A Square Pointillism Image Generation, and its GPU Acceleration

Hiroki Tokura, Yuki Kuroda, Yasuaki Ito, and Koji Nakano
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 Japan
Email: {tokura, kuroda, yasuaki, nakano}@cs.hiroshima-u.ac.jp

***Abstract*—Non-photorealistic rendering is one of the digital art techniques. It generates digital images resembling artistic representation. The main contribution of this paper is to show a non-photorealistic rendering for high quality pointillism image generation with squares by pasting square patterns on canvas. Our technique is inspired by the characteristic of the human visual system to optimize generated images. Although it can generate high quality pointillistic images, a lot of time is necessary. Hence, we have implemented our technique in a graphics processing unit (GPU) to accelerate the computation. The experimental results show that the GPU implementation can achieve a speed-up factor of 160 over the sequential CPU implementation.**

I. Introduction

Non-photorealistic rendering is a technique which produce an image resembling artistic representation, such as oil-painting, pencil drawing, pointillism, and mosaic. In this work, we focus on a non-photorealistic rendering for image generation of pointillism. Pointillism is a painting technique in which small dots of color are put on canvas in order to form an image represented by Georges Seurat who was a pioneer of the art style of pointillism in the neo-impressionism movement [1].

In the non-photorealistic rendering, there are several researches in which pointillism image generation have been tackled. They can be roughly classified into two major categories; *the Seurat's pointillism reproduction* and *the simple pattern pointillism*. In the Seurat's pointillism reproduction, pointillistic images are generated in order that the generated images resemble Seurat's style. The characteristics of the Seurat's pointillism consists of 11-color limitation, halo effect in the boundaries, and utilization of complementary colors for highlighting, and small dots of color. Based on the characteristics, several image generation techniques have been proposed [2]–[5]. On the other hand, in the simple pattern pointillism, pointillistic images are generated by putting simple patterns such as dots [6]–[8], circles [9], [10], ellipses [11], and arbitrary shapes [12] to canvas.

The main contribution of this paper is to propose a method for generating a pointillistic image which reproduces an input image by pasting colored squares to canvas. Figure 1 shows an example of the proposed pointillism method. Given an input image, the proposed method generates a pointillism image by pasting rotated square patterns one by one. Unlike the existing methods of the simple pattern pointillism in which patterns are arranged without overlap [6]–[8], [12], in the pointillism considered in this article, paste patterns can be put onto other patterns. To obtain high quality pointillistic images, we use the idea based on *the human visual system*. This idea has been utilized in the digital halftoning [13]–[15] and ASCII art generation [16]. In this idea, the goodness of a generated image is defined as the similarity between its projected image onto human eyes and the original image. The projected image is computed using a Gaussian filter that approximates the characteristic of the human visual system. The total error of the generated pointillistic image is defined as the sum of the difference of the intensity levels over all pixels between the original image and the projected image. Therefore, in this work, to obtain high quality pointillistic images, we try to generate a pointillistic image such that the total error is minimized. On the other hand, several researches consider features in the original image such as edges and directions of gradation [7], [8], [11]. However, without such features, the proposed pointillistic images generated by our proposed method can well represent such features that are larger than utilized patterns. In this work, based on the above idea, we propose two algorithms of the square pointillism image generation: *the serial pasting algorithm* and *the parallel pasting algorithm*. The serial pasting algorithm is pasting square patterns one by one, and the parallel pasting algorithm is putting multiple square patterns at once. Figure 1(b) shows an example of our square pointillism image generation of Lena [17] (Figure 1(a)) and its projected image is shown in Figure 2. In this example, 9786 square patterns of size 11×11 are pasted to generate the pointillistic image. By expanding the generated image, we can find the image consists of overlapped squares. However, edges and gradations that are larger than the square patterns are reproduced well.

The second contribution of this paper is to implement the above two algorithms on a GPU to accelerate the computation. GPUs (Graphics Processing Units), composed of a lot of processing units, can be used for general-purpose computation. Because GPUs have very high memory bandwidth, the performance of GPUs greatly depends on memory access. Therefore, many studies have been devoted to implement parallel algorithms using GPUs. Our experimental results show that the GPU implementation of the serial pasting algorithm can run up to 131.2 times faster than the sequential CPU implementation and 2.5 times faster than the parallel CPU implementation with 160 threads. On the other hand, regarding the parallel pasting

algorithm, the GPU implementation can run up to 160.7 times faster than the sequential CPU implementation and 7.1 times faster than the parallel CPU implementation with 160 threads.

This paper is organized as follows. Section II explains the proposed square pointillism image generation. In Section III, we propose two algorithms of the square pointillism image generation: the serial pasting algorithm and the parallel pasting algorithm. We then show GPU implementations to accelerate the computation in Section IV. Section V shows the resulting pointillistic images, and shows the computing time. Section VI concludes our work.

II. Proposed Square Pointillism Image Generation

A new pointillism technique by pasting squares is presented in this section. We first define the goodness of a generated pointillism image, that is, we introduce the error from an original image based on the human visual system. After that, we will show two algorithm of pointillism image generation in the next section.

First, a gray scale image is considered, and then we extend it to a color image. Consider an original image $A = (a_{i,j})$ of size $N \times N$, where $a_{i,j}$ denotes the intensity level at position (i, j) $(1 \leq i, j \leq N)$ taking a real number in the range $[0, 1]$. The pointillism image generation is to find an image $B = (b_{i,j})$ obtained by pasting a lot of squares, of the same size, that reproduces the original image A. The goodness of the output image B can be computed using the Gaussian filter that approximates the characteristic of the human visual system. Let $G = g_{p,q}$ denote a Gaussian filter, that is, a two-dimensional symmetric matrix of size $(2w + 1) \times (2w + 1)$, where each non-negative real number $g_{p,q}$ $(-w \leq p, q \leq w)$ is determined by a two-dimensional Gaussian distribution such that their sum is 1. In other words, $g_{p,q} = s \cdot e^{-\frac{p^2+q^2}{2\sigma^2}}$, where σ is a parameter of the Gaussian distribution and s is a fixed real number to satisfy $\sum_{-w \leq p,q \leq w} g_{p,q} = 1$. Let $R = (r_{i,j})$ denote the projected gray scale image of an image $B = (b_{i,j})$ obtained by applying the Gaussian filter as follows:

$$r_{i,j} = \sum_{-w \leq p,q \leq w} g_{p,q} b_{i+p,j+q} \quad (1 \leq i, j \leq N). \tag{1}$$

As $\sum_{-w \leq p,q \leq w} g_{p,q} = 1$ and $g_{p,q}$ is non-negative, each $r_{i,j}$ takes a real number in the range $[0, 1]$. Hence, the projected image R is a gray scale image. An image B is a good approximation of the original image A if the difference between A and R is small enough. The error $e_{i,j}$ at each pixel location (i, j) of image B is defined by

$$e_{i,j} = a_{i,j} - r_{i,j}, \tag{2}$$

and the total error is defined by

$$\mathrm{Error}(A, B) = \sum_{1 \leq i,j \leq N} |e_{i,j}|. \tag{3}$$

Because the Gaussian filter approximates the characteristic of the human visual system, the image B reproduces original image A if $\mathrm{Error}(A, B)$ is small enough.

We are now in a position to explain how we generate a pointillistic image. A pointillistic image generated in this work is obtained by putting fixed-size squares, each of which has a uniform color, of the same size $(2t + 1) \times (2t + 1)$. Squares are pasted one by one and they are allowed to be rotated and overlap other squares. Let P denote a set of square patterns and each element $p_{u,v}$ $(1 \leq u \leq N_L, 1 \leq v \leq N_R)$ in P denotes a square pattern, where N_L is the number of colors of square patterns and N_R is the number of the variation of rotation. Figure 3 depicts an example of the square patterns P.

To explain which and where square patterns are pasted, we introduce an improvement value I of the error by pasting a square pattern:

$$I(A, B, p, i, j) = \mathrm{Error}(A, B) - \mathrm{Error}(A, B'), \tag{4}$$

where B' is a canvas image when a square pattern p is pasted at (i, j) to B. Using this, we paste square patterns to B one by one. We put a square pattern that maximize the improvement I for all possible patterns to B. In other words, for each position (i, j) $(1 \leq i, j \leq N)$, we select a pattern $q_{i,j}$ whose value I is the maximum such that

$$q_{i,j} = \arg \max_{p_{s,t} \in P} I(A, B, p_{s,t}, i, j).$$

From $q_{i,j}$'s, we find the most improved pattern q_{best} whose improvement value is the maximum. After that, the pattern q_{best} is pasted to B. This procedure that the most improved pattern is put to B is repeated until no more improvement is possible.

In pointillism, dots are put on blank canvas whose color is generally white or black [11]. In this work, such background is completely covered with square patterns in generated pointillistic images. This is because in our experience, if the background can be peeked through the gap of square patterns, such area looks like noise since it is conspicuous. Therefore, in the following algorithms of pointillism image generation, we first cover background pixels with square patterns. We introduce the contribution ratio of covering background pixels when pattern p is put to B at (i, j), expressed as:

$$C(B, p, i, j) = \frac{\text{the number of covered background pixels}}{\text{the number of pixels of } p}.$$

Note that if a pattern is put on background pixels, that is, every pixel of the pattern covers a background pixel, $C(B, p, i, j) = 1$. On the other hand, if a pattern is put on non-background pixels, $C(B, p, i, j) = 0$. Hence, the goal that background is completely covered is to select a pattern such that $C(B, p, i, j) > 0$. Also, when the value $C(B, p, i, j)$ is larger, more pixels are covered by the pattern. By extending Eq. (4), we define the improvement I_{cover} to consider the background pixels covering as follows:

$$I_{\mathrm{cover}}(A, B, p, i, j) = (C(B, p, i, j), I(A, B, p, i, j)). \tag{5}$$

In other words, the improvement is a pair of 'the contribution ratio of covering background pixels' and 'the improvement of the total error' when pattern p is put to B at

(a) input image (Lena) (b) generated pointillism image

Fig. 1. An example of our pointillism image generation

Fig. 2. Projected image of Figure 1(b)

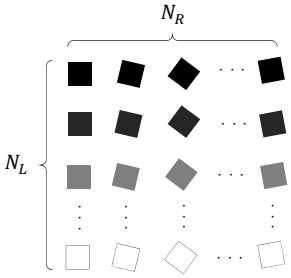

Fig. 3. Set of square patterns P

(i,j). We assume that the comparison of any two values of $I_{\text{cover}}(A,B,p,i,j)$ are based on the lexicographical order, that is, $I_{\text{cover}}(A,B,p,i,j) > I_{\text{cover}}(A,B,p',i,j)$ if and only if

- $C(B,p,i,j) > C(B,p',i,j)$ or,
- $C(B,p,i,j) = C(B,p',i,j)$ and $I(A,B,p,i,j) > I(A,B,p',i,j)$.

In the following, to cover background pixels, we use I_{cover} instead of I. If appearance of background pixels in generated images is allowed, I is used. Also, if background pixels in the Gaussian filter application are included, the values of background pixels affect the total error. Therefore, until all background pixels are covered, to exclude background pixels in the Gaussian filter application when I_{cover} is computed, instead of Eq. (2), we use

$$e_{i,j} = a_{i,j} - b_{i,j}.$$

Now, we extend the error computation for gray scale images to color images. In this work, we consider RGB colors whose value is specified with three real numbers in the range $[0,1]$ that represent red, green, and blue, respectively. For color images, projected image R and the error in Eq. (2) are computed for each color. Namely, for each color, Gaussian filter is pasted and the error is computed. Let $e^R_{i,j}$, $e^G_{i,j}$, and $e^B_{i,j}$ denote the errors of red, green, and blue at each pixel location (i,j), respectively. Eq. (3) is extended to the sum of each color value as follows;

$$\text{Error}(A,B) = \sum_{1 \le i,j \le N} (|e^R_{i,j}| + |e^G_{i,j}| + |e^B_{i,j}|). \qquad (6)$$

In the following explanation about our pointillism image generation method, the difference between gray scale and color is the above error computation. The other parts are common between them.

III. Algorithms for Pointillism Image Generation

The main purpose of this section is to propose two pointillistic image generation algorithms; *the serial pasting algorithm* and *the parallel pasting algorithm*. Briefly explaining, the serial pasting algorithm is pasting square patterns one by one, and the parallel pasting algorithm is putting multiple square patterns at once.

Consider an original image A and a canvas image B and let $W(i,j)$ be a window of size $(2t+1)\times(2t+1)$ whose center is at position (i,j) as illustrated in Figure 4. The window is the minimum upright square can include all square patterns in Figure 3. Also, Table I shows the size of window $W(i,j)$ for each size of square patterns. Because we use a Gaussian filter of size $(2w+1)\times(2w+1)$, pasting a square pattern affects the errors in a square region of size $(2t+2w+1)\times(2t+2w+1)$, which we refer to as the *affected region*. Figure 5 depicts a

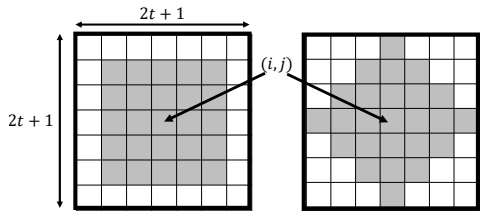

Fig. 4. Window $W(i,j)$

TABLE I
THE SIZE OF WINDOW $W(i,j)$

size of patterns	size of window
7×7	11×11 ($t = 5$)
9×9	13×13 ($t = 6$)
11×11	17×17 ($t = 8$)
13×13	19×19 ($t = 9$)
15×15	23×23 ($t = 11$)
17×17	25×25 ($t = 12$)
19×19	27×27 ($t = 13$)
21×21	31×31 ($t = 15$)
23×23	33×33 ($t = 16$)

window and the affected region. Note that the best pasting of a square pattern can be selected by computing the total errors of the affected region of size $(2t + 2w + 1) \times (2t + 2w + 1)$ because pasting the square pattern does not affect errors at pixels outside the affected region. The error of a fixed pixel in

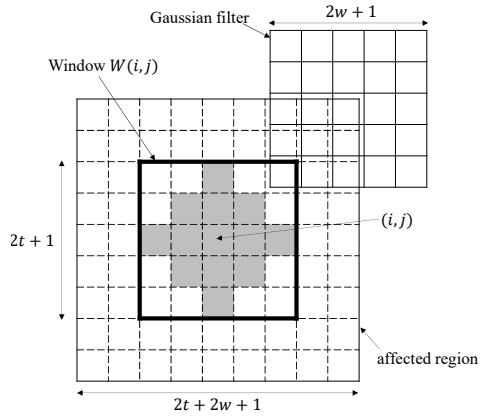

Fig. 5. Affected region

an affected region can be computed in $O((2w+1)^2) = O(w^2)$ time by evaluating Eqs. (1) and (2). Hence, all the errors in the affected region can be computed in $O(w^2(2t + 2w + 1)^2) = O(w^2(t^2 + w^2))$ time. After that, their sum can be computed in $O((2t + 2w + 1)^2) = O(t^2 + w^2)$ time. Thus, the total error in the affected region can be computed in $O(w^2(t^2 + w^2))$ time. Because we need to check all the possible $N_R N_L$ square patterns in P, the best square pattern can be obtained in $O(N_R N_L w^2(t^2 + w^2))$ time.

A. Serial pasting algorithm

In the following, we present a pointillistic image generation algorithm that are to paste square pattern one by one. Pasting square patterns and its error computation requires values from pixels outside of the image boundaries when patterns are put around borders. Therefore, in this work, the nearest border pixels in an original image A are extended as far as necessary to perform the LES around the boundaries [18]. More specifically, an original image A of size $N \times N$ is extended to the $(N + 2t + 2w) \times (N + 2t + 2w)$ image by copying the boundary pixel values. Also, a canvas image B is initialized such that every pixel is white, i.e., $b_{i,j} = 1$ $(1 \le i, j \le N)$ for gray scale images. After that, we find the best pattern $q_{i,j}$ for each point, and then the most improved pattern $q_{\rm best}$ from the patterns is selected and pasted to B. This procedure is repeated until no more improvement is possible. However, we do not have to perform an exhaustive search to find $q_{i,j}$'s for all the points once all $q_{i,j}$'s are obtained. If the projected image of the affected region, for the current window, does not change, then we can omit the exhaustive search using the previous best pattern as the current best pattern. Let $\mathcal{A}_{i,j}$ denote a set of positions in the affected region of the image in Figure 5 such that

$$\mathcal{A}_{i,j} = \{(i', j') | i - t - w \le i' \le i + t + w, j - t - w \le j' \le j + t + w\}.$$

Therefore, we compute the total error at pixel location (i, j) in Eq. (3) by evaluating the following formula:

$$\sum_{(i',j') \in \mathcal{A}_{i,j}} |e_{i',j'}|. \tag{7}$$

Similarly, for color images, the total error in Eq. (6) can be computed by

$$\sum_{(i',j') \in \mathcal{A}_{i,j}} (|e^R_{i',j'}| + |e^G_{i',j'}| + |e^B_{i',j'}|). \tag{8}$$

From the above, Algorithm 1 shows the serial pasting algorithm for pointillistic image generation.

B. Parallel pasting algorithm

In the above serial pasting algorithm, square patterns are pasted one by one. Therefore, it is difficult to implement the algorithm as parallel execution. Here, we show a pointillistic image generation algorithm that are to paste multiple square patterns at once. To paste multiple patterns, we split the input image A of size $N \times N$ into subimages of size $h \times h$. We partition the subimages into four groups such that

- Group 1: even columns and even rows;
- Group 2: odd columns and even rows;
- Group 3: even columns and odd rows; and
- Group 4: odd columns and odd rows.

Figure 6 illustrates the groups of subimages. Note that, if $h \ge 2t + 2w + 1$, then the Gaussian filter of two subimages in a group never affect each other, where the subimage is $h \times h$ and the affected region is $(2t + 2w + 1) \times (2t + 2w + 1)$. In other words, affected regions of a particular group do not overlap with each other. In the parallel pasting algorithm, we perform the serial pasting algorithm shown in the previous section for Group 1, Group 2, Group 3, and Group 4, in turn. Since there are $\frac{N}{h} \times \frac{N}{h}$ subimages in each group, at most $\frac{N}{h} \times \frac{N}{h}$ square patterns can be put for each group execution.

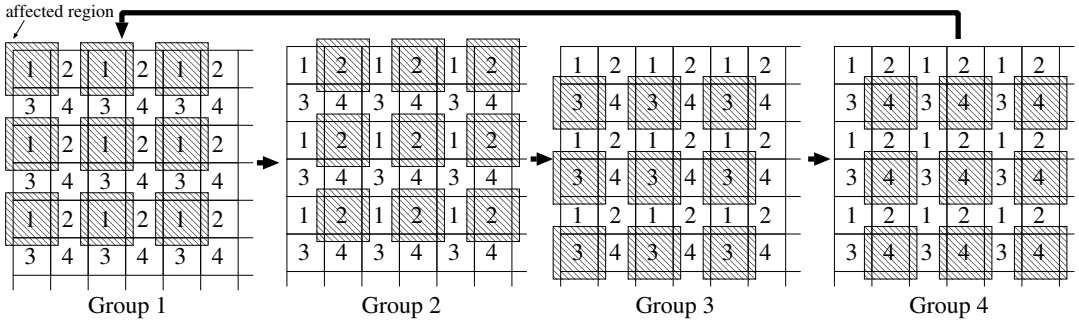

Fig. 6. Groups of subimages and parallel execution without race condition

Algorithm 1 Serial pasting algorithm

Input: Original image A
Output: Pointillistic image B

```
B_0 is initialized to a blank image
for i = 1 to N do
    for j = 1 to N do
        Find q_{i,j}.
    end for
end for
k ← 1
loop
    B_k ← B_{k−1}
    for i = 1 to N do
        for j = 1 to N do
            Update q_{i,j} if the projected image in the affected region
            of W(i, j) for B_k and B_{k−1} are not identical.
        end for
    end for
    Find the most improved pattern q_best from all q_{i,j}'s
    if the total error decreases when q_best is pasted to B_k then
        Paste q_best to B_k
    else
        return B_k
    end if
    k ← k + 1
end loop
```

IV. GPU Acceleration

This section shows an efficient GPU implementation of the pointillistic image generation to accelerate the computation.

We briefly explain CUDA architecture that we will use in the GPU implementation. NVIDIA provides a parallel computing architecture called *CUDA* on NVIDIA GPUs. CUDA uses two types of memories: *the global memory* and *the shared memory* [19]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-12GB, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-96KB. 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 multiprocessors such that all threads in a block are executed by the same streaming multiprocessor in parallel. All threads can access to the global memory. However, threads in a block can access to the shared memory of the streaming multiprocessor to which the block is allocated. Since blocks are arranged to multiple streaming multiprocessors, threads in different blocks cannot share data in the shared memories. CUDA C extends C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all blocks in the grid are allocated in streaming multiprocessors, and threads in each block are executed by processor cores in a single streaming processor.

We are now in position to explain how we implement the serial pasting algorithm. We assume that an input original image of size $N \times N$ is stored in the global memory in advance, the implementation writes the resulting pointillistic image to the global memory. In the serial pasting algorithm, we repeatedly perform a kernel that pastes a square pattern. In each kernel, we first find $q_{i,j}$ for each position (i, j) $(1 \leq i, j \leq N)$. Each CUDA block is responsible for finding $q_{i,j}$. In each CUDA block, multiple threads are used to compute improvement values I in Eq. (4) for each square pattern and then find the most improved pattern at (i, j) $q_{i,j}$ whose value I is the maximum is found. After that from $q_{i,j}$'s, the most improved pattern $q_{\rm best}$ is selected and pasted. After this kernel, the above is repeated for the affected area until no more improvement is possible. We note that in the first kernel, $q_{i,j}$'s for all positions need to be computed. However, the second and after kernels update $q_{i,j}$'s only for the affected region where a square pattern is pasted in the previous kernel.

In each CUDA block assigned to a pixel, we use three ideas efficiently to perform the computation, as follows.

Data caching using the shared memory: The input image A and the canvas image B stored in the global memory are frequently read during the computation. Therefore, to reduce the data access time to them, we cache the elements of A and B, that are necessary to perform the computation, to the shared memory.

Application of the Gaussian filter using addition: To obtain projected images R that are blurred by the Gaussian filter, we need to compute the convolution for each pixel value in Eq. (1). In this idea, we replace the convolution to addition of projected square patterns. More specifically, the sum of multiplications is replaced by the sum of additions. To do this, before the computation, we obtain the projected square patterns which are blurred by the Gaussian filter and store them

in the global memory. After that, the following operations are performed for each square pattern that is the same angle. First, a square pattern whose value is 0, that is, black square pattern, is pasted. After that, the blurred square pattern is added to B instead of the application of the Gaussian filter. From this computation, every square pattern which is the same angle, the convolution can be replaced by the addition except the computation of the first paste of the black square pattern.

Parallel sum-reduction with warp shuffle instructions: To obtain the total error in Eq. (7) and Eq. (8), the sum of the error values is computed. In our implementation, we use the parallel sum reduction technique using the warp shuffle instructions [20]. The warp shuffle instructions allow threads in a warp to perform the data communication between them without the shared memory [19]. In this technique, the parallel sum computation is efficiently performed using the warp shuffle instructions.

Next, we explain the GPU implementation of the parallel pasting algorithm. As shown in the previous section, in the parallel pasting algorithm, we perform the serial pasting algorithm for each subimage in one of the four groups illustrated in Figure 6 in turn. Since the computation for each subimage can be performed independently, in the GPU implementation of the parallel algorithm, we perform the same manner as the GPU implementation of the serial pasting algorithm for each subimage in parallel. This operation is repeated for each group until no more improvement is possible.

V. Experimental Results

In this section, we show the resulting pointillistic images and the computing time. We have used Lena [17] of size 512×512 in Figure 1(a). Also, we have utilized the square patterns consisting of 4096 colors and the patterns are rotated with angles of 0, 30, and 60 degrees. In total, the number of patterns in P is $4096 \times 3 = 12288$. We have evaluated the image generation by varying the size of patterns from 7×7 to 23×23. The Gaussian filter has been set with parameters $\sigma = 1.3$ and $w = 3$. Also, the size of subimage used in the parallel pasting algorithm is 23×23, that is, $h = 23$. We have evaluated the following implementations; *the sequential CPU implementation*, *the parallel CPU implementation*, and *the GPU implementation* for the serial pasting algorithm and the parallel pasting algorithm. In the sequential CPU implementation, a single thread performs each algorithm in serial. On the other hand, in the parallel CPU implementation, we use multiple threads to concurrently find $q_{i,j}$'s at lines 4 and 12 in Algorithm 1. Each thread computes Eq. (4) or Eq. (5) in serial. We have implemented it using OpenMP 3.1 [21]. We have used a multicore server with 4 Intel Xeon E7-8870V4 CPUs running in 3.0GHz and 1TB memory for the sequential and parallel CPU implementations. Each CPU has 20 physical cores each of which acts 2 logical cores by hyper-threading technology. In the parallel CPU implementation, we have deployed 160 threads that corresponds to the total number of logical cores. On the other hand, for the GPU implementation, we have used an NVIDIA TITAN X GPU with 3584 processing cores running in 1.531GHz and 12GB memory. The source code programs of the GPU implementations are compiled by nvcc version 8.0.60 with -O2 and -arch=sm_61 options.

Figure 7 shows a process of generating a pointillistic image of Figure 1(a) using the serial pasting algorithm. To generate the resulting pointillistic image in Figure 8(a), a total of 9786 square patterns have been pasted. Also, Figure 9 shows a graph of the change of the total error in Eq. (6). From the graph, first, the total error rapidly decreases and then the deceasing rate becomes small until 5706 square patterns are pasted. After that, the error slightly decreases again. This is because square patterns are put to cover the background pixels first even if the total error increases. Actually, by pasting the first 5878 square patterns, all the background pixels have been covered. Therefore, after all the background pixels are covered, the total error decreases again since the square patterns are pasted only to reduce the total error.

Figure 8(b) shows the resulting image in the parallel pasting algorithm. Compared with the image by the serial algorithm in Figure 8(a), by looking closely, both of them are a bit different, but the quality of them seems to be almost the same at a glance. In both images, the fine and intricate parts such as her hair cannot be represented since the size of patterns is not small enough to represent them. However, principal edges and gradations of colors are well-reproduced even though squares that are not covered by other squares are visible.

Figure 10 shows the resulting pointillistic images with square patterns of size 7×7 to 23×23. For each size of patterns, intricate area that is smaller than patterns cannot be represented. For example, her threads of hair are invisible for every size of patterns and her eyes can be seen for patterns of size 7×7 to 13×13. On the other hand, we can see the curves of large shapes including her hat and shoulder distinctly for every size. Table II shows the number of pasted square patterns and the *average error* of the resulting images. The average error is the total error per pixel that is computed by $\frac{\text{Error}(A,B)}{N^2}$. According to the table, when the size patterns is large, the number of pasted patterns is small. On the other hand, regarding the average errors, there is not much difference by size of pattern and algorithm.

TABLE II
The number of pasted square patterns and the average error

size of patterns	# pasted patterns		average error	
	serial	parallel	serial	parallel
7×7	17970	19570	18.194	18.186
9×9	11507	12746	20.277	20.313
11×11	9786	11045	22.104	22.060
13×13	6749	7385	23.537	23.730
15×15	5164	5604	25.105	25.051
17×17	5628	6296	26.582	26.567
19×19	3779	4218	27.900	27.865
21×21	4588	5027	29.123	29.211
23×23	3175	3717	30.604	30.496

Table III shows the computing time of pointillistic image generation for Lena. In the GPU implementation, the data communication time between the host PC and the GPU are included. According to the table, in the sequential CPU

Fig. 7. Snapshots of pasting patterns in the serial pasting algorithm with 11×11 square patterns

(a) the serial pasting algorithm (b) the parallel pasting algorithm

Fig. 8. The resulting pointillistic images with square patterns of size 11×11

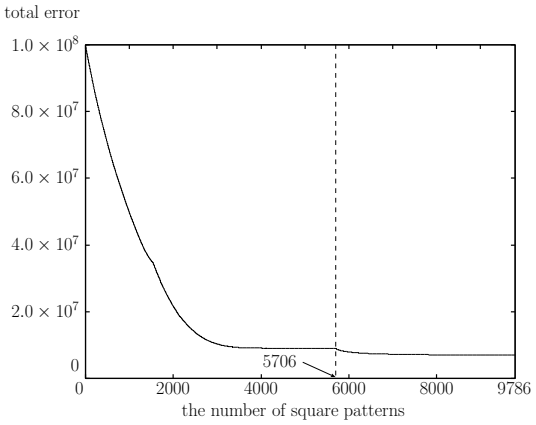

Fig. 9. The total error change transition of the serial pasting algorithm

implementation, the parallel pasting algorithm is faster than the serial pasting algorithm. This is because in the serial pasting algorithm, we need to find the most improved pattern from N^2 $q_{i,j}$'s to paste one square, where the input image is $N \times N$. On the other hand, in the parallel pasting algorithm, since one pattern is pasted in each subimage of size $h \times h$, we select a pattern only from h^2 $q_{i,j}$'s. Since $N = 512$ and $h = 23$ in this experiment, the parallel pasting algorithm is faster than the serial pasting algorithm.

In both of the parallel CPU implementation and the GPU implementation, we perform the computation of $q_{i,j}$'s in parallel. Furthermore, in the GPU implementation, each computation of $q_{i,j}$'s is concurrently executed by threads in a CUDA block. Therefore, the GPU implementation can run much faster than the parallel CPU implementation even if 160 threads are used on the multicore server. More specifically, the computing time of the GPU implementation reduced by a factor up to 131 and 160 over the sequential implementation for the serial and parallel pasting algorithms, respectively. On the other hand, compared with the parallel CPU implementation, the GPU implementation runs at most 2.5 times faster for the serial pasting algorithm and at most 7.1 times faster for the parallel pasting algorithm.

In the above, we have evaluated one image Lena of size 512×512. Therefore, we show the pointillistic image of a larger image. Figure 11 shows the resulting image of size 1920×1536 from the image database [22] by the parallel pasting algorithm. In the generation, 28072 square patterns of size 23×23 have been pasted and the average error per pixel is 36.27. Small characters in the top left image lose their shape, however, other parts of the image are well-reproduced. The computing time of the generation in the sequential CPU, parallel CPU and GPU implementations is 32391.9, 879.17, and 388.1 seconds, respectively.

VI. Conclusion

In this paper, we have proposed a new technique to generate a square pointillistic image. It is inspired by characteristic of the human visual system and the resulting pointillistic images well-reproduce the original images. To accelerate square pointillism image generation by our method, we have implemented it in the GPU. The experimental results show that the GPU implementation can achieve a speed-up factor up to 160 over the sequential CPU implementation.

References

[1] J. Hauptman, G. Seurat, and K. D. Buchberg, *Georges Seurat: the drawings*. The Museum of Modern Art, 2007.

[2] H.-L. Yang and C.-K. Yang, "A non-photorealistic rendering of Seurat's pointillism," in *Proc. of International Symposium on Visual Computing*, 2006, pp. 760–769.

[3] C.-K. Yang and H.-L. Yang, "Realization of Seurat's pointillism via non-photorealistic rendering," *The Visual Computer*, vol. 24, no. 5, pp. 303–322, 2008.

[4] J. Sugita and T. Takahashi, "A method for generating pointillism based on Seurat's color theory," *ITE Transactions on Media Technology and Applications*, vol. 1, no. 4, pp. 317–327, 2013.

[5] Y.-C. Wu, Y.-T. Tsai, W.-C. Lin, and W.-H. Li, "Generating pointillism paintings based on Seurat's color composition," *Computer Graphics Forum*, vol. 32, no. 4, pp. 153–162, 2013.

[6] A. Lu, C. J. Morris, J. Taylor, D. S. Ebert, C. Hansen, P. Rheingans, and M. Hartner, "Illustrative interactive stipple rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 2, pp. 127–138, 2003.

Fig. 10. The resulting pointillistic images with square patterns of size 7×7 to 23×23

TABLE III
THE COMPUTING TIME IN SECONDS OF POINTILLISTIC IMAGE GENERATION WITH SQUARE PATTERNS OF SIZE 7×7 TO 23×23

size of patterns	serial pasting algorithm					parallel pasting algorithm				
	CPU1 (1 thread)	CPU2 (160 threads)	GPU	$\frac{\text{CPU1}}{\text{GPU}}$	$\frac{\text{CPU2}}{\text{GPU}}$	CPU1 (1 thread)	CPU2 (160 threads)	GPU	$\frac{\text{CPU1}}{\text{GPU}}$	$\frac{\text{CPU2}}{\text{GPU}}$
7×7	1782.59	44.92	21.22	84.0	2.1	693.79	19.74	5.65	122.9	3.5
9×9	1782.13	40.30	17.90	99.6	2.3	748.74	20.93	5.50	136.1	3.8
11×11	2698.79	57.01	23.07	117.0	2.5	1400.96	39.29	9.12	153.7	4.3
13×13	2771.37	50.47	21.12	131.2	2.4	1292.69	35.79	8.04	160.7	4.5
15×15	3777.95	63.17	29.99	126.0	2.1	1821.22	85.70	12.06	151.0	7.1
17×17	4130.66	70.50	36.67	112.6	1.9	2868.84	131.62	21.38	134.2	6.2
19×19	4309.50	73.23	45.38	95.0	1.6	2294.44	93.64	23.27	98.6	4.0
21×21	6378.69	98.74	74.76	85.3	1.3	4687.03	147.49	56.28	83.3	2.6
23×23	6453.37	95.08	71.58	90.2	1.3	4107.68	124.97	43.18	95.1	2.9

Fig. 11. The pointillistic image of size 1920×1536 using square patterns of size 23×23

[7] D. Kim, M. Son, Y. Lee, H. Kang, and S. Lee, "Feature-guided image stippling," *Computer Graphics Forum*, vol. 27, no. 4, pp. 1209–1216, 2008.

[8] T. Houit and F. Nielsen, "Video stippling," in *Advanced Concepts for Intelligent Vision Systems: 13th International Conference*, 2011, pp. 384–395.

[9] N. Krüger and F. Wörgötter, "Symbolic pointillism: Computer art motivated by human perception," in *Proc. of Artificial Intelligence and Creativity in Arts and Science*, 2003, pp. 36–40.

[10] R. Bosch and A. Herman, "Pointillism via linear programming," *The UMAP Journal*, pp. 405–412, 2005.

[11] D. Chi, "A natural image pointillism with controlled ellipse dots," *Advances in Multimedia*, vol. 2014, pp. 1–16, 2014, art. ID 567846.

[12] Y.-S. Choi, B.-K. Koo, and J.-H. Lee, "Template based image mosaics," in *Proc. of 6th International Conference on Entertainment Computing*, 2007, pp. 475–478.

[13] M. Analoui and J. Allebach, "Model-based halftoning by direct binary search," in *Proc. SPIE/IS&T Symposium on Electronic Imaging Science and Technology*, vol. 1666, San Jose, CA, USA, 1992, pp. 96–108.

[14] H. Kouge, T. Honda, T. Fujita, Y. Ito, K. Nakano, and J. L. Bordim, "Accelerating digital halftoning using the local exhaustive search on the GPU," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 2, p. e3781, 2017.

[15] Y. Ito and K. Nakano, "A new FM screening method to generate cluster-dot binary images using the local exhaustive search with FPGA acceleration," *International Journal on Foundations of Computer Science*, vol. 19, no. 6, pp. 1373–1386, 2008.

[16] Y. Takeuchi, K. Nakano, D. Takafuji, and Y. Ito, "A character art generator using the local exhaustive search, with GPU acceleration," *International Journal of Parallel Emergent and Distributed Systems*, vol. 31, no. 1, pp. 47–63, 2016.

[17] "The USC-SIPI image database," http://sipi.usc.edu/database/.

[18] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 3rd ed. Prentice-Hall, Inc., 2006.

[19] *CUDA C Programming Guide Version 7.0*, NVIDIA Corporation, 2015.

[20] J. Luitjens. (2014) Faster parallel reductions on kepler. [Online]. Available: https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

[21] "OpenMP," http://www.openmp.org/.

[22] "JIS X 9204-2004 standard color image data."