Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186–5140
Volume 4, Number 1, pages 9–14, January 2015

# Implementations of Parallel Error Diffusion Optimized for GPUs (Preliminary version)

Akihiko Kasagi, Koji Nakano, and Yasuaki Ito
*Department of Information Engineering, Hiroshima University*
*Kagamiyama 1-4-1, Higashi-hiroshima, 739-8527 Japan*

***Abstract*—Error diffusion is a well-known algorithm used for converting a gray-scale image into a binary image. The main contribution of this paper is to present our new error diffusion algorithm optimized for CUDA-enabled GPUs. Our algorithm partitions the gray-scale image into parallelogram blocks and performs error diffusion operation for them in parallel. We have implemented several error diffusion algorithms on GeForce GTX 780 Ti. The experimental results show that our algorithm runs faster than any other error diffusion algorithms for any input matrices. Also, it runs 44 times faster than the sequential error diffusion algorithm using a single CPU.**

***Keywords*-error diffusion, GPU, CUDA**

## 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. 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.

Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPU has recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [4], the computing engine for NVIDIA GPUs.

CUDA-enabled GPUs have streaming multiprocessors (SMs) each of which executes multiple threads in parallel. CUDA can use two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [4]. Each SM has the shared memory, an extremely fast on-chip memory with lower capacity, say, 16-48 KBytes, and low latency. As illustrated in Figure 1, every SM shares the global memory implemented as an off-chip DRAM with 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 for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access and *the bank conflict* of the shared memory access [4],[5],[6].

CUDA parallel programming model has a hierarchy of thread groups called *grid*, *block* and *thread*. A single grid is organized by multiple blocks, each of which has equal number of threads. The blocks are allocated to streaming processors such that all threads in a block are executed by the same streaming processor in parallel. All threads can access the global memory. However, threads in a block can access the shared memory of the streaming processor to which the block is allocated. Since blocks are arranged to multiple streaming processors, threads in different blocks cannot share data in shared memories. In the execution, 32 threads in a block are split into groups of thread called *warps*. Each of these warps contains the same number of threads and is executed independently. When a warp is selected for execution, all threads execute the same instruction.

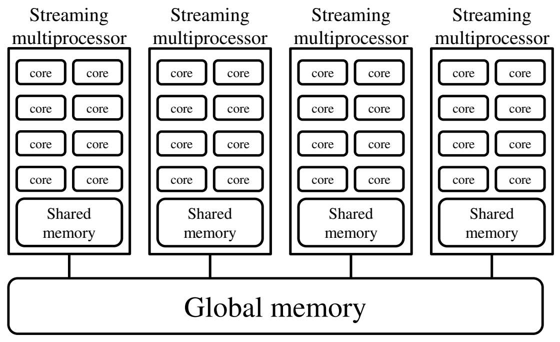

Figure 1. High-level GPU architecture

When threads in a warp access consecutive address in the global memory, the address space of the global memory can be accessed at the same time(*coalesced access*). However, if threads in a warp access distant address space in the global memory, these memory requests are divided into the several times(*stride access*). Since the coalesced access maximizes the bandwidth of memory access, we should avoid the stride access and perform the coalesced access whenever possible.

In the GPU architecture. shared memory is separated into 32 banks. The successive words are assigned to successive banks. If threads in a warp access distinct banks in the shared memory, the access can be serviced simultaneously. On the other hand, if threads access the same bank, the access has to be serialized(*bank conflict*). So, we should access distinct address and avoid the bank conflict to maximize throughputs. In addition, CUDA supports barrier synchronization and atomic operations such as atomicAdd. Since barrier synchronization and atomic operations impose certain overhead cost, these operations should be avoided when we design parallel algorithms.

Error diffusion is a method for generating a binary image that reproduces a gray-scale image. In error diffusion, pixel values are rounded to binary in raster scan order and the rounding error is distributed to neighboring four pixels that have not yet been processed. Let $a$ be a gray-scale image of size $\sqrt{n}\times\sqrt{n}$ such that pixel a[i][j]($0 \le i,j \le \sqrt{n}-1$) takes an intensively level(i.e. a real number) in the range[0,1]. *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)]**

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

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.

In this paper, we show our new method, *error collection*, which outputs exactly the same binary image as error diffusion. 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)]**

```
$i \leftarrow 0$ 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 $s \le \frac{1}{2}$ 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. Since the value of each binary pixel $b[i][j]$ can be determined by a constant number of instructions, both algorithms run $O(n)$ time for an input image with $n$ pixels.

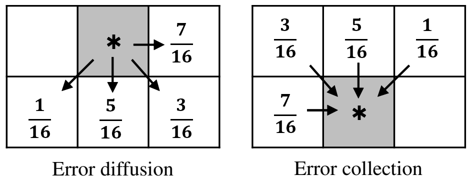

Figure 2. Error diffusion and error collection

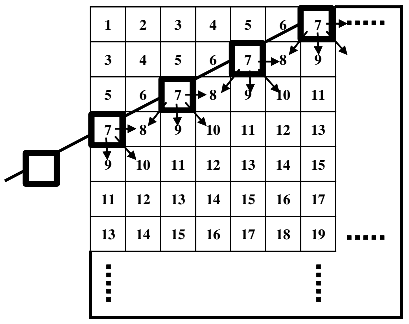

Figure 3. Parallel error diffusions

In [7][8], Metaxas parallelized error diffusion which assigns each of processors to each row of an input image and execute error diffusion operation from left to right as illustrated in Figure 3. A variant of error diffusion called pinwheel error diffusion [9] have been implemented in a GPU [10]. 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 16Mpixel halftoning for $14\times14$ 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 main contribution of this paper is to show the efficient algorithms, *block-wise error diffusion(BWED)* and *block-wise error collection(BWEC)*, which partitions an input image into the parallelogram blocks and performs the error diffusion/collection in block-wise. We have implemented several algorithms on GeForce GTX 780 Ti. The experimental results show that our BWEC runs faster than any other error diffusion algorithms. We have also implemented sequential algorithms on Intel Core-i7 3770K. The result shows that our algorithm runs 44 times faster than the sequential error diffusion algorithm using a single CPU.

## II. PWED and PWEC algorithms

The main purpose of this section is to show the *pixel-wise error diffusion algorithm (PWED algorithm)* and *pixel-wise error collection algorithm (PWEC algorithm)*. PWED algorithm assigns processors to each row and executes error diffusion from left to right as illustrated in Figure 3. Given a gray-scale image $a$ of the size $\sqrt{n} \times \sqrt{n}$, a binary image $b$ of the same size can be computed as follows:

**[PWED algorithm]**

```
for $k \leftarrow 0$ to $3(\sqrt{n}-1)$ do
  for $i \leftarrow 0$ to $\sqrt{n}-1$ do in parallel
    $j = k - 2i$;
    if $0 \le j \le \sqrt{n}-1$ then
      DIFFUSE$(i,j)$;
    barrier_synchronization;
```

In this algorithm, DIFFUSE$(i,j)$ denotes some computation for $a[i][j]$ and $b[i][j]$. Let us see how PWED algorithm works. First, when $k = 0$ a processor assigned to the first row performs DIFFUSE(0,0). After that, this processor performs DIFFUSE(0,1), DIFFUSE(0,2), ..., DIFFUSE(0,$\sqrt{n}-1$) one by one. When $k = 2$, it performs DIFFUSE(0,2), and a processor assigned to the second row performs DIFFUSE(1,0). The same procedure repeated until DIFFUSE($\sqrt{n}-1, \sqrt{n}-1$) is performed when $k = 3(\sqrt{n}-1)$. Hence, PWED algorithm performs barrier synchronization $3(\sqrt{n}-1)$ times. Since the the size of input image $a$ is $\sqrt{n} \times \sqrt{n}$, pixels in at most $\lfloor \frac{\sqrt{n}-1}{2} \rfloor + 1$ consecutive rows are processed in parallel. In PWED algorithm, we must use atomicAdd instruction to guarantee that the resulting values of additions are collect even if the two addition operations to the same pixel by different calls of DIFFUSE$(i,j)$ are executed at the same time. Thus, the details of DIFFUSE$(i,j)$ are spelled out as follows:

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

Even if DIFFUSE$(i,j)$, DIFFUSE$(i+1, j-2)$ and DIFFUSE$(i-1, j+2)$ are executed at the same time, additions to $a[i+1][j-1]$ and $a[i][j+1]$ are performed correctly. Clearly, each pixel is performed 5 read operations and 5 write operations. For exemple, when DIFFUSE$(i,j)$ is executed, a processor reads $a[i][j], a[i][j+1], a[i+1][j-1], a[i+1][j]$ and $a[i+1][j+1]$, and writes $b[i][j], a[i][j+1], a[i+1][j-1], a[i+1][j]$ and $a[i+1][j+1]$. However, each memory access request is not consecutive address. Thus, PWED algorithm performs $10n$ stride memory access operations.

Similarly to PWED algorithm, PWEC algorithm assigns processors to each row and executes error collection from left to right. Given a gray-scale image $a$ of the size $\sqrt{n} \times \sqrt{n}$, a binary image $b$ of the same size can be computed as follows:

**[PWEC algorithm]**

```
for $k \leftarrow 0$ to $3(\sqrt{n}-1)$ do
  for $i \leftarrow 0$ to $\sqrt{n}-1$ do in parallel
    $j = k - 2i$;
    if $0 \le j \le \sqrt{n}-1$ then
      COLLECT$(i,j)$;
    barrier_synchronization;
```

In this algorithm, COLLECT$(i,j)$ denotes some computation as follows:

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

This algorithm performs barrier synchronization $k = 3(\sqrt{n}-1)$ times in the same way as PWED algorithm However, PWEC algorithm has two differences from PWED algorithm. First, PWEC algorithm has no atomic operations. Even if the two read operations to the same pixel by different calls of COLLECT$(i,j)$ are executed at the same time, read operations to $a[i-1][j]$ and $a[i-1][j+1]$ are performed correctly. Second, PWEC algorithm performs fewer memory access operations than PWED algorithm. For example, when COLLECT$(i,j)$ is executed, a processor reads $a[i][j], a[i-1][j-1], a[i-1][j], a[i-1][j+1]$ and $a[i][j-1]$, and writes $b[i][j]$ and $a[i][j]$. So, PWEC algorithm performs 5 read operations and 2 write operations per pixels. However, each memory access request is not consecutive address. Thus, PWED algorithm performs $7n$ stride memory access operations.

## III. Our BWED and BWEC algorithms

The main purpose of this section is to show our novel algorithms: *block-wise error diffusion algorithm (BWED algorithm)* and *block-wise error collection algorithm (BWEC algorithm)* The idea is to separate an input image into several parallelogram blocks and execute error diffusion/collection in block-wise.

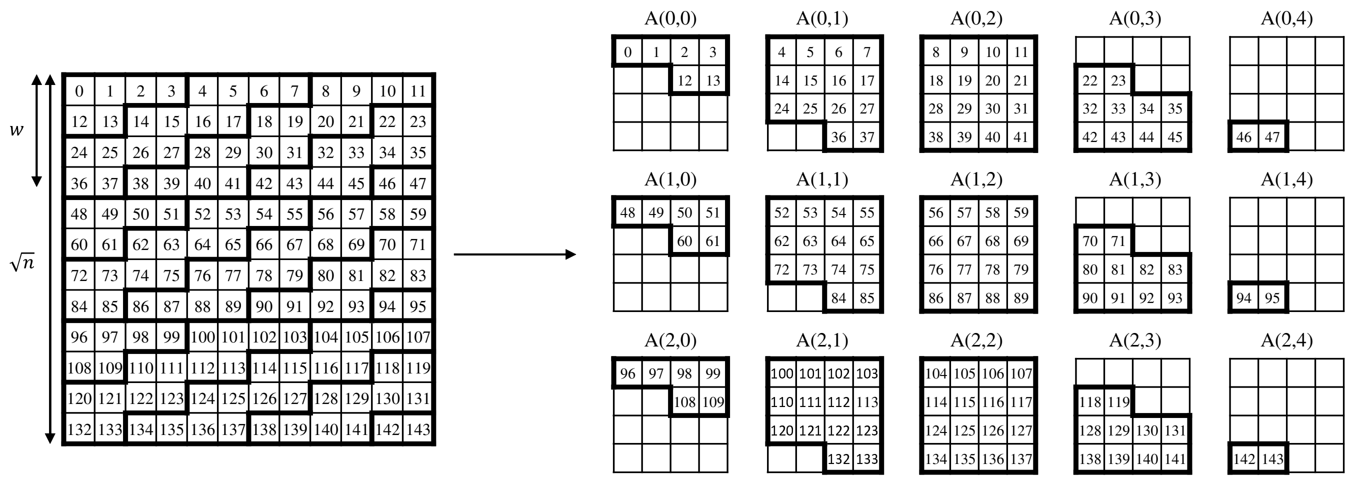

Figure 4. Assignment of an input image in our algorithm with $w = 4$

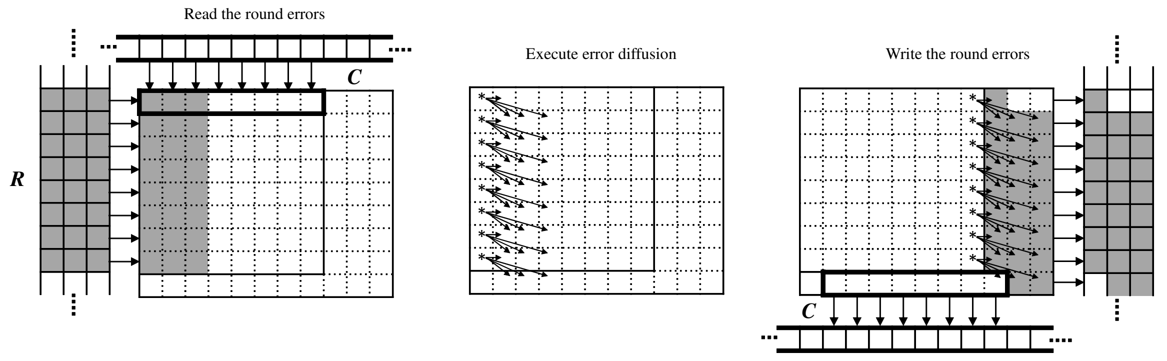

Figure 5. Computation of BWED algorithm in the shared memory with $w = 8$

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. Hence, we set $w = 32$ when we implement our parallel algorithm using CUDA for experiment. Support that $a$ is an input image of the size $\sqrt{n} \times \sqrt{n}$. In our algorithms, the input image $a$ of the size $\sqrt{n} \times \sqrt{n}$ is partitioned into $(\frac{\sqrt{n}}{w} + 2) \times \frac{\sqrt{n}}{w}$ blocks of the size $w \times w$ each. Let $A$ be CUDA blocks in this algorithm. Then, each pixels $a[i][j](0 \leq i, j \leq \sqrt{n} - 1)$ is assigned a block of $A(y, x)$ ($y = \frac{i}{w}$, $x = \frac{j - 3\cdot(i \bmod w)}{w}$). Figure 4 is shown an example with $w = 4$ that an image of the size $12 \times 12$ is assigned $5 \times 3$ parallelogram blocks of the size $4 \times 4$.

In error diffusion and error collection, the pixel $a[i][j]$ depends on its neighboring processed pixels $a[i-1][j-1]$, $a[i][j-1]$, $a[i+1][j-1]$ and $a[i-1][j]$. Therefore, in the block-wise computation, block of $A(y, x)$ depends on its neighboring processed blocks of $A(y-1, x+1)$, $A(y-1, x+2)$ and $A(y, x-1)$. Thus, the details of BWED algorithm and BWEC algorithm are spelled out as follows:

**[BWED and BWEC algorithm]**

```
for $k \leftarrow 0$ to $4\frac{\sqrt{n}}{w} - 2)$ do
  for $i \leftarrow 0$ to $\frac{\sqrt{n}}{w} - 1$ do in parallel
    if $3i \leq k \leq 3i + \frac{\sqrt{n}}{w} + 1$ then
      block$(i, k)$ executes COMP$(i, k)$
```

For a particular $k$, the number of values $i$ satisfying $3i \leq t \leq 3i + \frac{\sqrt{n}}{w} + 1$ is at most $\frac{\frac{\sqrt{n}}{w}+1}{3w} + 1$. Hence, approximately $\frac{\frac{\sqrt{n}}{w}+1}{3w}$ CUDA blocks are used to executes COMP$(i, k)$. COMP$(i, k)$ denotes some computation for block$(i, k)$. In the function of COMP$(i, k)$, 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. When we execute error diffusion in the shared memory, as illustrated in the Figure 5, first row in the parallelogram block is necessary to collect errors. Also three pixels from leftmost pixels of the parallelogram blocks are necessary. They are copied to the 2-dimensional array in the shared memory from array *C* and *R* with coalesced access. The array *C* of the size $\sqrt{n}$ and the array *R* of the size $3 \times \sqrt{n}$ are stored round errors which is written by the latest computed blocks. For example, The element of *C*$[j]$ $(0 \leq j \sqrt{n}-1)$ corresponds to the $j$-th column round error which is written by the latest computed blocks. Similarly, The three element of *R*$[0][i]$, *R*$[1][i]$ and *R*$[2][i]$ $(0 \leq i \sqrt{n}-1)$ correspond to the $i$-th row round errors which are written by the latest computed blocks.

Suppose that, BWED 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 $(i \cdot (w+3)+j)$ mod $w = (3i+j)$ mod $w$ of the shared memory. It should be clear that $w$ threads in a warp access the same column of $a'$. For example, they access $a'[0][j]$, $a'[1][j]$, ... ,$a'[w-1][j]$, which are in memory banks $(3 \times 0 + j)$ mod $w$, $(3 \times 1 + j)$ mod $w$, ... ,$(3 \times (w-1) + j)$ mod $w$. Since $w$ and 3 are relatively prime, these $w$ memory banks are distinct. Hence, all memory access operations performed parallel error diffusion are conflict-free.

Binary image $b$ of the size $\sqrt{n} \times \sqrt{n}$ is also stored in the shared memory. If $b$ is arranged in the shared memory as it is, writing operations of the same column by $w$ threads in a CUDA block caused bank conflict. We can avoid bank conflict if we use padding technique [4] or diagonal arrangement technique [11]. After that, we can copy the resulting binary image of the size $\sqrt{n} \times \sqrt{n}$ in the global memory. Also, as illustrated in Figure 6, we can copy the round errors $4w$ to the array *R* and the array *C*.

Let us evaluate the number of memory access operations to the global memory. Each block reads $(w \times w) + 4w$ elements in the global memory. Also, Each block writes $(w \times w) + 4w$ elements to the global memory. Since we have $(\frac{\sqrt{n}}{w}+2) \times \frac{\sqrt{n}}{w}$ parallelogram blocks, access operations to the global memory is performed $2 \cdot (\frac{\sqrt{n}}{w}+2) \times \frac{\sqrt{n}}{w}) \cdot (2w^2+8w)$ $= 2n + O(\frac{n}{w})$ times in BWED algorithm.

Similarly to BWED algorithm, we can implement BWEC algorithm using parallelogram blocks.

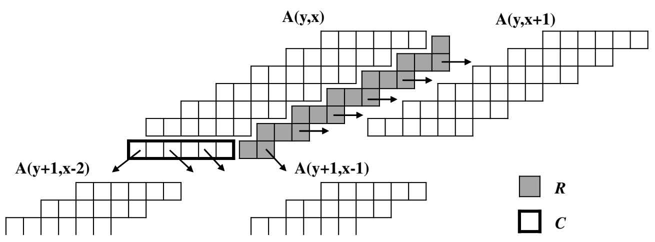

Figure 6. Copy the round errors to the array *R* and *C*

## IV. Experimental results

This section shows experimental results for sequential algorithm and parallel algorithms presented in this paper. We use Intel Core-i7 3770K (3.5GHz) for evaluating sequential algorithm and GeForce GTX 780Ti for evaluating parallel algorithms.

**[ED(CPU) algorithm]:** This algorithm performs error diffusion in a raster scan order. We use *register cache technique* to reduce the number of memory access operations. In this 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, the result values of $a[i+1][j-1]$ 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.

**[EC(CPU) algorithm ]:** This algorithm performs error collection in a raster scan order. Similarly to ED(CPU), we use five additional register to cache $a[i-1][j+1]$, $a[i-1][j]$, $a[i-1][j-1]$, $a[i][j]$ and $a[i-1][j]$ for error collection.

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. Also, barrier synchronization of all threads in a CUDA is implemented by invoking separated CUDA kernel calls. We have tested several configuration in terms of the number of threads in a CUDA blocks, and selected the best configuration. Table I shows the running time of error diffusion algorithm for a image of size from 1K $\times$ 1K $(= 1024 \times 1024)$ to 16K $\times$ 16K $(16384 \times 16384)$.

Table I shows the running time of sequential algorithms for EC(CPU) algorithm and ED(CPU) algorithm. From the table, we can see that EC(CPU) algorithm runs faster than ED(CPU) algorithm because it performs fewest memory access operations.

Table I also shows the running time of parallel algorithms on the GPU. Since PWEC algorithm and PWED algorithm performs a lot of kernel calls and stride memory access, they have large memory access latency overhead. PWEC algorithm runs faster than PWED algorithm for large input images because error collection performs fewer access operations than error diffusion. Also, we can see the overhead of atomic operations is small in the error diffusion. Since the global memory access is minimized, BWEC algorithm runs faster than any other algorithm for any size of image.

## V. Conclusion

The main contribution of this paper is to present several algorithmic technique 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. We have also implemented all algorithms, and

Table I
EXECUTION TIME OF ERROR DIFFUSION ALGORITHMS ON GPU AND CPU [MS]

| Algorithms | 1K | 2K | 3K | 4K | 5K | 6K | 7K | 8K | 9K | 10K | 12K | 14K | 16K |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PWED | 20.89 | 42.22 | 63.69 | 85.53 | 108.64 | 129.52 | 153.74 | 189.16 | 231.57 | 281.58 | 381.31 | 554.83 | 737.34 |
| PWEC | 20.10 | 40.36 | 62.01 | 90.27 | 119.84 | 145.58 | 155.76 | 187.6 | 218.08 | 251.91 | 356.98 | 472.74 | 650.28 |
| BWED | 1.89 | 3.94 | 6.05 | 7.99 | 10.16 | 12.19 | 14.30 | 16.45 | 18.92 | 24.26 | 29.59 | 35.05 | 55.77 |
| BWEC | 1.85 | 3.92 | 6.01 | 8.10 | 9.85 | 12.01 | 14.25 | 16.63 | 18.77 | 21.07 | 26.33 | 31.32 | 46.75 |
| ED(CPU) | 14.24 | 56.82 | 127.1 | 222.1 | 354.2 | 500.8 | 682.8 | 892.1 | 1129 | 1396 | 2020 | 2728 | 3555 |
| EC(CPU) | 8.29 | 32.79 | 73.77 | 130.8 | 203.2 | 294.2 | 397.5 | 526.5 | 659.2 | 821.9 | 1175 | 1598 | 2092 |

the experimental result on GeForce GTX 780 Ti shows that our BWEC algorithm runs faster than any other algorithm for any size of matrix. It also runs 44 times faster than the best sequential algorithm running on Intel Core-i7 3770K.

## REFERENCES

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

[2] Robert W. Floyd and L. Steinberg, "An Adaptive Algorithm for Spatial Grayscale," in *Proc. of the Society for Information Display 17*, 1976 , pp. 75–77.

[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, 1973 , pp. 11–15.

[4] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 5.0," 2012.

[5] 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.

[6] 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.

[7] Panagiotis T. Metaxas, "Optimal Parallel Error-Diffusion Dithering" in *Proc. of SPIE*, 1999.

[8] Panagiotis T. Metaxas, "Parallel digital halftoning by error-diffusion" in *Proc. of PCK50*, 2003.

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

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

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