Parallelization Techniques for Error Diffusion with GPU Implementations

Akihiko Kasagi*, Koji Nakano, and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan
* *Currently with Fujitsu Laboratories Ltd.*

Abstract*—Error diffusion is a classical but still popular method for generating a binary image that reproduces an original gray-scale image. In error diffusion, pixel values are rounded to binary in raster scan order and the rounding error is distributed to neighboring pixels that have not yet been processed. The main contribution of this paper is to show several parallel algorithms and implementation techniques for error diffusion. We first present error collection, which collects the quantization error from neighboring pixels that have already been processed. Error collection, which outputs the same binary image as error diffusion, performs fewer memory writing operations, and thus it is more efficient than error diffusion. We also present parallel implementations for error diffusion and error collection on the asynchronous CRCW-PRAM. From the theoretical analysis, we show that parallel error diffusion must use one of the three costly sidestep techniques: lower parallelism, atomic addition operations, or extra barrier synchronization steps, while parallel error collection does not need them. We have implemented parallel error diffusion and parallel error collection designed for the asynchronous CRCW-PRAM in the GPU. Experimental results show that parallel error collection runs the fastest on the GPU. Further, we have designed parallel algorithms for error diffusion and error collection optimized for CUDA-enabled GPUs using various implementation techniques. From the theoretical point of view, our parallel algorithms are global memory access optimal. Our parallel error collection algorithm for 256M pixels on GeForce GTX 780Ti runs only 46.75ms and achieves a speedup factor of 43.9 over the best sequential error collection algorithm running on Intel Core-i7 3770K CPU.

***Keywords*-digital halftoning, parallel algorithms, CUDA, GPGPU.**

I. Introduction

Halftoning is an important task to convert a continuous-tone image into a binary image with pure black and white pixels [1]. This task is necessary when printing a monochrome or color image by a printer with limited number of ink colors. Error diffusion [2] is a classical but still popular method for generating a binary image that reproduces an original gray-scale image. In error diffusion, pixels are rounded to binary in raster scan order and the rounding error is distributed to neighboring pixels that have not yet been processed. Ordered dithering [3] is also a popular halftoning method, which is used for inkjet and laser printers. It generates a binary image by applying a threshold map to an original gray-scale image. In general, error diffusion can generate binary images with better printing results than ordered dithering as we can see in Figure 1. Although both error diffusion and ordered dithering can be done in $O(n)$ time for a gray-scale image with n pixels, computation by error diffusion is much more complicated than that by ordered dithering. Hence, error diffusion is used only for high-quality printing in many inkjet printers, because it takes a lot of time to prepare a binary image for printing. On the other hand, since ordered dither can run much faster than error diffusion, it is mainly used for regular printing. Thus, it is very important to accelerate error diffusion for getting high-quality printing results in a short time.

The GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [4], [5]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [6], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [7], since they have thousands of processor cores and very high memory bandwidth.

When we develop programs running on GPUs, we can use CUDA programming model to support scalability. Usually, a CUDA program executed on the host computer invokes CUDA kernels one or more times. A CUDA kernel executes one or more CUDA blocks running on SMs (Streming Multiprocessors) of the GPU. CUDA blocks in a CUDA kernel are identical in the sense that they have the same number of threads executing the same program. CUDA blocks are dispatched to a SM in turn. Hence, to synchronize all threads in all CUDA blocks, we need to use separate CUDA kernel calls, because SMs in the GPU executes CUDA blocks in turn. Since the synchronization of all CUDA blocks are very costly, we should minimize the number of such synchronization operations.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [6]. The

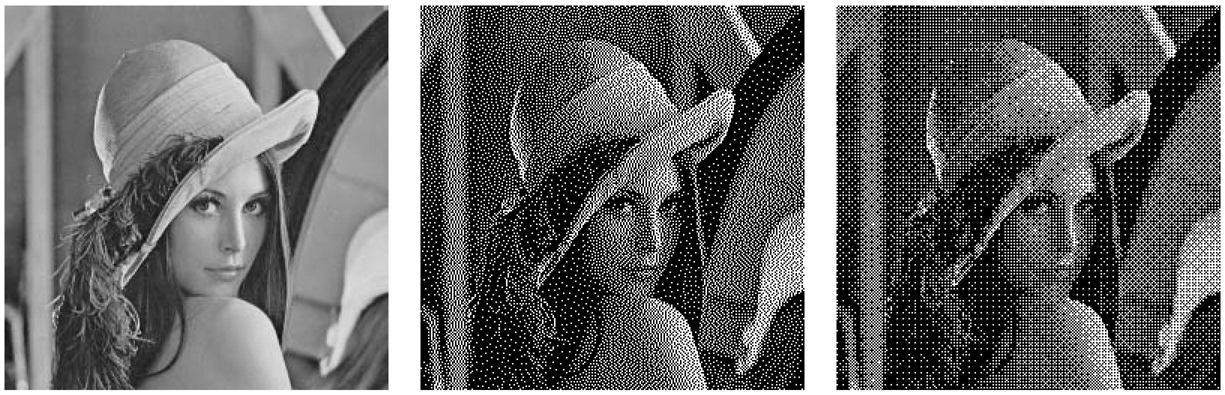

Figure 1. An original gray scale image “lena” and binary images generated by error diffusion and ordered dithering

shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key issue for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflicts* of the shared memory access and *coalescing* of the global memory access [8]–[10]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid bank conflicts of memory access. To maximize the bandwidth between the GPU and the DRAM chips, consecutive addresses of the global memory must be accessed at the same time. Thus, threads should perform coalesced access when they access the global memory. Further, it is important for accelerating computation to reduce the size of global memory access.

The main contribution of this paper is to present several parallelization techniques and algorithms for error diffusion and implement it on the GPU. We first present error collection, which collects rounding errors from neighboring pixels that have already been processed. Error collection outputs the same binary image as error diffusion. Figure 2 illustrates error diffusion and error collection. In error diffusion, the rounding error is distributed to four unprocessed neighboring pixels. On the other hand, error collection gathers the rounding errors from processed neighboring pixels. Error diffusion involves four writing operations to neighboring pixels, while error collection performs one writing operation to store the sum of four rounding errors. Hence, error collection performs fewer writing operations than error diffusion and runs faster especially when writing operation is costly.

We then go on to show parallel algorithms for error

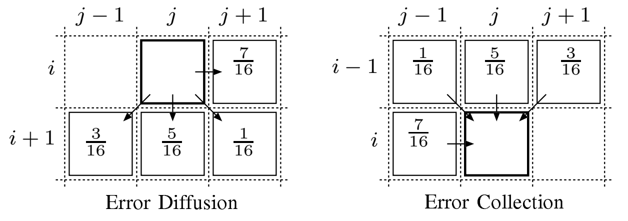

Figure 2. Error diffusion and error collection

diffusion and error collection. To see the essence of parallelism of error diffusion and error collection, we use the asynchronous version of the CRCW-PRAM (Concurrent Read Exclusive Write-Parallel Random Access Machine) model [11], in which every processor works asynchronously. We assume that processors in the asynchronous CRCW-PRAM can execute BARRIR_SYNC instruction for barrier synchronization. Also, atomic operations such as atomicAdd is supported. Since barrier synchronization and atomic operations impose certain overhead costs, these operations should be avoided when we design parallel algorithms. Our algorithm for error diffusion on the asynchronous CRCW-PRAM is based on a parallel error diffusion technique for a linear array [12]. Each processor is assigned to a row of an input image, and it executes error diffusion operation from left to right. The reader should refer to a snapshot of 2-delay parallel error diffusion (2-D PED) in Figure 3 to see how error diffusion is performed in each row. Adjacent error diffusion operations may diffuse error to the same pixel at the same time and the resulting binary image may be incorrect. We will show that, to resolve this problem, parallel error diffusion need to use one of the three costly sidestep techniques: lower parallelism, atomic addition operations, or extra barrier synchronization steps. We will show that 2-delay parallel error collection (2-D PEC) does not have such simultaneous writing operation as we can see in Figure 3. We

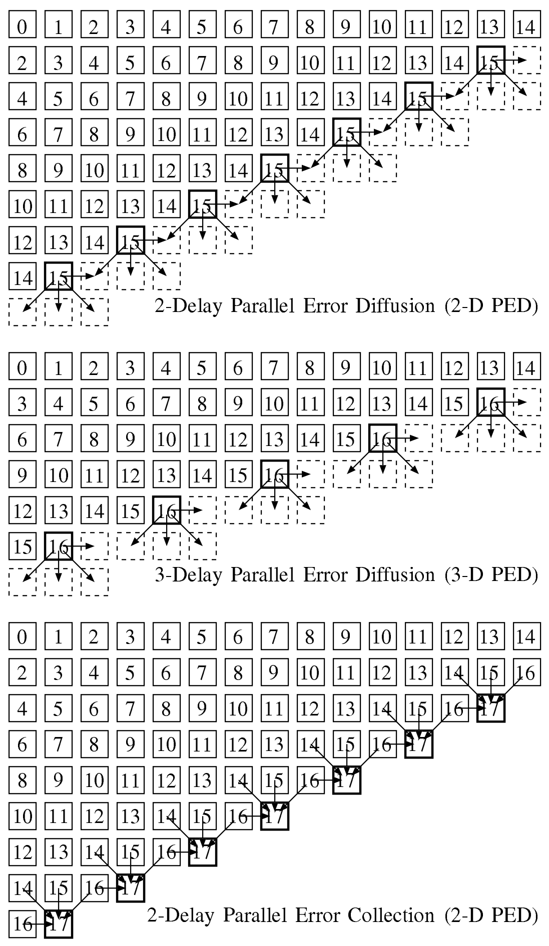
2-Delay Parallel Error Diffusion (2-D PED)
3-Delay Parallel Error Diffusion (3-D PED)
2-Delay Parallel Error Collection (2-D PED)

Figure 3. Parallel error diffusion and parallel error collection

have implemented parallel error diffusion and parallel error collection using CUDA and evaluated the performance on GeForce GTX 780Ti. The experimental results show that, 2-D PEC runs faster than 2-D PED with each of three sidestep techniques. Hence, 2-D PEC is more efficient than 2-D PED from the practical point of view.

Since 2-D PED and 2-D PEC are designed for the asynchronous CRCW-PRAM, there is a lot of room for improvement to implement these algorithms on CUDA-enabled GPUs. Our new idea is to partition the input gray-scale image into parallelogram blocks, and block-wise parallel error diffusion/error collection is executed for them at the same time. To guarantee that errors are diffused/collected correctly, parallelogram blocks are processed from the top-left corner to the bottom-right corner of the gray-scale image. One CUDA block is assigned to a parallelogram block and 2-D PED/2-PEC are executed on the shared memory of a streaming multiprocessor of a GPU in parallel. Our implementations for an n-pixel gray-scale image perform $n + O(\frac{n}{w})$ writing operations, and $n + O(\frac{n}{w})$ reading operations to the global memory, where w is the number of threads in a warp and the number of memory banks of the GPU. Since at least n writing operations and n reading operations are necessary, we can say that our implementations are global memory access optimal from the theoretical point of view. Further, all memory access operations to the global memory and the shared memory are coalesced and conflict-free, respectively. Implementation results show that our parallel error collection algorithm for 256M pixels on runs only 46.75ms. Since the best sequential algorithm runs 2052ms in Intel Core-i7 processor, achieves a speedup factor of 43.9.

GPUs have been used for accelerating various halftoning algorithms [13]–[15]. There are several implementation results for accelerating error diffusion using GPUs [16], [17]. It was shown in [16] that parallel error diffusion implemented in GeForce 8600M run 400ms for 16M pixels. On the other hand, our implementation runs only 16.63ms on GeForce GTX 780 Ti for an image of the same size. A variant of error diffusion called pinwheel error diffusion [18] have been implemented in a GPU [17]. The basic idea of pinwheel error diffusion to partition an input gray-scale image into square blocks and execute error diffusion operation for all blocks in parallel. Since all blocks of images can be processed independently, high parallelism can be obtained very easily. However, rounding errors are trapped within each block, the resulting binary images have uncomfortable periodical artifacts especially when the size of block is small. Their implementation achieves 475.5M pixels/s for 16M-pixel halftoning for 14×14 blocks using GeForce GTX 460. Our implementation follows the original error diffusion [2] that distributes errors to whole pixels. Although the original error diffusion is hard to parallelize, the performance of our implementation is better.

This paper is organized as follows. We first review error diffusion and present error collection in Section II. Also, we show register caching technique to accelerate error diffusion and error collection, and evaluate the number of memory access operations. Section III shows parallel algorithms for error diffusion and error collection on the asynchronous CRCW-PRAM and Section IV shows implementations of these parallel algorithms in a CUDA-enabled GPU. In Section V, we show parallel algorithms for error diffusion and error collection that are optimized for CUDA-enabled GPUs. We show experimental results for all sequential/parallel algorithms for error diffusion and error collection on a single Intel CPU and a CUDA-enabled GPU in Section VI. Section VII concludes our work.

II. Error Diffusion and Error Collection

This section first reviews *error diffusion* [2], and then shows our new method, *error collection*, which outputs exactly the same binary image as error diffusion. Since error collection performs fewer writing operations to image arrays than error diffusion, it is more efficient and runs faster.

Let a be a gray-scale image of size $\sqrt{n} \times \sqrt{n}$ such that each pixel $a[i][j]$ $(0 \leq i, j \leq \sqrt{n}-1)$ takes an intensity level (i.e. a real number) in the range $[0,1]$. For simplicity, we assume that an image is square, but all algorithms presented in this paper can be modified to run on non-square rectangular images easily. *Error Diffusion (ED)* outputs a binary image b of the same size such that each pixel $b[i][j]$ takes a binary value (i.e. 0 or 1). Input image a is scanned in raster scan order and error diffusion operation is performed one by one. Error diffusion operation rounds the value of $a[i][j]$ to 0 or 1 and the resulting binary value is stored in $b[i][j]$. Rounding error e $(= a[i][j] - b[i][j])$ is diffused to neighboring unprocessed four pixels as illustrated in Figure 2. The details are spelled out as follows:

[Error Diffusion (ED)]

```
for $i \leftarrow 0$ to $\sqrt{n}-1$ do
    for $j \leftarrow 0$ to $\sqrt{n}-1$ do
        if $a[i][j] \leq \frac{1}{2}$ then $r \leftarrow 0$ else $r \leftarrow 1$;
        $b[i][j] \leftarrow r$; $e \leftarrow a[i][j] - r$;
        $a[i][j+1] \leftarrow a[i][j+1] + \frac{7}{16} \cdot e$;
        $a[i+1][j+1] \leftarrow a[i+1][j+1] + \frac{1}{16} \cdot e$;
        $a[i+1][j] \leftarrow a[i+1][j] + \frac{5}{16} \cdot e$;
        $a[i+1][j-1] \leftarrow a[i+1][j-1] + \frac{3}{16} \cdot e$;
```

We assume that two temporary variables r and e are allocated as registers in a processor. Variable r is used to store the resulting binary value and variable e stores the rounding error to be diffused. For simplicity, we assume that the values of $a[i][j]$ such that $i = -1, \sqrt{n}$ or $j = -1, \sqrt{n}$ are zero to avoid special treatment for boundary pixels.

Similarly to error diffusion, *error collection* scans input image a in raster scan order, and for each pixel in a, rounding errors are collected from neighboring processed four pixels as illustrated in Figure 2. The details of error collection are spelled out as follows:

[Error Collection (EC)]

```
for $i \leftarrow 0$ to $\sqrt{n}-1$ do
    for $j \leftarrow 0$ to $\sqrt{n}-1$ do
        $s \leftarrow a[i][j] + \frac{7}{16} \cdot a[i][j-1] + \frac{1}{16} \cdot a[i-1][j-1]$
            $+ \frac{5}{16} \cdot a[i-1][j] + \frac{3}{16} \cdot a[i-1][j+1]$;
        if $s \leq \frac{1}{2}$ then $r \leftarrow 0$ else $r \leftarrow 1$;
        $a[i][j] \leftarrow s - r$; $b[i][j] \leftarrow r$;
```

Similarly, we assume that two temporary variables r and s are allocated as registers. Variables r and s are used to store the sum of rounding errors and the resulting binary value. The reader should have no difficulty to confirm that, for each pair of neighboring pixels, errors diffused/collected are the same, and thus resulting binary images b generated by error diffusion and error collection are identical.

Since the value of each binary pixel $b[i][j]$ can be determined by a constant number of instructions, both halftoning algorithms run $O(n)$ time for an input image with n pixels. However, error collection performs fewer memory access operations to gray-scale a and binary image b than error diffusion. To determine the value of a binary pixel $b[i][j]$, error diffusion performs five reading operations for $a[i][j], a[i][j+1], a[i+1][j+1], a[i+1][j], a[i+1][j-1]$ and five writing operations for $b[i][j], a[i][j+1], a[i+1][j+1], a[i+1][j], a[i+1][j-1]$. On the other hand, error collection performs five reading operations for $a[i][j], a[i][j+1], a[i+1][j+1], a[i+1][j], a[i+1][j-1]$, and two writing operations for $b[i][j]$ and $a[i][j]$. Thus, error collection performs fewer writing operations to image a and runs faster than error collection.

Table I
MEMORY ACCESS OPERATIONS OF ERROR DIFFUSION (ED) AND ERROR COLLECTION (EC) WITH AND WITHOUT REGISTER CACHING (RC)

	main memory		register caching	
	read	write	read	write
ED	$5n$	$5n$	-	-
EC	$5n$	$2n$	-	-
ED with RC	$2n$	$2n$	$9n$	$10n$
EC with RC	$2n$	$2n$	$9n$	$7n$

We can reduce the number of memory access operations to gray-scale image a by error diffusion and error collection using *register caching technique*. In register caching technique, we use five additional registers in a processor to store the current values of $a[i][j], a[i][j+1], a[i+1][j+1], a[i+1][j]$, and $a[i+1][j-1]$. Hence, to perform error diffusion operation for pixel $a[i][j]$, the current values of $a[i][j+1]$ and $a[i+1][j+1]$ are copied to registers. After error diffusion operation for pixel $a[i][j]$ is completed, and the resulting values of $a[i][j-1]$ in a register are copied to a in the main memory. For next error diffusion operation for $a[i][j+1]$, the values of registers are shifted by one from right to left. Hence, error diffusion with register caching technique performs two read operations and one write operation per pixel in the main memory for a One write operation to b is also performed. Similarly to error diffusion, we can apply register caching technique to error collection. We use five additional registers to cache $a[i-1][j+1], a[i-1][j], a[i-1][j-1], a[i][j]$, and $a[i][j-1]$ for error collection.

Table I summarizes the number of memory access operations for error diffusion and error collection with and without register caching. The number of memory access operations for register caching does not include that for temporal register variables such as r, e, and s. Note that the number of memory access to the register caching is evaluated based on straightforward register caching implementations. We can see that EC performs fewer memory access operations than ED.

III. Parallel Algorithms for Error Diffusion and Error Collection

This section shows parallel algorithms for error diffusion and error collection. To see the essence of parallelization of error diffusion and error collection, we use the asynchronous CRCW-PRAM as a target parallel machine. The PRAM (Parallel Random Access Machine) is a standard theoretical model for parallel computing, which has a shared memory can be accessed by all processors at the same time [19], [20]. In CRCW-PRAM (Concurrent Read Exclusive Write-PRAM), simultaneous reading from the same address is allowed. Simultaneous writing is also allowed, but one of them succeeds in writing and the others fails if two or more processors write to the same address at the same time. The asynchronous PRAM is a variant of the PRAM, in which processors work asynchronously [11]. For the purpose of barrier synchronization, processors can execute barrier synchronization instruction BARRIER_SYNC. If a processor executes BARRIER_SYNC, it is stalled until all the other processors execute it. Also, they can execute atomic writing operations such as atomicAdd(x, y) in which three instructions "load x", "add y", and "store x" are executed without any interruption to guarantee that "$x \leftarrow x + y$" can be completed consistently.

We first show a basic parallel algorithm called *k-delay parallel scan* (k-D PS) for a fixed parameter k (≥ 1), which will be used in parallel error diffusion and parallel error collection. In k-delay parallel scan, a processor is assigned to each i-th row of gray-scale image a and binary image b and performs some computation for pixels in this row from left to right. The details are spelled out as follows:

[k-delay parallel scan (k-D PS)]

```
for $t \leftarrow 0$ to $(k+1)(\sqrt{n}-1)$ do
    for $i \leftarrow 0$ to $\sqrt{n}-1$ do in parallel
        $j = t - k \cdot i$;
        if $0 \leq j \leq \sqrt{n}-1$ then
            COMP$(i, j)$;
            BARRIER_SYNC;
```

In k-D PS, COMP(i, j) denotes some computation for $a[i][j]$ and $b[i][j]$. It can be error diffusion operation or error collection operation for $a[i][j]$ and $b[i][j]$. Let us see how k-D PS works. First, when $t = 0$, a processor assigned to the first row performs COMP$(0, 0)$. After that, it performs COMP$(0, 1)$, COMP$(0, 2)$, ..., COMP$(0, k-1)$ one by one, when $t = 1, 2, \ldots, k-1$. When $t = k$, it performs COMP$(0, k)$, and a processor assigned to the second row performs COMP$(1, 0)$. The same procedure is repeated until COMP$(\sqrt{n}-1, \sqrt{n}-1)$ is performed when $t = (k+1)(\sqrt{n}-1)$. Hence, k-D PS performs BARRIER_SYNC $(k+1)(\sqrt{n}-1)+1$ times. Clearly, k-D PS performs COMP for one pixel in every k columns. Hence, pixels in at most $m = \lfloor \frac{\sqrt{n}-1}{k} \rfloor + 1$ consecutive rows are processed, and m processors are sufficient to execute k-D PS. More specifically, the $(i \bmod m)$-th processor $(0 \leq i \leq \sqrt{n}-1)$ works for computation of i-th row of a. Since m COMP(i, j)'s are executed by m processors in parallel, we can say that k-D PS has a parallelism factor of $m \approx \frac{\sqrt{n}}{k}$.

If COMP(i, j) in k-D PS performs error diffusion operation or error collection operation for $a[i][j]$, we call them k-delay parallel error diffusion (k-D PED) and k-delay parallel error collection (k-D PEC), respectively. The reader should refer to Figure 3 illustrating 2-delay parallel error diffusion (2-D PED), 3-delay parallel error diffusion (3-D PED), and 2-delay parallel error collection (2-D PEC). From the figure, we can see that 3-D PED always diffuses errors to distinct pixels and thus, it works correctly. Also, in 2-D PEC there is a pixel from which rounding error may be read at the same time. Since simultaneous reading in the asynchronous CRCW-PRAM is possible, it computes a binary image properly. On the other hand, in 2-D PED, there is a pixel to which two errors are diffused at the same time. So, we need to use several sidestep techniques to avoid simultaneous additions to the same pixel to get a binary image correctly.

For the purpose of avoiding simultaneous additions in 2-D PED, we can use extra BARRIER_SYNC not to execute addition operations to $a[i][j+1]$ and $a[i+1][j-1]$ by different calls of COMP(i, j) as follows:

[2-D PED with extra BARRIER_SYNC]

```
COMP$(i, j)$
{   if $a[i][j] \leq \frac{1}{2}$ then $r \leftarrow 0$ else $r \leftarrow 1$;
    $b[i][j] \leftarrow r$; $e \leftarrow a[i][j] - r$;
    $a[i][j+1] \leftarrow a[i][j+1] + \frac{7}{16} \cdot e$;
    BARRIER_SYNC;
    $a[i+1][j+1] \leftarrow a[i+1][j+1] + \frac{1}{16} \cdot e$;
    $a[i+1][j] \leftarrow a[i+1][j] + \frac{5}{16} \cdot e$;
    $a[i+1][j-1] \leftarrow a[i+1][j-1] + \frac{3}{16} \cdot e$; }
```

By BARRIER_SYNC, processors are stalled until all processors execute $a[i][j+1] \leftarrow a[i][j+1] + \frac{7}{16} \cdot e$. Since BARRIER_SYNC is also executed after COMP(i, j) is finished, processors are stalled until all processors execute $a[i+1][j-1] \leftarrow a[i+1][j-1] + \frac{3}{16} \cdot e$. Hence, additions to $a[i][j+1]$ and $a[i+1][j-1]$ are never executed at the same time.

We can use atomicAdd instruction to guarantee that the resulting values of additions are correct even if the two addition operations to the same pixel by different calls of COMP(i, j) are executed at the same time. Error diffusion operation is modified using atomicAdd as follows:

[2-D PED with atomicAdd]

```
COMP$(i, j)$
{   if $a[i][j] \leq \frac{1}{2}$ then $r \leftarrow 0$ else $r \leftarrow 1$;
    $b[i][j] \leftarrow r$; $e \leftarrow a[i][j] - r$;
    atomicAdd$(a[i][j+1], \frac{7}{16} \cdot e)$;
    $a[i+1][j+1] \leftarrow a[i+1][j+1] + \frac{1}{16} \cdot e$;
    $a[i+1][j] \leftarrow a[i+1][j] + \frac{5}{16} \cdot e$;
    atomicAdd$(a[i+1][j-1], \frac{3}{16} \cdot e)$; }
```

Even if two atomicAdd instructions are executed at the same

Table II
THE NUMBERS OF MEMORY ACCESS OPERATIONS TO THE MAIN MEMORY, THE NUMBER OF BARRIER SYNCHRONIZATIONS, AND THE NUMBER OF EXECUTED ATOMICADD INSTRUCTIONS, AND PARALLELISM OF PARALLEL SCAN (PS), PARALLEL ERROR DIFFUSION (PED), AND ERROR COLLISION (PEC) ON THE ASYNCHRONOUS CRCW-PRAM

	read	write	BARRIER_SYNC	atomicAdd	parallelism
k-D PS	-	-	$(k+1)\sqrt{n}$	-	$\frac{\sqrt{n}}{k}$
3-D PED	$5n$	$5n$	$4\sqrt{n}$	-	$\frac{\sqrt{n}}{3}$
2-D PED with extra BARRIER_SYNC	$5n$	$5n$	$6\sqrt{n}$	-	$\frac{\sqrt{n}}{2}$
2-D PED with atomicAdd	$5n$	$5n$	$3\sqrt{n}$	$6\sqrt{n}$	$\frac{\sqrt{n}}{2}$
2-D PEC	$5n$	$2n$	$3\sqrt{n}$	-	$\frac{\sqrt{n}}{2}$

time, additions to $a[i][j+1]$ and $a[i+1][j-1]$ are performed correctly.

Table II summarizes the performance of parallel algorithms presented in this section. We can see that 2-D PEC executes fewer BARRIER_SYNC instructions and no atomicAdd instruction, and higher parallelism. Note that if register caching technique is used, both read and write operations can be reduced to $2n$. Even if this is the case, the performance of 2-D PEC is better than the others in terms of the number of BARRIER_SYNC instructions and parallelism. Actually, in Section VI, we will show that 2-D PEC implemented on a GPU runs faster than the others.

IV. GPU IMPLEMENTATIONS OF PARALLEL ERROR DIFFUSION AND PARALLEL ERROR COLLECTION USING THE GLOBAL MEMORY

Let us implement parallel error diffusion and parallel error collection designed for the asynchronous PRAM to CUDA-enabled GPU. We assume that input gray-scale image a is stored in the global memory and the resulting binary image b is also written in the global memory.

Each thread in the GPU can work as a processor of the asynchronous CRCW-PRAM. We use m CUDA threads if parallelism is m. A CUDA block of CUDA compute capability 2.x or later can have up to 1024 threads [6]. If we use CUDA block with 32 threads, $\frac{m}{32}$ CUDA blocks are invoked. By a CUDA kernel call, CUDA blocks are invoked asynchronously, and there is no way to synchronize CUDA blocks during the execution of a kernel call. Hence, we need to use separate CUDA kernel calls for barrier synchronization. Further, since separate CUDA kernel calls cannot share values in registers, register caching technique cannot be applied.

If we implement k-D PS based algorithm using CUDA as it is, memory access to the global memory is not coalesced. As illustrated in Figure 3, addresses accessed by threads are not consecutive. For coalesced global memory access, we can permute pixels of input image a in advance as illustrated in Figure 4. More specifically, each row is shifted (row-wise shift) and then image is transposed. Row-wise shift can be done by reading every pixel and writing it to appropriate address in an obvious way. Transpose can be also done by block-wise reading/writing [21], [22]. Hence, data permutation for coalesced memory access can be done $2n$ reading and $2n$ writing operations with two kernel calls. Similarly, writing operations to output binary image b can be coalesced. For this purpose, we need to perform inverse data permutation after binary image b is obtained in the global memory.

Table III summarizes the numbers of memory access operations, barrier synchronizations, and processors performed by parallel error diffusion, parallel error collision and data permute for coalesced memory access. Please note that 2-D PED with extra BARRIER_SYNC has additional global memory read because extra kernel calls must read pixels in gray-scale image a. By executing data permutation for gray-scale image a and inverse data permutation for binary image b, parallel error diffusion and parallel error collision implementations can be changed not to perform non-coalesced global memory access.

V. PARALLEL ERROR DIFFUSION AND PARALLEL ERROR COLLECTION OPTIMIZED FOR GPUS

The main purpose of this section is to show parallel error diffusion and parallel error collection optimized for CUDA-enabled GPUs. We assume that image a are stored in the global memory of a GPU. We focus on the implementation of 2-delay parallel error diffusion (2-D PED) in a GPU. 3-delay and 2-delay parallel error diffusion can be implemented in the same way.

We use parameter w to denote the number of threads in a warp and the number of memory banks in the shared memory of a streaming multiprocessor. Each warp has 32 threads in CUDA for all compute capability, and each shared memory has 32 threads in CUDA compute capability 2.x or later [6]. Hence, we set $w = 32$ when we implement our parallel algorithms using CUDA for experiment. For theoretical analysis, we use parameter w for the number of threads in a warp and the number of memory banks in the shared memory.

In many GPU algorithms, images and matrices are partitioned into square blocks [6], [8], [10], [21]. Our new idea is to partition image a into parallelogram blocks with width w and height w, each of which has w rows with w pixels.

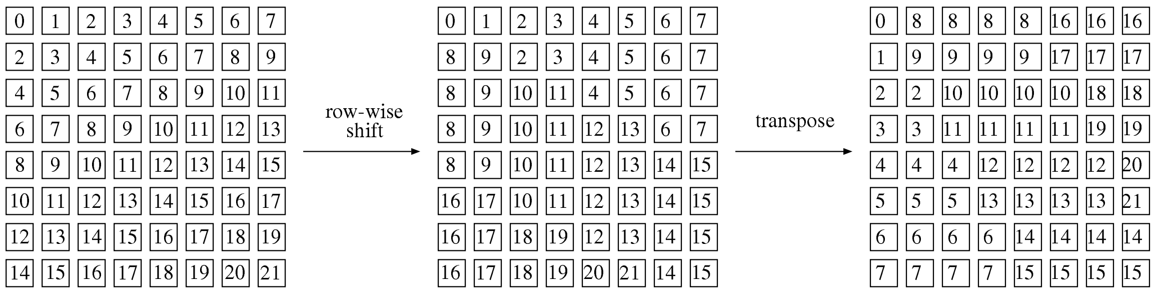

Figure 4. Data permutation for coalesced access using 2-delay parallel error diffusion

Table III
THE ANALYTICAL PERFORMANCE OF PARALLEL ERROR DIFFUSION (PED), PARALLEL ERROR COLLISION (PEC) AND DATA PERMUTATION FOR COALESCED MEMORY ACCESS ON A GPU

	global memory			kernel calls	atomicAdd	threads
	read	write	coalesced			
3-D PED	$5n$	$5n$	no	$4\sqrt{n}$	-	$\frac{\sqrt{n}}{3}$
2-D PED with extra BARRIER_SYNC	$5n$	$6n$	no	$6\sqrt{n}$	-	$\frac{\sqrt{n}}{2}$
2-D PED with atomicAdd	$5n$	$5n$	no	$3\sqrt{n}$	$3\sqrt{n}$	$\frac{\sqrt{n}}{2}$
2-D PEC	$5n$	$2n$	no	$3\sqrt{n}$	-	$\frac{\sqrt{n}}{2}$
Data permutation/inverse data permutation	$2n$	$2n$	yes	2	-	n

Figure 5 illustrates a parallelogram block for $w = 8$. Every row is shifted by two pixels to the left from the above row.

A gray-scale image a of size $\sqrt{n} \times \sqrt{n}$ is partitioned into parallelogram blocks as illustrated in Figure 6. First, image a is partitioned into $\frac{\sqrt{n}}{w}$ strips of w rows each. Each strip is further partitioned into $\frac{\sqrt{n}}{w} + 2$ parallelogram blocks. In the figure, 64×64 pixels are partitioned into 8 strips of 8 rows each. Each strip is partitioned into 10 parallelogram blocks. We use one CUDA block with w threads to execute 2-D PEC for a parallelogram block. For this purpose, a CUDA block copies pixels in a parallelogram blocks and additional pixels necessary to compute the resulting values of pixels in parallelogram blocks to the shared memory. CUDA block executes 2-D PEC using w threads.

Figure 5 illustrates how a parallelogram block and additional pixels in image a are copied to the shared memory. We use a 2-dimensional array of size $(w+1)\times(w+3)$ in the shared memory to store them. As illustrated in the figure, one row above the parallelogram block is necessary to collect errors. Also, three pixels to the left from the leftmost pixels of the parallelogram block are necessary. They are copied to the 2-dimensional array in the shared memory as illustrated in the figure, and 2-D PEC is executed using w threads in a CUDA block.

Suppose that, 2-D PEC is executed for a parallelogram block of gray-scale image a arranged on a 2-dimensional array of size $(w + 1) \times (w + 3)$ in the shared memory. Let $a'[i][j]$ $(0 \leq i \leq w, 0 \leq j \leq w + 2)$ be an element of the array. We can think that $a'[i][j]$ is arranged in address $i \cdot (w + 3) + j$, and thus, it is in memory bank $(i \cdot (w + 3) + j) \bmod w = (3i+j) \bmod w$ of the shared memory. It should be clear that w threads in a warp access to the same column of a'. For example, they access $a'[0][j], a'[1][j], \ldots, a'[w-1][j]$, which are in memory banks $(3 \cdot 0 + j) \bmod w, (3 \cdot 1 + j) \bmod w, \ldots, (3 \cdot (w - 1) + j) \bmod w$. Since w and 3 are relatively prime, these w memory banks are distinct. Hence, all memory access operations performed by 2-delay parallel error collection are conflict-free.

Binary image b is also stored in the shared memory. If b is arranged in the shared memory as it is, writing operations to the same column by w threads in a CUDA block cause bank conflicts. We can avoid bank conflicts if we use padding technique [6] or diagonal arrangement technique [21] easily.

After 2-D PEC for a parallelogram block of image a terminates, we need to write resulting values of the parallelogram block and resulting binary subimage to the global memory. We can copy the resulting binary subimage in the global memory in an obvious way. Note that it is not necessary to copy all values in the parallelogram block of image a. It is sufficient to copy values that will be used for later computation by the other blocks, that is, to copy additional pixels of the other CUDA blocks. Additional pixels consists of those in the last row and the rightmost three pixels in each row. In Figure 5, such pixels are highlighted. We have $w + 3 \cdot (w - 1) = 4w - 3$ additional pixels. We should arrange such pixels in a 1-dimensional array in the global memory for coalesced memory access.

We are now in a position to show how all parallelogram

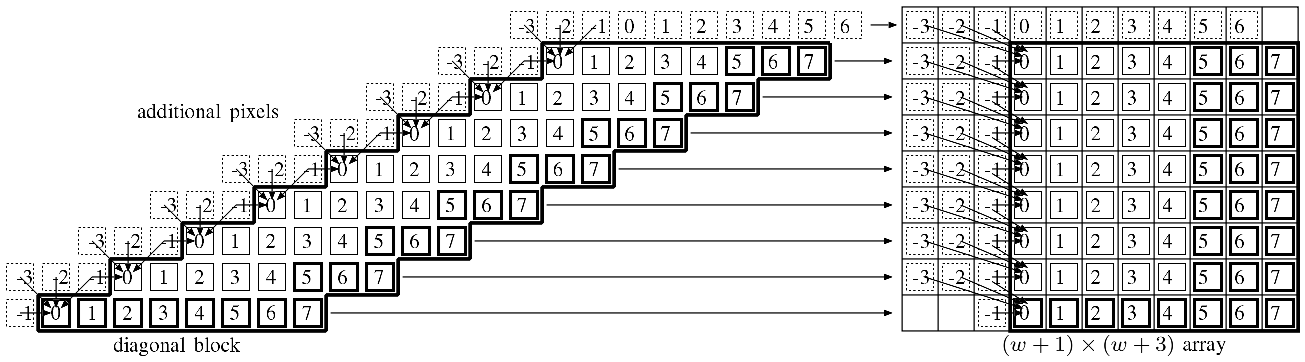

Figure 5. A diagonal block of image a and copy operation to the shared memory when $w = 8$

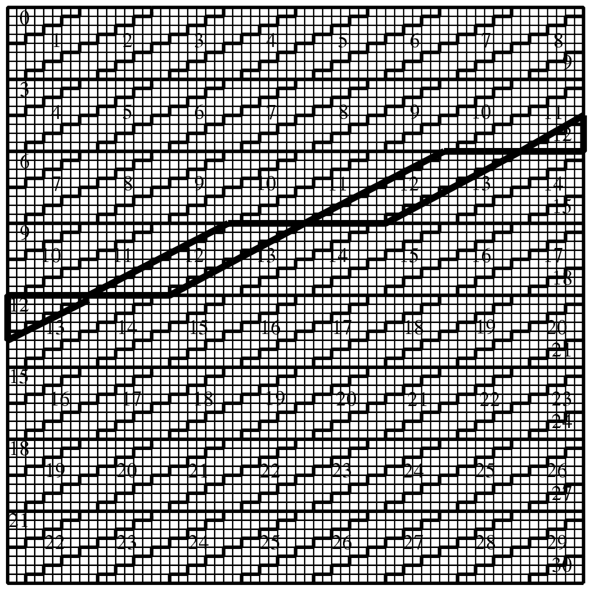

Figure 6. Partition of image a into parallelogram blocks for $w = 8$

blocks are processed. As illustrated in Figure 6, we assign integer labels to parallelogram blocks. In the figure, 10 parallelogram blocks of the first strip are assigned labels from 0 to 9. In general, $\frac{\sqrt{n}}{w} + 2$ parallelogram blocks of the i-th strip ($0 \leq i \leq \frac{\sqrt{n}}{w} - 1$) are assigned labels from $3i$ to $3i + \frac{\sqrt{n}}{w} + 1$. Let $\mathrm{PB}(i, j)$ denote a parallelogram block with label j in the i-th strip. It should be clear that 2-D PEC for parallelogram blocks with the same label is independent and can be executed at the same time. For example, in Figure 6, we can execute 2-D PEC for four parallelogram blocks with label 12 at the same time. Hence, we can execute 2-D PEC for all parallelogram blocks to obtain the resulting binary image of error collection as follows:

[2-D PEC on a GPU]

```
for $t \leftarrow 0$ to $4\frac{\sqrt{n}}{w} - 2$ do
    for $i \leftarrow 0$ to $\frac{\sqrt{n}}{w} - 1$ do in parallel
        if $3i \leq t \leq 3i + \frac{\sqrt{n}}{w} + 1$ then
            A CUDA block executes 2-D for $\mathrm{PB}(i, t)$
```

For a particular t, the number of values i satisfying $3i \leq t \leq 3i + \frac{\sqrt{n}}{w} + 1$ is at most $\frac{\frac{\sqrt{n}}{w}+1}{3} + 1$ ($\approx \frac{\sqrt{n}}{3w}$). Hence, approximately $\frac{\sqrt{n}}{3w}$ CUDA blocks are used to execute 2-D PEC in parallel. Also, for each t, one CUDA kernel call is used to synchronize the computation, because 2-D PEC for parallelogram blocks with label $t+1$ can be started only after that for parallelogram blocks with label t is completed. Hence, approximately $4\frac{\sqrt{n}}{w}$ CUDA kernel calls are invoked.

Let us evaluate the number of memory access operations to the global memory. We assume that image a is so large that $w \ll n$ holds. A CUDA block for $\mathrm{PB}(i, t)$ reads $(w + 1) \times (w + 3)$ pixels of gray-scale image a in the global memory. Since we have $\frac{\sqrt{n}}{w} \cdot (\frac{\sqrt{n}}{w} + 2) = \frac{n}{w^2} + 2\frac{\sqrt{n}}{w}$ parallelogram blocks, read operation is performed for $(\frac{n}{w^2} + 2\frac{\sqrt{n}}{w}) \cdot (w + 1) \cdot (w + 3) = n + O(\frac{n}{w})$ pixels in the global memory. After the computation of 2-D PEC, a CUDA block writes $4w - 3$ values of a and w^2 values of b in the global memory. Hence, write operation is performed $(\frac{n}{w^2} + 2\frac{\sqrt{n}}{w}) \cdot (w^2 + 4w - 3) = n + O(\frac{n}{w})$ pixel values in the global memory.

Similarly to 2-D PEC, we can implement 2-D PED using parallelogram blocks. The readers may think that extra BARRIER_SYNC or atomicADD operation are necessary to avoid simultaneous additions. However, it is guaranteed that w threads in the same warp work synchronously [6], that is, they always execute machine instruction in the same address. Hence, we do not have to use such sidestep techniques for 2-D PED. Also, please note that register caching (RC) technique can be used for computation on parallelogram blocks. Table IV summarizes analytical evaluation of 2-D PED and 2-D PEC with and without RC.

Table IV
THE ANALYTICAL PERFORMANCE OF 2-DELAY PARALLEL ERROR DIFFUSION (2-D PED) AND 2-D PARALLEL ERROR CORRECTION (2-D PEC) WITH AND WITHOUT REGISTER CACHING (RC)

	global memory		shared memory		register caching		kernel calls	threads
	read	write	read	write	read	write		
2-D PED	$n+O(\frac{n}{w})$	$n+O(\frac{n}{w})$	$6n+O(\frac{n}{w})$	$6n+O(\frac{n}{w})$	-	-	$\frac{4\sqrt{n}}{w}$	$\frac{\sqrt{n}}{3}$
2-D PED with RC	$n+O(\frac{n}{w})$	$n+O(\frac{n}{w})$	$3n+O(\frac{n}{w})$	$3n+O(\frac{n}{w})$	$9n$	$10n$	$\frac{4\sqrt{n}}{w}$	$\frac{\sqrt{n}}{3}$
2-D PEC	$n+O(\frac{n}{w})$	$n+O(\frac{n}{w})$	$6n+O(\frac{n}{w})$	$3n+O(\frac{n}{w})$	-	-	$\frac{4\sqrt{n}}{w}$	$\frac{\sqrt{n}}{3}$
2-D PEC with RC	$n+O(\frac{n}{w})$	$n+O(\frac{n}{w})$	$3n+O(\frac{n}{w})$	$3n+O(\frac{n}{w})$	$9n$	$7n$	$\frac{4\sqrt{n}}{w}$	$\frac{\sqrt{n}}{3}$

VI. EXPERIMENTAL RESULTS

This section shows experimental results for sequential algorithms and parallel algorithms presented in this paper. We use Intel Core-i7 3770K (3.5GHz) for evaluating sequential algorithms and GeForce GTX 780Ti for evaluating parallel algorithms. We have used 8-bit "unsigned char" for input gray-scale image a and output binary image b. Since most gray-scale images have 8-bit depth, it makes sense to use 8-bit unsigned integers. Also, we use 32-bit "unsigned int" to store intermediate pixel values as fixed-point numbers. Table V shows the running time of sequential/parallel algorithms for images of size from 1K$\times$1K (1024$\times$1024) to 16K$\times$16K (16384$\times$16384).

Table V (1) shows the running time of sequential algorithms for error diffusion (ED) and error collection (EC) with and without register caching (RC). From the table, we can see that EC with RC runs faster than the others because it performs fewest memory access operations.

Table V (2) shows the performance of parallel algorithms designed for the asynchronous CRCW-PRAM. They are implemented in the GPU using the global memory as they are. Hence, it performs a lot of non-coalesced memory access to the global memory of the GPU. Clearly, 2-D PEC runs faster than the others. By comparing the running time of PED-based algorithms, we can see that kernel calls have large overhead, while the overhead of atomicAdd is small. Hence, we should minimize the number of kernel calls to improve the performance.

As we have shown in Section IV, we can avoid non-coalesced memory access if data permutation for input gray-scale image and output binary image is used. Table V (3) shows the running time of each parallel algorithms if data permutation is used. The running time includes that for data permutation of input gray-scale image and output binary image. By comparing (2) and (3) in the table, we can see that non-coalesced access to the global memory has very large overhead.

Table V (4) shows the running time of 2-D PED and 2-D PEC optimized for the CUDA-enabled GPU architecture. Since the global memory access is minimized, they run much faster than parallel algorithms designed for the asynchronous CRCW-PRAM. As we have shown in analytical performance, 2-D PEC with RC runs faster than the others. In particular, the running time of 2-D PEC with RC runs 46.75ms for 256M pixels, while the best sequential algorithm EC with RC runs 2052ms. Thus, 2-D PEC with RC achieves a speedup factor of 43.9.

Table V (5) shows the time necessary to transfer images between the host PC and the GPU. We can say that, even if the data transfer time is included, 2-D PEC with RC runs approximately 151ms for 256M pixels. Hence, our GPU implementation is practically fast.

VII. CONCLUSION

The main contribution of this paper is to present several algorithmic techniques for error diffusion. Although error diffusion involves sequential operations that scan an input image in raster scan order, our new technique that partition the image into parallelogram blocks can extract enough parallelism. Our parallel algorithm optimized for CUDA-enabled GPUs runs 43.9 times faster than the best sequential algorithm on a single Intel CPU. Even if the data transfer time between the host PC and the GPU is included, it runs more than 13.5 times faster than the best sequential algorithm.

REFERENCES

[1] D. L. Lau and G. R. Arce, *Modern Digital Halftoning, Second Edition.* CRC Press, 2008.

[2] R. W. Floyd and L. Steinberg, "An adaptive algorithm for spatial gray scale," *SID 75 Digest,Society for Information Display*, pp. 36–37, 1975.

[3] B. E. Bayer, "An optimum method for two-level rendition of continuous-tone pictures," in *proc. of IEEE International Conference on Communications*, vol. 1, June 1973, pp. 11–15.

[4] W. W. Hwu, *GPU Computing Gems Emerald Edition.* Morgan Kaufmann, 2011.

[5] R. Farber, *CUDA Application Design and Development.* Elsevier, 2011.

[6] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 7.0," Mar 2015.

Table V
THE RUNNING TIME (IN MS) OF SEQUENTIAL/PARALLEL ERROR DIFFUSION AND ERROR COLLECTION

$\sqrt{n} =$	1K	2K	3K	4k	5K	6K	7K	8K	10K	12K	14K	16K
(1) Sequential algorithms on the Intel CPU												
ED	19.63	77.63	174.4	309.8	491.5	696.4	952.0	1242	1938	2789	3797	4967
EC	19.60	74.21	165.8	293.9	467.7	663.6	902.5	1177	1846	2654	3671	4712
ED with RC	14.72	55.38	124.9	220.2	357.5	505.7	676.6	878.1	1384	1974	2690	3521
EC with RC	**8.616**	**33.14**	**73.43**	**128.8**	**201.2**	**292.4**	**393.3**	**514.2**	**802.8**	**1165**	**1571**	**2052**
(2) Parallel algorithms designed for the asynchronous CRCW-PRAM												
3-D PED (low parallelism)	32.23	68.50	99.50	140.5	167.3	232.2	257.2	304.3	383.0	501.1	613.8	805.8
2-D PED with atomicAdd	23.43	56.14	78.35	107.4	128.3	165.4	199.5	233.2	306.2	421.1	589.1	796.1
2-D PED with double kernel calls	40.49	82.42	127.1	173.8	224.4	282.6	332.9	403.8	528.7	691.3	941.9	1254
2-D PEC	**17.41**	**35.27**	**53.16**	**78.28**	**115.3**	**136.9**	**158.9**	**191.6**	**261.7**	**357.1**	**476.2**	**650.5**
(3) Parallel algorithms on the GPU using the global memory with data permutation												
3-D PED (low parallelism)	18.77	37.56	57.76	77.40	99.29	119.8	150.5	162.5	212.0	258.2	305.0	355.5
2-D PED with atomicAdd	14.06	28.37	42.35	55.63	71.12	86.75	101.6	127.0	155.0	209.9	227.9	266.8
2-D PED with double kernel calls	24.26	49.43	81.12	98.80	125.7	186.8	195.8	203.9	266.8	353.2	404.1	448.6
2-D PEC	**14.28**	**26.10**	**39.10**	**51.33**	**65.92**	**85.26**	**94.04**	**118.0**	**146.5**	**176.5**	**207.7**	**260.6**
(4) Parallel algorithms on the GPU using the global memory and the shared memory												
2-D PED	2.192	4.559	6.944	9.298	11.66	14.13	16.64	19.09	24.21	29.74	35.27	56.27
2-D PED with RC	1.891	3.945	5.976	7.993	10.04	12.19	14.30	16.45	24.26	29.59	35.05	55.77
2-D PEC	2.009	4.365	6.519	8.748	10.94	13.37	15.65	18.12	22.98	28.88	34.26	51.36
2-D PEC with RC	**1.847**	**3.919**	**6.000**	**8.097**	**10.08**	**12.26**	**14.25**	**16.63**	**21.07**	**26.33**	**31.32**	**46.75**
(5) Data transfer between the host and the GPU												
Host$\rightarrow$ GPU	0.2979	0.8501	1.834	3.346	4.782	6.855	9.245	12.39	18.92	29.82	38.43	49.83
GPU$\rightarrow$ Host	0.3107	0.8529	1.828	3.389	4.896	7.386	9.475	12.36	19.95	30.85	39.97	54.75

[7] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[8] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[9] K. Nakano and S. Matsumae, "The super warp architecture with random address shift," in *Proc. of High Performance Computing (HiPC)*, Dec. 2013, pp. 256–265.

[10] A. Kasagi, K. Nakano, and Y. Ito, "Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations," in *Proc. of International Conference on Parallel Processing (ICPP)*, Sept. 2014, pp. 251–250.

[11] C. U. Martel, R. Subramonian, and A. Park, "Asynchronous PRAMs are (almost) as good as synchronous prams," in *Proc. of Symposium on Foundations of Computer Science*, vol. 2, 1990, pp. 590 – 599.

[12] P. T. Metaxas, "Parallel digital halftoning by error-diffusion," in *Proc. of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge*, 2003, pp. 35– 41.

[13] H. Kouge, Y. Ito, and K. Nakano, "A GPU implementation of clipping-free halftoning using the direct binary search," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (LNCS 8630)*, Aug. 2014, pp. 57–70.

[14] Y. Zhang, J. L. Recker, R. Ulichney, I. Tastl, and J. D. Owens, "Plane-dependent error diffusion on a GPU," in *Proc. SPIE*, vol. 8295, Jan. 2012.

[15] K. Chandu, M. Stanich, B. Trager, and C. W. Wu, "A GPU implementation of color digital halftoning using the direct binary search algorithm," in *International Symposiun on Circuits and Systems*, May 2012, pp. 185 – 188.

[16] A. Deshpande, I. Misra, and P. J. Narayanan, "Hybrid implementation of error diffusion dithering," in *Proc. of International Conference on High Performance Computing (HiPC)*, 2011, pp. 1 – 10.

[17] Y. Zhang, J. Recker, R. Ulichney, G. Beretta, I. Tastl, I.-J. Lin, and J. D. Owens, "A parallel error diffusion implementation on a GPU," in *Proceedings of SPIE*, vol. 7872, Jan. 2011.

[18] P. Li and J. P. Allebach, "Block interlaced pinwheel error diffusion," *Journal of Electronic Imaging*, vol. 14, no. 2, June 2005.

[19] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[20] J. JáJá, *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[21] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2012, pp. 788–797.

[22] A. Kasagi, K. Nakano, and Y. Ito, "An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation," in *Proc. of International Conference on Parallel Processing (ICPP)*, Oct. 2013, pp. 1–10.