generates lots of short diagonal worms in the highlights and shadows. Therefore, we need to spread the quantization error over a wider region in the problematic gray levels to disperse the worms. In 4-row serpentine TDFED, diffusing the errors to a further location requires an increase in the delay. We explored delays of 2, 3, and 6 pixels, which allow an increasingly larger spatial spread in the tone-dependent weight location matrix. To ensure minimum computations, we chose to use 4 non-zero weights as originally described in Floyd and Steinberg error diffusion. The matrix set designed for 3-pixel delay is presented in Table 1. The weights allocation and input partition are determined empirically. The values of the weights and thresholds are obtained using a search-based method, which we will discuss in the following section.

			*	w_1
w_4		w_3	w_2	

$$\frac{1}{255} \leq a \leq \frac{63}{255}, \frac{192}{255} \leq a \leq \frac{254}{255}$$

	*	w_1
w_4	w_3	w_2

$$\frac{64}{255} \leq a \leq \frac{191}{255}$$

	*	
	w_2	

$$a = 0, a = 1$$

Table 1: Expanded error weight location matrices designed for 4-row serpentine TDFED with 3 pixel delay. The asterisk sign denotes the current pixel.

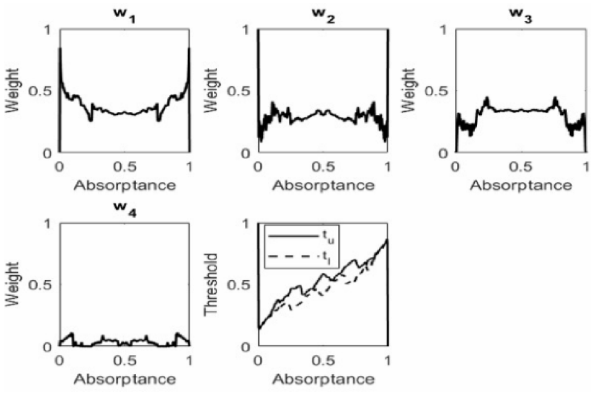


Fig. 3: Optimal tone dependent weights and thresholds for 4-row serpentine scan with 3 pixel delay

2.4 Training System for Tdfed Parameters

Although the expanded weight location matrix reduces worm artifacts, if the weights and thresholds are not properly designed, there will be correlated patterns and non-homogeneous textures in the output image. Thus, they must be optimized to achieve the best visual quality. We propose to use four non-zero error weights and two thresholds. There are only 4 degrees of freedom due to the constraints $\sum_{k,l} w[k,l;a] = 1$ and $t_u(a) + t_l(a) = 1$. We choose to optimize $w[0,1;a], w[1,0;a], w[1,1;a]$, and $t_u(a)$. Generally speaking, the TDFED training system searches for the optimal parameters by minimizing a cost function ε.

Li and Allebach [18] used two different cost functions depending on the tone level. For the extreme gray levels, the perceived mean squared error between the constant valued continuous-tone patch and a TDFED halftone is minimized based on Nasanen's HVS model [22]. For the midtones, the total squared error between the power spectra of the halftone patch generated by TDFED and by DBS is minimized, which is given by:

$$\varepsilon_{Li} = \sum_u \sum_v (\bar{G}_{DBS}[u,\ v;a] - \bar{G}_{TDFED}[u,\ v;a])^2, \tag{6}$$

where $\bar{G}_{TDFED}[u,v;a]$ and $\bar{G}_{DBS}[u,v;a]$ represent the average magnitude of the 2D Fourier amplitude spectra obtained from TDFED and DBS halftone patches, respectively.

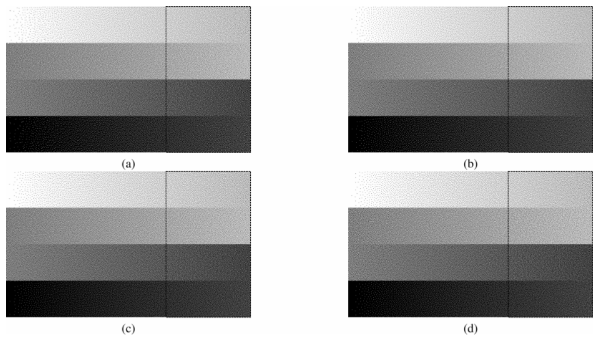


Fig. 4: Halftones of a folded ramp image generated by: (a) 4-Row serpentine TDFED with 1 pixel delay, (b) 4-row serpentine TDFED with 3 pixel delay, (c) 4-row serpentine TDFED with 6 pixel delay, (d) the original 1-row serpentine tdfed. It is recommended to zoom in on the figure so that individual pixels are clearly displayed and to view from a sufficient distance at which these individual pixels are not visually resolved.

Chang and Allebach [16] presented a cost function with a normalization as:

$$\varepsilon_{Chang} = \frac{\sum_u \sum_v (\bar{G}_{DBS}[u,v;a] - \bar{G}_{TDFED}[u,v;a])^2}{\bar{G}_{DBS}[u,v;a]^2}. \tag{7}$$

Han and Allebach [19] further modified the cost function by adding the power spectra of TDFED to the denominator:

$$\varepsilon_{Han} = \frac{\sum_u \sum_v (\bar{G}_{DBS}[u,v;a] - \bar{G}_{TDFED}[u,v;a])^2}{(\bar{G}_{DBS}[u,v;a] + \bar{G}_{TDFED}[u,v;a])^2}. \tag{8}$$

We concluded from our experiments that Han and Alle-bach's cost function yields the most reliable results for all levels. Thus, their cost function is adopted as the error metric.

As for the search strategy, we compared the performance of pattern search [23] with the downhill search [18], and established that downhill search is better suited to our application. The optimized weights and thresholds of 4-row serpentine TDFED with 3 pixel delay are shown in Fig. 3.

3 Experimental Results

Figures 4(a)-(c) show the result of 4-row serpentine TDFED with 1 pixel, 3 pixel, and 6 pixel delay, respectively. Figure 4(d) is the original 1-row serpentine TDFED result. It can be observed that in the highlights and shadows, both 3-pixel and 6-pixel delay 4-row serpentine TDFED significantly reduce the short diagonal structures seen in (a) and are comparable to 1 row serpentine TDFED. Moreover, Fig. 4(d) contains some vertical veining patterns in the region marked by the dotted square box. However, all three 4-row halftones are very smooth and homogeneous in that area.

4 Conclusions

With a modest delay value, 4-row serpentine TDFED can achieve essentially the same or better image quality than that provided by the original 1-row serpentine TDFED, except perhaps in the extreme gray levels. Besides, it will also boost efficiency and reduce memory cost in some hardware implementations.

译文 1

作者：Yafei Mao; Lluis Abello; Utpal Sarkar; Robert Ulichney; Jan Allebach

出处：IEEE International Conference on Image Processing. 2018:3973-3977.

四行蛇形色调自适应快速误差扩散算法

摘要：误差扩散是桌面打印机（尤其是喷墨打印机）中广泛应用的技术。然而，传统误差扩散采用光栅扫描顺序计算，会在高光和阴影区域产生“蠕虫伪影”；同时，其串行执行的特性限制了硬件实现的效率与灵活性。为解决这两个问题，本文提出一种基于蛇形扫描的新型误差扩散算法，该算法采用色调自适应的误差权重与阈值，并设计了扩展误差权重位置矩阵以提升极端色调下的半色调质量。与原始色调自适应快速误差扩散算法相比，本文算法能生成更优的半色调图像，尤其在四分之一色调区域表现突出。

关键词：半色调、改进型色调自适应快速误差扩散、并行实现

1　引言

数字半色调技术通过调整点的大小或间距，利用有限的色调层级模拟连续色调的输出效果，广泛应用于打印机、显示器等仅支持有限色调的设备中。

根据计算复杂度，半色调算法可分为三类：加网算法、误差扩散算法和搜索型算法。误差扩散的细节呈现效果优于加网算法，同时成本低于搜索型算法，因此成为喷墨打印机等能稳定渲染孤立点的标记引擎的主流算法。本文将聚焦于误差扩散技术。

弗洛伊德-斯坦伯格（Floyd and Steinberg）提出的传统误差扩散是一种邻域操作：按光栅顺序遍历输入图像，量化扫描线中的每个像素，并将量化误差传递给后续未二值化的邻域像素。尽管细节呈现效果出色，但误差扩散有时会产生蠕虫状图案和可见结构。为解决这些问题，已有研究提出了多种改进方案，包括采用替代扫描路径 [2]-[4]、阈值调制 [5]-[9]、可变权重 [4]、[10]-[12] 及色调自适应参数 [13]、[14] 等。文献 [15] 值得特别关注：它总结了基于权重训练以匹配蓝噪声频谱特性的近期方法，并提出了改进方案，同时将阈值训练纳入其中以消除边缘锐化。

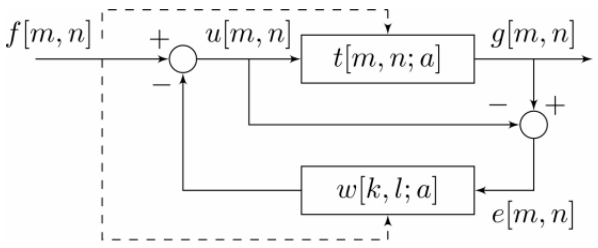


图 1: 色调自适应快速误差扩散系统。

除了生成视觉上不愉悦的纹理外，误差扩散的另一缺点是缺乏局部性。这意味着无论采用传统光栅扫描还是蛇形扫描路径 [4]，必须等前一条扫描线的所有像素完成量化后，才能处理当前像素。因此，硬件需存储空间上较远的像素状态信息，效率较低。而皮亚诺扫描 [2]（并行扫描路径的先驱）虽能并行处理，但误差扩散的输出质量无法令人满意。实际上，打印市场的客户同时关注打印质量与实现效率，因此在不损失质量的前提下实现硬件的局部二值输出决策具有重要意义。为此，本文设计了一种新型四行蛇形扫描路径，其创新点在于结合传统光栅与蛇形扫描模式，大幅提升了硬件效率。其他将蛇形光栅与新型误差扩散架构结合的相关工作包括 [16]、[17]。

本文将四行蛇形扫描路径与色调自适应快速误差扩散（TDFED）[18] 算法结合：为减少蠕虫状纹理，设计了色调自适应的四权重位置矩阵，将误差沿下一行进一步向后扩散；为进一步优化半色调输出，基于文献 [19] 提出的视觉代价函数，通过离线训练过程优化每个灰度级的权重与阈值。但本文的核心是开发适用于高效硬件实现的新型误差扩散架构，而非权重与阈值的优化方法及代价函数（这是文献 [15] 的核心内容）。实际上，本文提出的概念也可与文献 [15] 中的权重与阈值优化方法结合使用。

本文后续结构如下：第 2 节介绍四行蛇形色调自适应快速误差扩散算法；第 3 节讨论实验结果；第 4 节总结全文。

2　四行蛇形色调自适应快速误差扩散算法

本节首先在 2.1 小节概述色调自适应快速误差扩散算法，随后介绍扫描路径的细节，2.3 小节阐述扩展误差位置矩阵，最后在 2.4 小节讨论训练过程。

2.1　色调自适应快速误差扩散概述

由于本文提出的误差扩散算法面向单色打印设备，像素值以吸收率表示（$0 \leq a \leq 1$），其中 0 对应白色，1 对应黑色。

图 1 展示了 TDFED 系统的框图：$f[m,n] = a$ 是连续色调图像的像素吸收率，$u[m,n]$ 是更新后的像素值，$g[m,n]$ 是二值输出。与弗洛伊德-斯坦伯格误差扩散不同，

TDFED 的阈值 $t[m,n;a]$ 和误差权重 $w[k,l;a]$ 依赖于输入吸收率，其中 k,l 是相对位置索引，用于表示 $f[m,n]$ 的邻域像素位置。弗洛伊德-斯坦伯格定义的邻域像素为当前像素的右侧、右下、下方及左下（假设图像按光栅顺序从左到右扫描）；由于 TDFED 采用蛇形光栅顺序遍历图像，当从右到左扫描时，权重位置会采用镜像形式。

系统的二值输出通过对更新后的像素值进行阈值化得到：

$$g[m,\ n] = \begin{cases} 1, & \text{若 } u[m,\ n] \geq t[m,\ n;a], \\ 0, & \text{否则}. \end{cases} \tag{1}$$

更新后的连续色调像素值 $u[m,n]$ 计算如下：

$$u[m+k,\ n+l] \leftarrow u[m+k,\ n+l] - w[k,\ l;a] \cdot e[m,\ n], \tag{2}$$

其中 $e[m,n]$ 是量化误差，计算方式为：

$$e[m,\ n] = g[m,\ n] - u[m,\ n], \tag{3}$$

且权重 $w[k,l;a]$ 满足 $\sum_{k,l} w[k,l;a] = 1$，以保持局部平均色调。为消除中间调的棋盘格图案，TDFED 的阈值矩阵 $t[m,n;a]$ 基于吸收率为 0.5 时由 DBS 生成的半色调图案 $p[m,n;0.5]$（周期为 128×128）定义：

$$t[m,\ n;a] = \begin{cases} t_u(a), & \text{若 } p[m,\ n;\ 0.5] = 0, \\ t_l(a), & \text{否则}. \end{cases} \tag{4}$$

其中 $t_u(a)$ 和 $t_l(a)$ 是色调自适应参数，分别作为上阈值和下阈值，满足 $t_l(a) \leq t_u(a)$。将式 (5) 代入式 (1) 可得：

$$g[m,\ n] = \begin{cases} 1, & \text{若 } u[m,\ n] \geq t_u(a), \\ 0, & \text{若 } u[m,\ n] \leq t_l(a), \\ p[m,\ n;0.5], & \text{否则}. \end{cases} \tag{5}$$

为降低计算复杂度，Li 和 Allebach 选择当色调层级高于 127/256 时，最优滤波器满足 $t_u(a) = t_u(1-a)$ 且 $w[k,l;a] = w[k,l;1-a]$。本文所有实验均采用此策略。

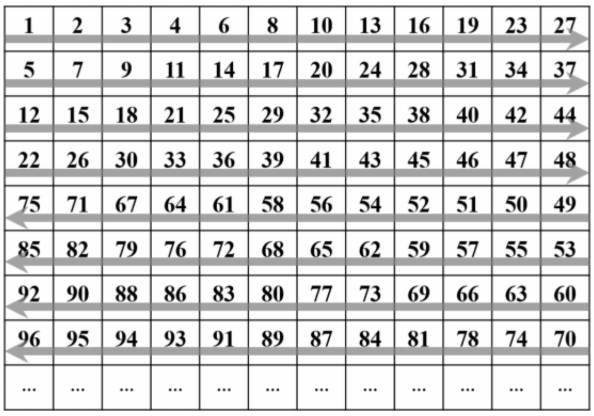
2.2 四行蛇形扫描路径

文献 [4] 表明，采用蛇形顺序实现误差扩散可有效减少极端灰度级的蠕虫伪影，但蛇形误差扩散本质上是串行过程：必须完成整条扫描线的处理后才能进入下一条，

这限制了硬件实现所需的并行性与局部性。因此，需将蛇形与光栅扫描结合以解决该问题。

在四行蛇形 TDFED 中，输入图像的每 4 行被分为一个“条带（swath）”。如图 2 中的箭头所示，同一条带内的 4 条扫描线按同一方向处理，相邻条带则按相反方向处理，从而在条带间保留蛇形特性。

具体而言，在每条带内，误差扩散从第一行的第一个像素开始，按光栅顺序沿第一行向右推进，直到下一行被激活。激活条件是前一行中 d 个像素完成二值化，其中 d 定义为四行条带中连续行处理的延迟，可根据实际硬件架构的需求调整。误差扩散在激活的扫描线间来回遍历；当第四行的最后一个像素处理完成后，扫描路径向左移动，按上述方式处理下一个条带。图 2 是 $d = 3$ 像素时四行蛇形扫描模式的实例：矩阵中的每个元素代表连续色调输入图像的一个像素，数字表示该位置像素的处理顺序。



1	2	3	4	6	8	10	13	16	19	23	27
5	7	9	11	14	17	20	24	28	31	34	37
12	15	18	21	25	29	32	35	38	40	42	44
22	26	30	33	36	39	41	43	45	46	47	48
75	71	67	64	61	58	56	54	52	51	50	49
85	82	79	76	72	68	65	62	59	57	55	53
92	90	88	86	83	80	77	73	69	66	63	60
96	95	94	93	91	89	87	84	81	78	74	70
...	...	...	...	...	...	...	...	...	...	...	...

图 2: 具有 3 像素延迟的四行蛇形扫描路径

2.3 误差权重位置矩阵

为减少传统光栅扫描误差扩散的蠕虫状图案，已有研究提出了随机权重 [4] 和低频调制阈值矩阵 [6]，但这些方法会给半色调图像引入噪声。Jarvis 等人和 Stucki[20] 提出了包含 24 项的更大误差权重集，但计算量较大；Shiau 和 Fan[21] 将弗洛伊德-斯坦伯格权重中的 1/16 项从位置 (1, 1) 移至 (-2, 1)；Li 和 Allebach[18] 开发了一组更宽的矩阵，将误差进一步向后扩散。本文方法基于文献 [18]。

从图 4(a) 可以看出，四行蛇形扫描模式不会产生长对角线结构，但会在高光和阴影区域生成大量短对角线蠕虫，因此需要在这些问题灰度级中将量化误差扩散到更宽的区域以分散蠕虫。在四行蛇形 TDFED 中，将误差扩散到更远位置需要增加延迟。本文测试了 2、3、6 像素的延迟，这些延迟允许色调自适应权重位置矩阵的空间

扩散范围逐渐增大。为确保计算量最小，本文选择弗洛伊德-斯坦伯格误差扩散中最初描述的 4 个非零权重。表 1 展示了为 3 像素延迟设计的矩阵集，权重分配和输入分区由经验确定，权重与阈值的值通过基于搜索的方法获得（下一小节将讨论）。

			*	w_1
w_4		w_3	w_2	

$$\frac{1}{255} \leq a \leq \frac{63}{255}, \frac{192}{255} \leq a \leq \frac{254}{255}$$

	*	w_1
w_4	w_3	w_2

$$\frac{64}{255} \leq a \leq \frac{191}{255}$$

	*	
	w_2	

$$a = 0, a = 1$$

表 1：为具有 3 像素延迟的四行蛇形 TDFED 设计的扩展误差权重位置矩阵。星号表示当前像素。

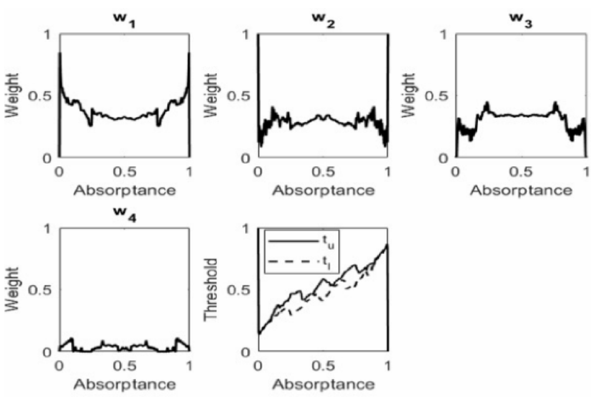


图 3: 具有 3 像素延迟的四行蛇形扫描的最优色调自适应权重与阈值

2.4 TDFED 参数的训练系统

尽管扩展权重位置矩阵可减少蠕虫伪影，但如果权重与阈值设计不当，输出图像会出现相关图案和非均匀纹理，因此必须对其进行优化以获得最佳视觉质量。本文采用 4 个非零误差权重和 2 个阈值，由于约束条件 $\sum_{k,l} w[k,l;a] = 1$ 和 $t_u(a) + t_l(a) = 1$，仅存在 4 个自由度，因此选择优化 $w[0,1;a]$、$w[1,0;a]$、$w[1,1;a]$ 和 $t_u(a)$。一般而言，TDFED 训练系统通过最小化代价函数 ε 来搜索最优参数。

Li 和 Allebach[18] 根据色调层级使用两种不同的代价函数：对于极端灰度级，基于 Nasanen 的人类视觉系统（HVS）模型 [22]，最小化恒定值连续色调块与 TDFED

半色调块之间的感知均方误差；对于中间调，最小化 TDFED 生成的半色调块与 DBS 生成的半色调块的功率谱之间的总平方误差，即：

$$\varepsilon_{Li} = \sum_u \sum_v (\bar{G}_{DBS}[u,\ v; a] - \bar{G}_{TDFED}[u,\ v; a])^2, \tag{6}$$

其中 $\bar{G}_{TDFED}[u, v; a]$ 和 $\bar{G}_{DBS}[u, v; a]$ 分别表示从 TDFED 和 DBS 半色调块获得的二维傅里叶振幅谱的平均值。

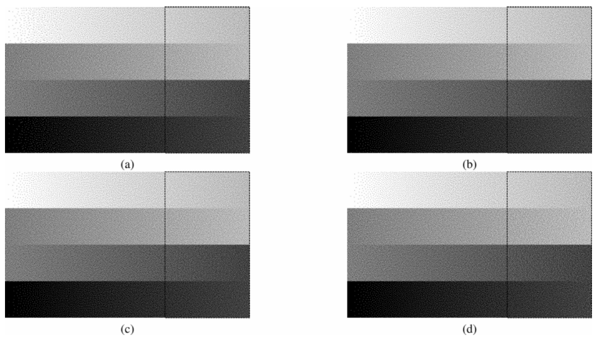


图 4: 折叠斜坡图像的半色调结果：(a) 具有 1 像素延迟的四行蛇形 TDFED；(b) 具有 3 像素延迟的四行蛇形 TDFED；(c) 具有 6 像素延迟的四行蛇形 TDFED；(d) 原始一行蛇形 TDFED。建议放大图像以清晰显示单个像素，并从足够远的距离观察（此时单个像素无法被视觉分辨）。

Chang 和 Allebach[16] 提出了带归一化的代价函数：

$$\varepsilon_{Chang} = \frac{\sum_u \sum_v (\bar{G}_{DBS}[u, v; a] - \bar{G}_{TDFED}[u, v; a])^2}{\bar{G}_{DBS}[u, v; a]^2}. \tag{7}$$

Han 和 Allebach[19] 进一步修改了代价函数，在分母中加入 TDFED 的功率谱：

$$\varepsilon_{Han} = \frac{\sum_u \sum_v (\bar{G}_{DBS}[u, v; a] - \bar{G}_{TDFED}[u, v; a])^2}{(\bar{G}_{DBS}[u, v; a] + \bar{G}_{TDFED}[u, v; a])^2}. \tag{8}$$

实验表明，Han 和 Allebach 的代价函数在所有层级均能产生最可靠的结果，因此本文将其作为误差度量。

在搜索策略方面，本文比较了模式搜索 [23] 与 downhill 搜索 [18] 的性能，发现 downhill 搜索更适合本文应用。图 3 展示了具有 3 像素延迟的四行蛇形 TDFED 的优化权重与阈值。

3 实验结果

图 4(a)-(c) 分别展示了具有 1 像素、3 像素和 6 像素延迟的四行蛇形 TDFED 的结果，图 4(d) 是原始一行蛇形 TDFED 的结果。可以观察到，在高光和阴影区域，3 像素和 6 像素延迟的四行蛇形 TDFED 显著减少了 (a) 中出现的短对角线结构，与一行蛇形 TDFED 的效果相当；此外，图 4(d) 中虚线方框标记的区域存在一些垂直纹理，而所有三种四行半色调图像在该区域均非常平滑均匀。

4　结论

通过适度的延迟值，四行蛇形 TDFED 的图像质量与原始一行蛇形 TDFED 基本相当或更优（极端灰度级可能除外）；同时，它还能提升部分硬件实现的效率并降低内存成本。

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.

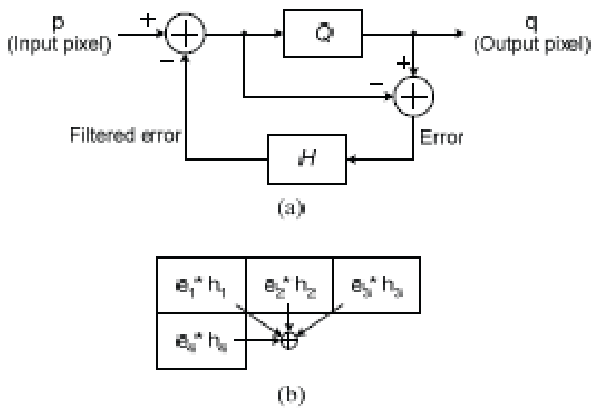


Fig. 1: (a) Block diagram of the error diffusion algorithm. (b) Floyd-Steinberg filter kernel, H

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.

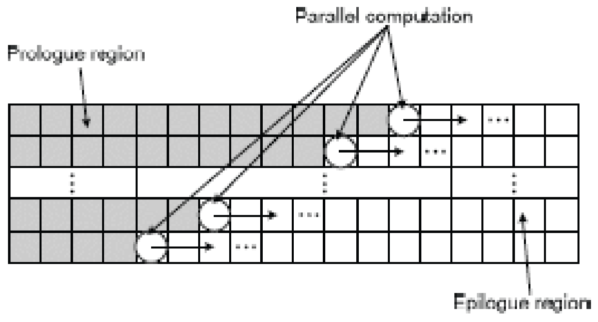


Fig. 2: A strip of an image assigned to a thread-block and independent pixels in different scan-lines of the strip

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.

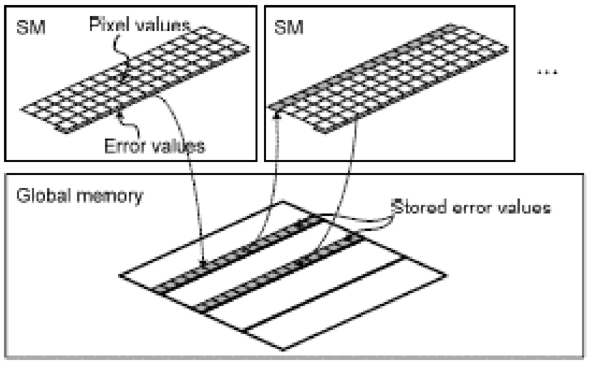


Fig. 3: Exchange of the error values between thread-blocks

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).

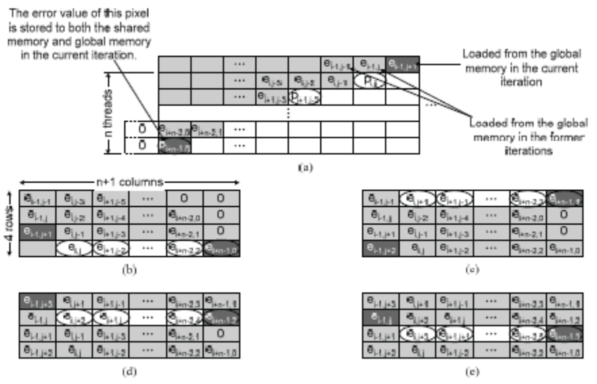

The error value of this pixel is stored to both the shared memory and global memory in the current iteration.
Loaded from the global memory in the current iteration
Loaded from the global memory in the former iterations
n threads
n+1 columns
4 rows
(a)
(b)
(c)
(d)
(e)

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.

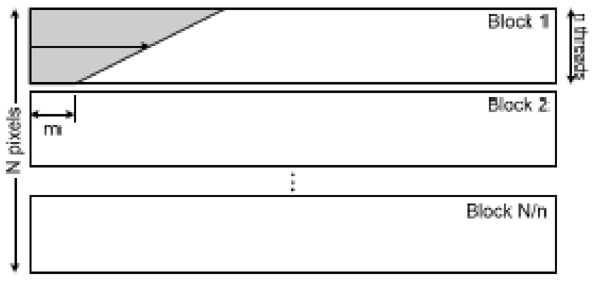


Fig. 5: Synchronization between inter-thread-block

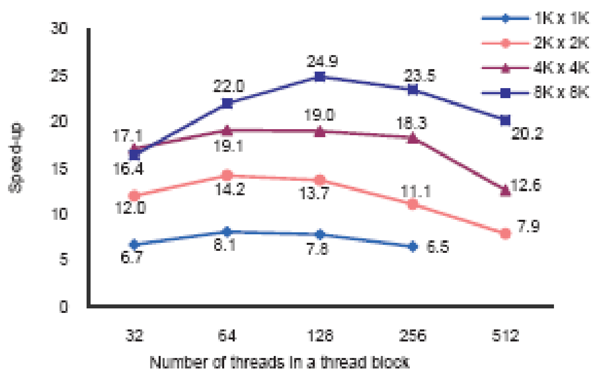


Fig. 6: Performance according to the thread-block size and image size

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.

译文 2

作者：Yafei Mao; Lluis Abello; Utpal Sarkar; Robert Ulichney; Jan Allebach

出处：IEEE International Conference on Image Processing. 2018:3973-3977.

基于通用图形处理器的误差扩散半色调算法并行实现

摘要：通用图形处理器（GPGPU）集成了大量执行单元，因此非常适用于高速图像处理。然而，误差扩散半色调算法难以利用大规模并行处理架构的优势——该算法不仅依赖邻域像素的计算结果，还需反馈输出误差。本研究通过分析依赖图识别出无依赖关系的可并行处理像素，并结合 GPGPU 的特性，提出一种同步开销更低的并行处理方法。

关键词：半色调、误差扩散、通用图形处理器、并行图像处理

1　引言

以 NVIDIA GTX 285 为代表的通用图形处理器（GPGPU）可通过数千个线程加速各类图像处理核函数的执行。但误差扩散算法 [1]（一种将多灰度级图像转换为二值图像的数字半色调技术）因包含反馈机制，难以实现并行化。图 1 展示了经典误差扩散过程的框图。已有研究针对误差扩散算法的并行实现展开探索：包括基于单指令多数据（SIMD）架构的设计（对比了三种并行处理方案 [2]），以及基于 MasPar 架构的实现 [3]。然而，GPGPU 需要数百以上的并行度，而 SIMD、超长指令字（VLIW）或超标量等传统并行架构的并行度通常不超过 16[4]。此外，误差扩散算法的像素处理顺序严格，其并行实现还需同步操作；GPGPU 中不同流式多处理器（SM）间的同步操作不仅困难且耗时，因此需尽可能减少同步次数。

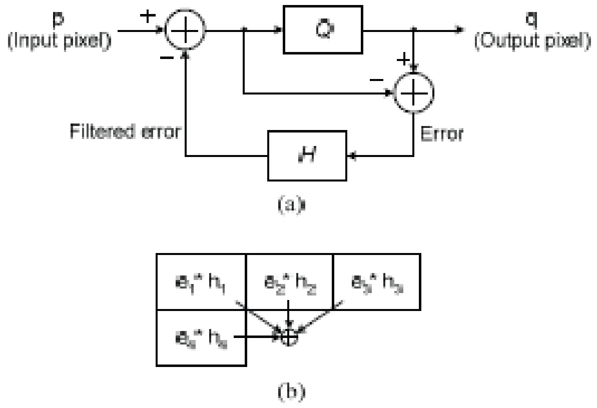


图 1:（a）误差扩散算法框图；（b）弗洛伊德-斯坦伯格滤波核 H

本研究采用 NVIDIA GTX 285 硬件及统一计算设备架构（CUDA）[5]：该硬件包含 30 个流式多处理器（SM），每个 SM 配备 8 个处理单元。为充分利用流水线处理单元，需为每个 SM 分配至少 32 个线程（称为一个线程束，warp）；同时需为每个 SM 分配多个线程束，通过任务切换掩盖全局内存访问延迟。在同一 SM 上执行的线程构成线程块（thread-block），一个线程块可包含多个线程束（每个线程束 32 个线程），因此线程块的线程数通常为 32 的整数倍。GPU 支持两级同步机制：同一线程块内的线程可借助专用硬件实现快速本地同步，而线程块间的同步需通过全局内存（DRAM）完成，耗时显著。每个 SM 包含四种片上内存：寄存器、共享内存、纹理缓存和常量缓存，其中共享内存可供同一线程块内的线程访问，且访问速度远高于全局内存。程序通过运行时调用 CUDA 接口管理设备内存（DRAM）空间，包括全局内存、常量内存和纹理内存。

本研究基于 GPGPU 的数千个线程实现了图 1 所示的误差扩散算法。尽管已有部分更适配并行处理硬件的误差扩散算法 [6]-[8]，但本研究仍选用图 1 中的经典算法——因其在数字印刷行业应用最为广泛。为消除线程间的依赖关系，采用斜扫描线（SSL）技术 [2]，该技术可利用相邻扫描线中斜向像素的独立性；同时，仅在线程块操作开始时执行耗时的线程块间同步，以降低同步开销，并解决了斜扫描线技术存在的负载不均衡问题。

本文结构如下：第 2 节阐述基于 GPU 的误差扩散算法并行化与实现技术；第 3 节展示实现结果；第 4 节给出结论。

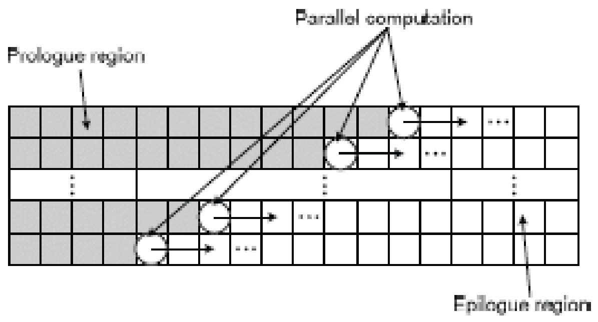


图 2: 分配给线程块的图像条带及条带内不同扫描线中的独立像素

2 基于 GPGPU 的并行实现

2.1 并行算法

GPU 可支持高度并发执行，因此不宜在单个像素处理层面挖掘并行性（误差扩散滤波核 H 仅包含少量可并行执行的算术运算），而需同时处理多个像素。同扫描线多像素处理的推测量化（SQ）方法 [2] 可适配 GPGPU 架构，但该方法虽能实现完美负载均衡，却存在补偿步骤耗时随并行度提升而增加、线程块间同步需求高等问题。

本研究采用改进版斜扫描线方法 [2]：在误差扩散算法中，看似需完成第 i 行所有像素处理后才能开始第 i+1 行，但深入分析可知，当处理第 i 行第 j 列像素时，可并行处理第 i+1 行第 j-2 列像素——这是因为第 i+1 行第 j-2 列像素仅依赖第 i 行 j-3、j-2、j-1 列像素的误差值。因此，(i,j)、(i+1,j-2)、(i+2,j-4)⋯位置的像素可并行处理（其中 (i,j) 表示第 i 行第 j 列像素）。当图像宽度为 W 时，该方法可支持多达 W/2 个并行线程；考虑到多数印刷图像的宽度达数千像素，该方法提供的并行度已足够。但该方法存在两大缺陷：一是斜向处理导致的负载不均衡，二是同步开销过大。如图 2 所示的起始区域（prologue），并非所有线程都能被有效利用；同理，算法还存在收尾区域（epilogue），需让提前启动的线程等待其他线程完成。对于宽度为 W 的图像，若使用 W/2 个线程，负载不均衡导致的效率下限为 50

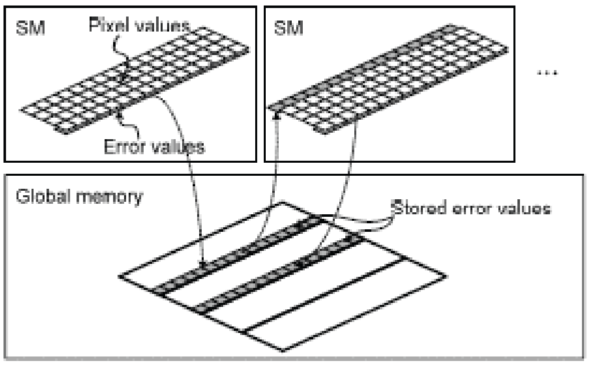


图 3: 线程块间的误差值交换

图 2 展示了分配给线程块的图像条带：灰色框代表已处理像素，带圈框代表当前正在处理的像素。线程块中的一个线程负责处理条带的一条扫描线，线程需与其他线程同步，按顺序处理扫描线中的像素。需注意，同一线程块内的线程同步是必需的——若斜向像素组中任一像素（首行像素除外）未与其他像素同步，将无法获取正确的误差值。与线程块间同步不同，线程块内同步的开销极低，仅需 4 个时钟周期 [5]。但该方法仍需线程块间同步：为使下一个条带尽快启动，需在当前条带计算出所需误差值后触发同步。线程块间同步通过全局内存访问实现消息传递，并通过极少执行同步（例如仅在条带首行的起始像素处同步）来最小化同步次数；同时，通过实验确定线程块间处理的时间裕量，确保下一条带的线程不会违反依赖关系。

负载不均衡、同步开销与全局内存访问次数之间存在权衡：增加条带高度（即每个线程块分配更多线程）会加剧负载不均衡，但可减少全局内存访问次数和线程块间同步开销。

2.2 实现细节

在 GPU 实现中，线程从全局内存加载自身负责的输入像素，利用存储在共享内存或全局内存中的其他误差值计算当前像素的误差值，随后计算输出像素值并写入全局内存。该过程中需尽可能减少全局内存访问次数——读取全局内存数据至少需要 400 600 个时钟周期 [5]。线程块间共享误差值时需将其存储在全局内存，但仅需存储线程块最后一行的误差值（供下一个线程块使用），而非所有误差值（图 3）。

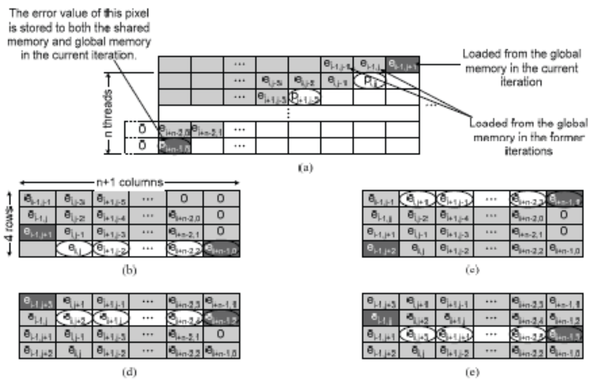

The error value of this pixel is stored to both the shared memory and global memory in the current iteration.
Loaded from the global memory in the current iteration
Loaded from the global memory in the former iterations
n threads
n+1 columns
4 rows
(a)
(b)
(c)
(d)
(e)

图 4: 线程块中的像素与计算得到的误差值，以及共享内存数组的内容

图 4 展示了误差扩散算法 GPU 实现中共享内存的使用方式：（a）为线程块正在处理的图像条带，浅灰色框为已处理且包含误差值的像素，带圈框 $p_{i,j}, p_{i+1,j-2}, \ldots, p_{i+n-1,0}$ 为当前步骤正在量化的像素（即构成一条并行处理的斜向线）。处理 $p_{i,j}$ 需使用误差值 $e_{i-1,j-1}, e_{i-1,j}, e_{i-1,j+1}$ 和 $e_{i,j-1}$——其中 $e_{i-1,j+1}$ 需从全局内存读取，其余误差值存储在共享内存中。（b）（e）为共享内存中存储的误差矩阵（大小为 4 行 ×(n+1) 列，n 为线程块的线程数）：-（b）中，本轮迭代生成的新误差值 $e_{i,j}, e_{i+1,j-2}, \ldots, e_{i+n-1,0}$ 存储在第四行；计算第二列第四行的 $e_{i,j}$ 时，需使用第一列的 $e_{i-1,j-2}, e_{i-1,j}, e_{i-1,j+1}$ 和正上方的 $e_{i,j-1}$。这种 L 型访问模式适合同一斜向线的所有像素，且因采用步长为 4 字节的线性寻址，无存储体冲突（bank conflict）。- 完成当前像素量化并计算误差值后，处理下一条斜向线 $p_{i,j+1}, p_{i+1,j-1}, \ldots, p_{i+n-1,1}$，新误差值存储在共享内存误差矩阵的第一行（c）——即采用循环缓冲区机制，最后一行的下一行是第一行。由于（b）中第一行的数值不再复用，可存储新误差值 $e_{i,j+1}, e_{i+1,j-1}, \ldots, e_{i+n-2,3}$ 和 $e_{i+n-1,1}$。- 后续迭代重复上述过程，持续更新共享内存内容（d）和（e）；当（e）中带圈框对应的误差值计算完成后，数组中误差值的相对位置与（b）一致，完成一个循环。

所有线程块完成计算后，即可得到最终的半色调图像。但如前所述，线程块依赖前一个线程块的计算结果，因此需同步。图 5 展示了整幅图像分配给 N/n 个线程块的情况（N 为图像高度，n 为线程块的线程数）：处理过程中，下一个线程块需等待前一个线程块处理到图像的特定像素位置 m；当前一个线程块到达该位置时，向全局内存写入特定值，下一个线程块从核函数启动时持续读取该值，确认值被修改后开始自身处理。该同步过程持续至最后一个线程块启动。当图像宽度较小或裕量 m 足够大时，每个线程块仅需在起始点执行一次同步操作，因此开销可忽略。

3　实现结果

为评估实现性能，对比了两种方案的执行时间：运行在 CPU 上的传统 C 程序，以及运行在 GPU 上的 CUDA 程序。实验平台为 Intel Core 2 Quad CPU（主频 2.83 GHz）和 NVIDIA GTX 285 GPU（240 个可用核心，主频 1.48 GHz），操作系统为 Linux Fedora 10，分别使用 GCC 编译 C 代码、NVCC 编译 CUDA 代码。

实验中有两个关键参数影响执行时间：1. 线程块大小：如前所述，影响负载均衡、全局内存访问次数和线程块间同步开销；2. 裕量 m：用于避免频繁同步，其取值依赖线程块大小和图像宽度（通过实验确定）。例如，图像尺寸为 1K×1K 像素、线程数为 32 时，m 取 40。m 越小，加速比越高，但过小可能导致当前线程块的处理速度超过前一个线程块；为稳妥起见，可检查从先前条带读取的误差值是否在预设范围内。

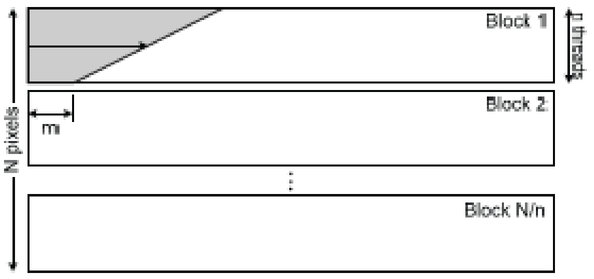


图 5: 线程块间的同步机制

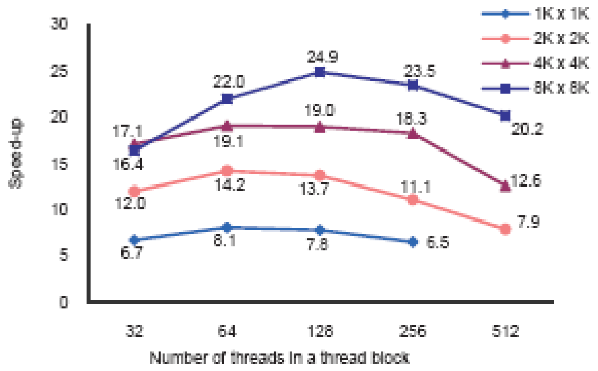


图 6: 不同线程块大小和图像尺寸下的性能表现

综合上述因素，测试了不同尺寸图像的加速比（图 6）：总体而言，线程块大小为 64 128 时可获得最大加速比——此时负载均衡、全局内存访问次数与线程块间开销的权衡达到最优。需注意流式多处理器（SM）的占用率：线程块过大时性能下降的主要原因是片上内存资源受限（线程数增加导致每个线程可用内存减少）。此外，图

像尺寸越大，加速比越高——这是因为大尺寸图像可激活更多线程，提升 GPU 利用率。

若计入主机与设备间的数据传输时间，加速比会降低（表 1）——数据传输时间占比显著。例如，8K×8K 像素图像的核函数执行加速比达 24.9，但计入数据传输时间后降至 9.8。不过，当多个应用以核函数组形式在 GPU 上执行时，该开销的影响可忽略。

表 1：不同图像尺寸（像素）下的核函数执行时间（秒）与数据传输时间（秒）对比

图像尺寸	**核函数执行时间（a）**	**数据传输时间（b）**	**CPU 时间（c）**	**加速比 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　结论

本研究基于 GPGPU 的数千个线程实现了误差扩散半色调算法：采用斜扫描线方法识别独立像素，通过减少线程块间同步次数、高效利用共享内存最小化开销。结果表明，与运行在 PC 上的传统 C 程序相比，该技术可实现约 24.9 倍的加速比。