IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 27, 2019, accepted May 24, 2019, date of publication June 24, 2019, date of current version July 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2924658

Parallel Fast Pencil Drawing Generation Algorithm Based on GPU

JIYAN QIU[1], BIN LIU[1,2,3], JINRONG HE[4], CHAOYANG LIU[1], AND YUANCHENG LI[5]

[1]College of Information Engineering, Northwest A&F University, Yangling 712100, China
[2]Key Laboratory of Agricultural Internet of Things, Ministry of Agriculture and Rural Affairs, Northwest A&F University, Yangling 712100, China
[3]Shaanxi Key Laboratory of Agricultural Information Perception and Intelligent Service, Northwest A&F University, Yangling 712100, China
[4]College of Mathematics and Computer Science, Yan'an University, Yan'an 712100, China
[5]School of Computer Science and Technology, Xi'an University of Science and Technology

Corresponding author: Bin Liu (liubin0929@nwsuaf.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61602388, in part by the Natural Science Basic Research Plan in Shaanxi Province of China under Grant 2017JM6059, in part by the Fundamental Research Funds for the Central Universities under Grant 2452019064, in part by the Postdoctoral Science Foundation of Shaanxi Province of China under Grant 2016BSHEDZZ121, in part by the China Postdoctoral Science Foundation under Grant 2017M613216 and Grant 2018M633585, in part by the Natural Science Basic Research Plan in Shaanxi Province of China under Grant 2018JQ6060, in part by the Key Program of the National Natural Science Foundation of China under Grant 61834005, in part by the Fundamental Research Funds for the Central Universities under Grant 2452016081, in part by the Doctoral Starting up Foundation of Yan'an University under Grant YDBK2019-06, and in part by the Innovation and Entrepreneurship Training Program of Northwest A&F University of China under Grant 2201810712307.

ABSTRACT With the development of image processing technology, pencil drawing has been widely used in video games and mobile phone applications. However, the existing pencil drawing algorithms require a large amount of time to convert a real picture into a pencil drawing; hence, it is difficult to apply them to real-time systems. This paper proposes a parallel fast pencil drawing generation algorithm based on the graphics processing unit (GPU) to accelerate the real-time rendering process of sketch painting. The parallelism of the pencil drawing generation algorithm is identified via a theoretical analysis at first. Then, sub-algorithms of the sequential algorithm are designed in parallel using the compute unified device architecture (CUDA) programming model and executed via thread-level parallel techniques. Furthermore, an optimal cache pattern of data that reduce the access time of the most frequently used data is structured using shared memory and constant memory. Finally, task-level parallelism is achieved by the CUDA stream technology, which overlaps independent sub-tasks for further acceleration. On the CUDA platform, the experimental results demonstrate that the proposed parallel algorithm can achieve a significant increase in speedup. The proposed algorithm achieves a performance improvement of 448.59 times compared with the sequential algorithm, on 2560×1920-resolution images, and maintains a high degree of similarity with the real pencil paintings. Hence, the proposed algorithm is suitable for real-time pencil drawing rendering and has promising application prospects in non-photorealistic rendering.

INDEX TERMS Non-photorealistic rendering, pencil drawing, parallel algorithm, GPU platform, convolution operation, CUDA.

I. INTRODUCTION

Pencil drawing is a prevalent art form, which is widely used in art, architecture, games, animation and other fields. With the development of digital image technology, people began to use software, such as Adobe Photoshop, to create paintings. Pencil drawing generation algorithms can convert a picture that was captured by a camera into a sketch automatically, without any human involvement. Most of those algorithms [1]–[5] are based on line integral convolution (LIC) [6]. Based on various criteria, digital image processing (DIP) algorithms [7]–[11] achieve a better performance. However, both types of algorithms require a long time to provide real-time rendering and the results of the LIC-based algorithms lack texture details.

Using compute unified device architecture (CUDA), which is a general parallel computing platform that was created by NVIDIA, it is possible to solve this problem

The associate editor coordinating the review of this manuscript and approving it for publication was Ligang He.

efficiently [12]–[18]. In such data-parallel scenarios, graphics processing unit (GPU) has been applied with satisfactory results over the past decade, mainly due to the high parallel processing performance offered by the single instruction multiple thread (SIMT) model [19]–[21]. Both LIC-based [22], [23] and DIP-based [24]–[27] algorithms were transplanted onto the GPU platform to reach a large acceleration. However, the results that were rendered by the LIC-based and DIP-based algorithms differed from the real pencil paintings.

In recent years, Lu C et al. proposed an algorithm based on a combination of sketch and tone, which achieves an excellent performance on various types of art [28]. However, because it is a sequential algorithm, it requires a substantial amount of time to produce a pencil drawing. Consequently, the original algorithm is unable to render a pencil drawing in real time. In this paper, this sequential algorithm was transplanted onto the GPU parallel platform successfully. The main contributions of this paper are as follows:

1) The sequential algorithm is designed and implemented in parallel on the CUDA platform. The speedup is up to 448.59 times on 2560 × 1920-resolution images, and for image for each resolution, less than 0.5 milliseconds are required to automatically produce a pencil drawing.
2) Each sub-algorithm is designed and implemented in parallel using the CUDA programming model and executed concurrently via thread-level parallelism technology. The data are maximum-blocked for execution in parallel by SIMT. High-frequency data are placed in caches such as shared memory and constant memory to minimize the memory access time.
3) Multiple independent tasks are concurrently performed via the CUDA Stream technology. The algorithm is divided into multiple sub-tasks and these sub-tasks are scheduled and executed in parallel by overlapping independent parts of each sub-task, such as data transmission and calculation.

The pencil drawing generation algorithm is implemented by the CUDA programming model on a high-performance server with two NVIDIA Tesla P100 GPUs. The parallel algorithm makes full use of the parallelism of the pencil drawing generation algorithm and accelerates the drawing process. The experimental results demonstrate that the parallel pencil drawing generation algorithm achieves an average acceleration of 448.59 times on 2560×1920-resolution images and the grayscale histogram is very close to the true pencil drawing.

The remainder of this paper is organized as follows: In Section II, the pencil drawing generation algorithm, the CUDA platform and CUDA Stream are described briefly. In Section III, the parallel design and implementation of the parallel fast pencil drawing algorithm are described. Section IV analyzes experimental results. In Section V, related work is introduced and summarized. Finally, the conclusions of this paper are presented in Section VI.

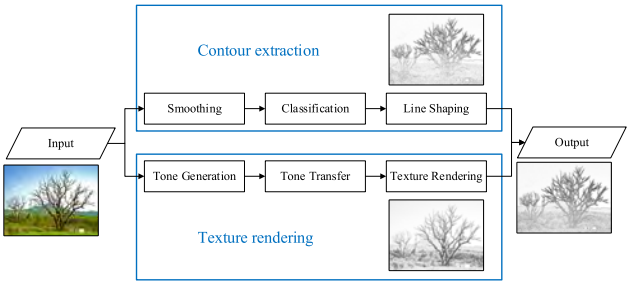


FIGURE 1. Image processing in a sketch algorithm.

II. PRELIMINARIES

A. PENCIL DRAWING GENERATION ALGORITHM

Non-photorealistic rendering (NPR) converts a traditional computer image to a variety of expressive styles for art. A pencil drawing generation algorithm is a type of NPR algorithm and it transforms a digital image into a sketch. The algorithm is improved based on the combination of sketch and tone [28]. As shown in Figure 1, it mainly includes two functions: contour extraction and texture rendering. Contour extraction is aimed to draw the edge of the object. Convolution operations, as the main step of contour extraction, are used to do this according to the gradients and lines in 8 directions. Texture rendering is used to render the real image into the pencil-drawn textures via the grayscale transform. Finally, the two parts of the processing results are combined to obtain the final pencil drawing.

1) CONTOUR EXTRACTION

First, the original image is denoised via median filtering. Then, 8 response maps are calculated by performing convolution operations on the gradients of the denoised images for classification. The response maps can be calculated via Eq. (1).

$$G_i = \mathcal{L}_i * M \tag{1}$$

where $i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$ denotes eight directions with 22.5-degree increments. M is the gradient of the denoised image, G_i for $i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$ denotes the 8 response maps after the grouping, and $\mathcal{L}_i$ for $i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$ denotes 8 convolution kernels whose spans are set to 1/30 of the image height or width to yield more accurate results. The convolution kernels are 8 edge detection templates, each of which is 22.5 degrees apart in direction, as shown in Figure 2 using a size of 5 × 5 as an example.

After that, pixels will be classified by selecting the maximum value of G_i. The pixels that have the maximum response value in direction i are classified into a sub-image, which is expressed by Eq. (2).

$$C_i(p) = \begin{cases} M(p) & \text{if } \max \arg_i\{G_i(p)\} = i \\ 0 & otherwise \end{cases} \tag{2}$$

Via this approach, the pixels in the same direction can be correctly classified. In Eq. (2), p represents a pixel in the image, $M(p)$ denotes value of the gradient of the denoised

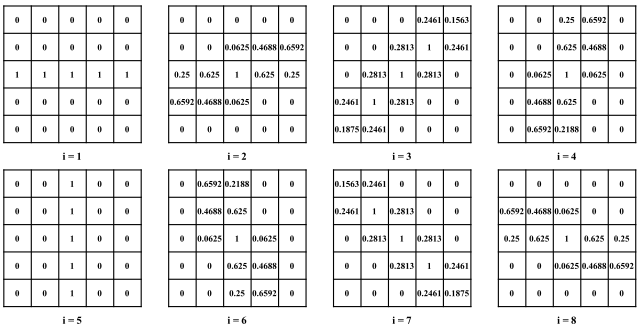


FIGURE 2. 8 edge detection templates.

image at p and $C_i(p)$ for $i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$ denotes the sub-image represented by the classification in eight directions.

Since each group in $C_i(p)$ has the same orientation, a straight line is used instead of discrete pixels. The convolution operation can gather the pixels along the direction and the steps of line drawing are completed again using the same kernel, namely, $\mathcal{L}_i$, for a second convolution.

$$S_i' = \mathcal{L}_i * C_i \tag{3}$$

Finally, all the $S_i' i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$ matrices are added together to calculate the combined result of and the final pencil outline is obtained, as expressed in Eq. (4).

$$S = \sum_{i=1}^{8} S_i' \tag{4}$$

2) TEXTURE RENDERING

In texture rendering, as illustrated in Figure 3, a histogram of the original image and the target histogram, which is expressed in Eq. (5), are calculated. Then, the original histogram is converted to the target histogram via histogram matching and the actual pencil texture is added to the image.

To express the tone map distribution of pencil drawings, Lu Cewu et al. proposed a parametric model for representing the tone distribution of pencil drawings [28]. Eq. (5) describes this parametric model of target histogram J, in which v is the color grayscale value, which ranges from 0 to 1 after normalization. Here, $p_i(v)$ for $i \in \{1, 2, 3\}$ denotes the probability that a pixel in a pencil drawing has value v and represents the three tonal layers of the pencil drawing, as expressed in Eq. (6) to Eq. (8); ω_i represents the weight of the corresponding tonal layer; and z is the normalization factor such that $\int_0^1 p(v)dv = 1$.

$$J = \frac{1}{z} \sum_{i=1}^{3} \omega_i p_i(v) S \tag{5}$$

$$p_1(v) = \begin{cases} \frac{1}{\sigma_b} \exp(-\frac{1-v}{\sigma_b}) & v \le 1 \\ 0 & otherwise \end{cases} \tag{6}$$

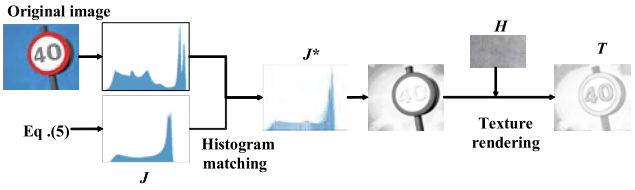


FIGURE 3. Procedure of texture rendering.

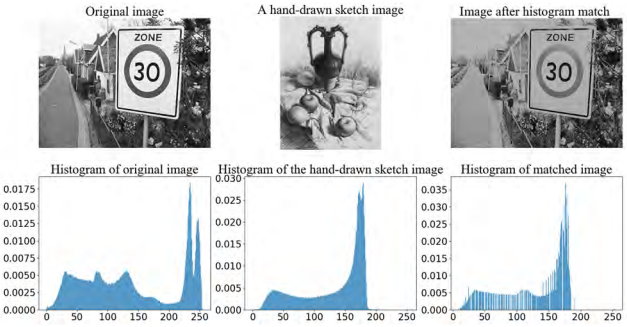


FIGURE 4. Histogram matching of pencil drawings.

$$p_2(v) = \begin{cases} \frac{1}{u_b - u_a} & u_a \le v \le u_b \\ 0 & otherwise \end{cases} \tag{7}$$

$$p_3(v) = \frac{1}{\sqrt{2\pi}\sigma_d} \exp(-\frac{(v-\mu_d)^2}{2\sigma_d^2}) \tag{8}$$

The first layer is the bright tone layer, which is represented by the Laplacian distribution with standard deviation σ_b. This distribution converges when v is close to 1. Then, the middle-tonal layer is represented by a descriptive model expressed as a uniform distribution from u_a to u_b, which indicates the uniform brightness of the image background. Finally, the dark tone layer is represented by a descriptive model expressed as a normal distribution with the mean μ_d and standard deviation σ_d.

Figure 4 illustrates the process of pencil drawing histogram matching. This process requires 2 input images: an original image and a pencil drawing. By changing the gray values of the pixels in the original image, the gray distribution of the original image is made consistent with the distribution of the pencil drawing. The matching target histogram is calculated via Eq. (5), instead of using a real pencil drawing image.

The method of pencil texture rendering is to determine the parameter β by solving a linear equation of the tone map, as expressed in Eq. (9).

$$\beta = 1 - J^* \tag{9}$$

Here, J^* is the result of matching histogram J. The final pencil texture map, which is denoted as T, is computed via an exponentiation operation, as expressed in Eq. (10), in which H is an actual pencil texture image.

$$T = H^{\beta} \tag{10}$$

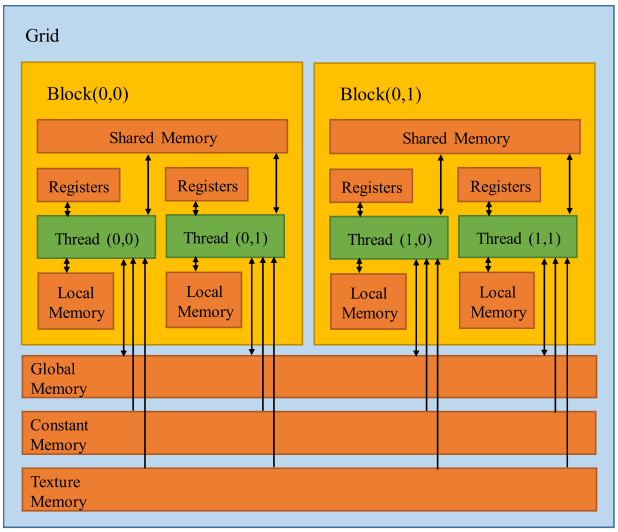


FIGURE 5. CUDA structure.

Finally, the pencil outline and the pencil texture are combined to obtain the pencil drawing result and the calculation method is expressed as Eq. (11).

$$R = S \times T \tag{11}$$

B. COMPUTE UNIFIED DEVICE ARCHITECTURE

As high-performance computing has evolved, CUDA, with its excellent performance and outstanding ecology, has attracted the attention of many developers and researchers. The main advantage of CUDA in image processing is that it commands many thread-level parallel tasks through the single instruction multiple data (SIMD) parallel computer architecture. By binding a pixel to a thread to complete pixel-level parallelism, it does not increase the geometrical growth of the combining time as the image size increases.

Figure 5 illustrates a typical CUDA in one GPU. Threads are the smallest unit of GPU execution and can perform parallel tasks. A warp is a collection of 32 threads that perform as a SIMD operation and a block is a collection of multiple warps. Only threads in the same block can communicate directly, e.g., access the same shared memory or synchronize quickly. The grid consists of several blocks and the global memory, constant memory and texture memory, which can be accessed by every block and thread in it. These three types of memory in the grid differ in terms of access speed and application and must be used according to the characteristics of the algorithms. Warps are executed via the lockstep synchronous approach; hence, branches of threads will cause divergent execution paths, which will lead to an increase in the total number of instructions executed, which needs to be avoided. In the CUDA parallel computing framework, locking operations are implemented via atomic commands. An atomic command is an independent operation and there is no interference from other threads, which satisfies the requirements of the lock operation.

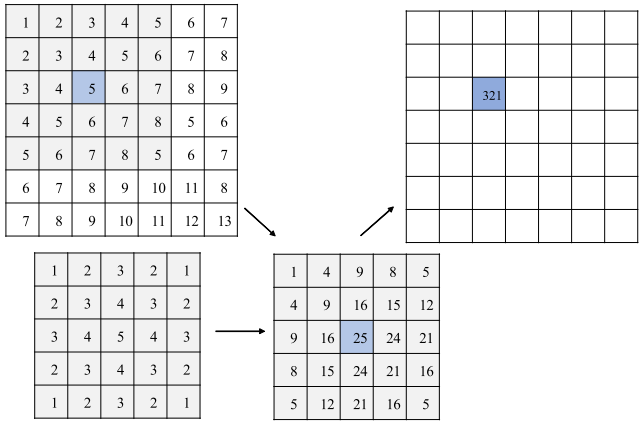


FIGURE 6. Calculation process of the two-dimensional convolution operation.

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the host refers to the CPU and its memory, while the device refers to the GPU and its memory. The code run on the host can manage memory on both the host and device and launch GPU kernels, which are functions executed on the device and by many GPU threads in parallel. The CUDA Stream processor (SP) is a mechanism for implementing the concurrent execution of the CUDA task level. It is similar to the multi-task scheduling of the CPU; however, it lacks the ability to adapt to new conditions or scenarios. The concurrent execution of tasks can be achieved due to the CUDA streaming mechanism. Therefore, the speed of the GPU program can be further accelerated by computing the data overlap in preparation for the data of the next computation, rather than performing the computations separately.

III. PARALLEL DESIGN AND IMPLEMENTED OF PENCIL GENERATING ALGORITHM

The pencil drawing algorithm can be divided into two functions: contour extraction and texture rendering. In this section, the parallelism of these functions is analyzed and the parallel design and implementation are presented using the CUDA programming model.

A. CONTOUR EXTRACTION

1) IDENTIFYING THE PARALLELISM

The time spent on 2D convolution grows geometrically as the size of the image increases for the sequential contour extraction function, and the most time-consuming part of contour extraction is the convolution operation. The performance of the function would be significantly improved if the convolution operation could be parallelized.

Figure 6 illustrates the calculation process of the 2D- convolution operation. The kernel is placed on the image, each pixel value that overlaps with the kernel is multiplied by the corresponding kernel value, and the products are summed. The result is placed in a new image that corresponds to the kernel center. The kernel moves one pixel and repeats this

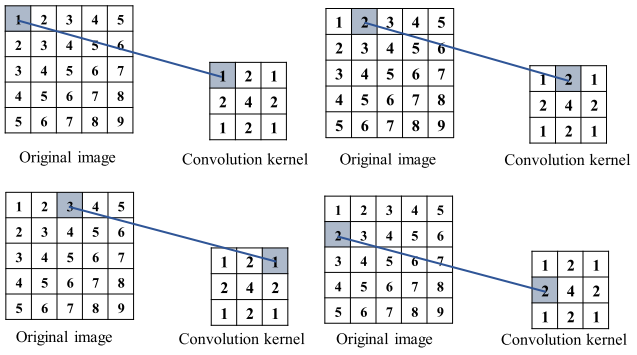


FIGURE 7. Serial algorithm processing mechanism of the convolution operation.

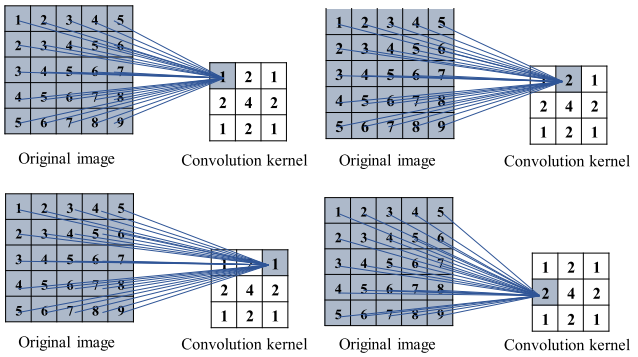


FIGURE 8. Parallel algorithm processing mechanism of the convolution operation.

process until all possible locations in the image have been visited. For convenience of description, two-dimensional data of size 7×7 and a convolution kernel of size 5×5 are considered. According to Figure 6, the convolution operation requires 25 multiplication operations for each element in the image and the total number of multiplication operations to be performed is $49 \times 25 = 1225$. In a 1024×1024 image, the number of calculations will reach $1024 \times 1024 \times 25 = 26214400$. It would be very time consuming to use a serial algorithm to implement this process.

However, in a GPU parallel architecture, it will be possible to complete this operation in an instruction cycle because the data involved in the convolution operation are irrelevant.

Figure 7 and Figure 8 illustrate the disparity in computing power between the serial and parallel platforms. The original image is represented by the squares on the left, while the convolution kernel is represented by the squares on the right. Two-dimensional data of size 5×5 and a convolution kernel of size 3×3 are considered in these figures. One pixel or one convolution kernel unit is represented by a number in the square. Each line that connects the left picture and the right picture represents a calculation performed by a thread. Within one fetch-and-execute cycle, Figure 7 and Figure 8 illustrate the differences in the processing methods between the serial and parallel processing mechanisms. If only multiplication operations are considered, the serial algorithm requires $5 \times 5 \times 3 \times 3 = 225$ steps to complete the task, while the parallel algorithm requires only $3 \times 3 = 9$ steps. The time-cost is reduced from $M \times N \times J \times K$ to $M \times N$ for an $M \times N$ image and a $J \times K$ convolution kernel. The number of computations that can be performed at the same time depends on the number of threads.

According to the calculation process of the convolution operation, the following parallelism can be obtained. The convolution operation is independent of other calculations; consequently, the calculation results between the elements have no dependencies and can be calculating in parallel, which is consistent with the characteristics of large-scale parallel computing. Each step of the convolution operation involves a neighborhood element; as a result, data are shared between the input elements. Therefore, the input image can be chunked using shared memory. This shared-memory-based blocking strategy can increase the memory bandwidth and reduce the global memory access frequency. Constant memory is visible to all threads and has lower access latency than global memory; hence, using constant memory to store convolution kernel data can enhance the memory access efficiency.

2) DESIGN AND IMPLEMENTATION

First, a convolution algorithm for calculating the response map and drawing the lines is designed. Since the output of a convolution operation involves only one pixel, assigning a thread to each pixel would be a reasonable strategy. However, the convolution operation requires the eight pixels around each pixel as input, which inevitably poses the problem of boundary processing. In addition, read conflicts will occur, which will directly affect the data reading speed of the sub-algorithm.

The image is divided into blocks such that the number of pixels in each picture is the same as the number of threads in a thread block. The objective is to allocate shared memory for each thread block such that threads must only access the high-speed shared memory instead of the low-speed global memory.

However, pixels at thread block boundaries cannot retrieve data from the shared memory. To overcome this problem, this paper considers a circle of ''halo elements'' when initializing the shared memory. As shown in Figure 9, ''halo elements'' will fill elements outside the boundary to prevent going out of bounds when the array is visited, which will be performed by the thread associated with each pixel at the edge of the image.

Since the shared memory size is related to the image size and the size of the convolution kernel, the dynamic shared memory function provided by CUDA is used to determine the size of the shared memory at runtime. After the data have been chunked, thread synchronization is required to ensure that the partition loading process has been completed so that the weighted summation calculation can be performed safely.

Sub-algorithm 1 details the CUDA kernel implemented for the convolution. The input of the algorithm is an original image, a convolution kernel, and the image size. The thread identifier of each thread and memory is initialized.

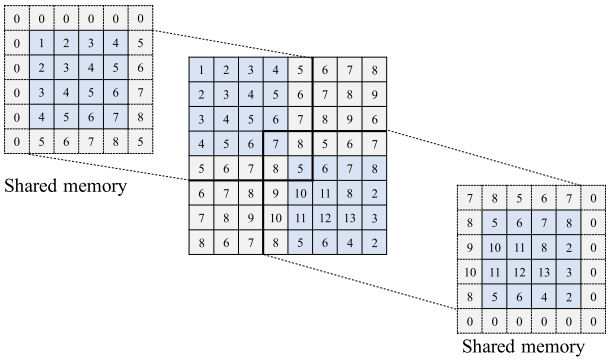


FIGURE 9. Halo elements.

Sub-Algorithm 1 Parallel Algorithm for the Two-Dimensional Convolution Operation

Input: I: Input image; M: Convolution kernel; *width*: Width of the image;
height: Height of the image; *ks*: Kernel size; *bs*: Block size;
Output: R: Result image;

1 initialize *value* to 0;
2 compute thread indices: *idx* and *idy*;
3 compute indices in every block: *lidx* and *lidy*;
4 allocate shared memory cache of size $bk \times bk$;
5 $r \leftarrow ks/2$;
6 $cache_{lidx,lidy} \leftarrow I_{idx,idy}$;
7 **if** $threadIdx_x < r$ or $threadIdx_y < r$
8 fill border elements;
9 synchronize threads;
10 $value \leftarrow 0$;
11 **for** y **from** 0 to ks
12 **for** x **from** 0 to ks
13 $value \leftarrow value + cache_{lidx-r+x,lidy-r+y} \times M$x,y;
14 **end for**
15 **end for**
16 $R_{idx,idy} \leftarrow$ value;

After that, the thread checks whether the data are out of bounds or not. Lines 7-10 in the algorithm fill the ''halo elements''. The warps must be synchronized after completion. The threads each read one pixel in parallel from the original image according to their ids. Because the id of each thread is unique, the input data differ and are independent. After that, the threads will put the data into the shared memory in parallel and the threads at the edge of the block will fill the ''halo elements'' by performing extra readings from the original image. Lines 11-16 in Sub-algorithm 1 correspond to the loop in which 8 adjacent pixels are computed by each thread. All threads will convolve based on the data in the shared memory, after which each thread will return a value and write to the result image.

Then, image smoothing is implemented via median filtering. The parallel implementation method and data assessment of median filtering are approximately the same as the convolution operation, in which each point requires the values of the 9 pixels around it. The median filter will encounter the same problem when calculating data on the edge of a block. Consequently, the median filter also requires ''halo elements'' for filling the boundary. The general method of finding the median is implemented via a sorting algorithm. Due to the small number of elements that are sorted, a simpler sorting approach is used here and determining the median does not require sorting all the elements, namely, only half of the elements must be sorted. This technique theoretically reduces the computing time by nearly half. The pseudocode and analysis are skipped here.

Sub-Algorithm 2 Parallel Algorithm for Pixel Classification

Input: Gs: Response maps; M: Gradient image; *width*: Width of the image; *height*: Height of the image; t: Time of classification; *depth*: Number of response maps
Output: Cs: Result images;

1 compute thread indices: *idx* and *idy*;
2 compute offsets: $offset_x$ and $offset_y$;
3 **while** $idx < width$ **and** $idy < height$
4 compute the index of the maximum element in the 8 response maps;
5 **if** $index_{max} = t$
6 $C_t id_x, id_y \leftarrow M_{idx,idy}$;
7 **end if**
8 $idx \leftarrow idx + offset_x$
9 $idy \leftarrow idy + offset_y$
10 **end while**

Next, the pixel classification algorithm is designed in parallel. To compare the values of eight response maps, threads must process the same point in 8 images in parallel. First, the maximum ID of each position in the 8 response maps is calculated. Then, according to the maximum id, the values of the pixels of the denoised image will be written back to the classification result images in parallel.

Sub-algorithm 2 details the CUDA kernel implemented for the pixel classification. The inputs are the 8 response maps generated by Sub-algorithm 1, the gradient image generated by Sub-algorithm 3 and the classification time. Lines 1-2 initialize the index of the target pixel for processing. Then, lines 3-10 define the loop for finding the largest response value among 8 directions and recording the result.

The gradient calculation algorithm is also parallelized. This algorithm uses matrix operations to implement gradient calculations for images. The gradient calculation based on matrix subtraction must perform matrix subtraction in the X-direction and the Y-direction of the image and sum the absolute values of the subtraction results. In the matrix subtraction operation, not all elements are subtracted. In the subtraction operation along the X-axis or Y-axis direction, the first column or row element of the subtracted matrix and the last column element of the subtraction matrix do not participate in the operation.

Sub-Algorithm 3 Parallel Algorithm for Gradient Calculation

Input: I: Input image; *width*: Width of the image; *height*: Height of the image;
Output: M: Gradient;

1 initialize *value* to 0;
2 compute thread indices: *idx* and *idy*;
3 **if** $idx+1 < width$
4 $gradx \leftarrow I_{idx,idy} - I_{idx+1,idy}$;
5 **end if**
6 **if** $idy+1 < height$
7 $grady \leftarrow I_{idx,idy} - I_{idx,idy+1}$;
8 **end if**
9 $M_{idx,idy} \leftarrow |grad_x| + |grad_y|$

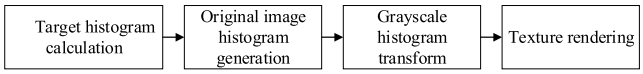


FIGURE 10. Flow chart of texture rendering.

Sub-algorithm 3 details the CUDA kernel implemented for the gradient calculation. The partial derivatives in the X- and Y-directions are calculated and the gradient is calculated from the result of partial derivatives. Here, the partial derivatives are represented by the differences between a pixel and its adjacent pixels. The gradient is simplified as the sum of the absolute values of the differences. The indices of the target points are initialized to calculate the gradient and to write back after the parallel calculation has been completed.

B. TEXTURE RENDERING

1) IDENTIFYING THE PARALLELISM

As illustrated in Figure 10, texture rendering can be divided into the following steps and the parallelism of each step is analyzed.

The target histogram calculation aims at generating a histogram in accordance with a stroke based on the input parameters. In this algorithm, the output calculations are independent of one another and can be easily parallelized by calculating one corresponding grayscale value per thread.

Image histogram generation is used to calculate the frequency of each gray scale value. Generating a histogram of a two-dimensional image is simple. It only requires the preparation of an array of histograms and the traversal of every pixel of the image. According to the pixel value, the value at the corresponding position of the array of histograms can be selected.

In grayscale histogram matching, which is the main process of texture rendering, the original image is converted to the target histogram. In this process, the calculations of the elements are independent of one another. Histogram calculation requires access to all data of a two-dimensional image, where the traversal of each element is independent; therefore, thread-level parallelism can be achieved.

Sub-Algorithm 4 Parallel Algorithm for Parameter Model Generation

Input: None
Output: H: Target histogram

1 compute thread index: *idx*;
2 compute offset: *offset*;
3 **while** $idx < 256$
4 $value \leftarrow p_1(idx) + p_2(idx) + p_3(idx)$;
5 $H_{idx} \leftarrow value$;
6 **atomic operation**: $H_{256} \leftarrow H_{256} + value$;
7 $idx \leftarrow idx + offset$
8 **endwhile**

The processes of pencil texture rendering and image merging are mainly performed using matrix calculations; consequently, the parallelism of linear algebra operations can be easily exploited. Pencil texture rendering mainly uses matrix subtraction and matrix exponentiation operations. Both operations are point-to-point calculations. The calculation of each element is independent and does not require data from other pixels. The process of image merging mainly uses matrix multiplication, which has the same characteristics as the previous two operations. Therefore, texture rendering and image merging can be parallelized.

2) DESIGN AND IMPLEMENTATION

The pseudocode of target histogram calculation is presented in Sub-algorithm 4. Atomic operations are used to ensure that no conflicts occur when competing to write to memory. The accumulated result and the gray histogram array are successively stored in the same block of memory to avoid multiple discontinuous memory transfer overheads. This paper simplifies the operation process by combining three tonal functions into a single function. This process does not require any additional input; the gray level range of 0-255 is traversed and the corresponding function value is calculated.

The pseudocode of histogram calculation is presented in Sub-algorithm 5. First, the image is loaded to the shared memory for each block and each thread only counts the data in the shared memory. In this process, atomic operations are needed for counting, which guarantees that no write conflicts will occur. Then, the statistics in all blocks in parallel reduction are summarized. Because these two steps have strict dependencies, after the end of the previous step, the algorithm must use a thread synchronization fence to ensure that the operations in all thread blocks are completed.

Pixel mapping and histogram calculations are implemented in the opposite direction. They do not involve sharing data; hence, each thread must only traverse each pixel of the input image, find the value in the histogram and assign it to the specified position of the target image. Since the calculation methods of pixel mapping, texture rendering and final image merging are similar and closely related, the 3 steps are combined in the same kernel function. The pseudocode of this kernel function is presented in Sub-algorithm 6.

Sub-Algorithm 5 Parallel Algorithm for Histogram Calculation

Input: I: Image
Output: H: Histogram

1 allocate shared memory: T;
2 $T_{idx,idy} \leftarrow 0$;
3 synchronize threads;
4 compute thread indices: i_{dx} and i_{dy};
5 compute offsets: $offset_x$, $offset_y$;
6 **while** $idx < width\ and\ idy < height$
7 **atomic operation**: $TI_{idx,idy} \leftarrow TI_{idx,idy} + 1$;
8 $i_{dx} \leftarrow i_{dx} + offsetx$
9 $i_{dy} \leftarrow i_{dy} + offsety$
10 **endwhile**
11 **synchronize threads;**
12 **atomic operation:** $H_{idx,idy} \leftarrow H_{idx,idy} + T_{idx,idy}$;

Sub-Algorithm 6 Parallel Algorithm for Pixel Mapping, Texture Rendering and Image Merging

Input: I: Image; H: Histogram; S: Stroke image; P: Pencil image; J: Tone map
Output: H: Histogram R: *Final result*

1 compute thread indices: idx and idy;
2 $D_{idx,idy} \leftarrow H(I_{idx,idy})$;
3 $value = P\ (1-Jidx,idy)\ idx,\ idy;$
4 $R_{idx,idy} \leftarrow value \times S_{idx,idy}$;

TABLE 1. Configuration information of hardware and software.

Configuration item	Details
CPU	Intel® Xeon(R) CPU E5-2650 v4 @ 2.20 GHz × 48
GPU	NVIDIA Tesla P100-PCIE-16 GB × 2
Memory	512 GB
Hard disk	16 TB
Solid-state disk	1.2 TB
OS	Ubuntu 16.04.2 LTS (64-bit)
OpenCV	OpenCV 2.4.13
CMake	CMake 3.5.1
CUDA	CUDA V8.0.61
GDB	DNU gdb 7.6.2

IV. EXPERIMENTAL EVALUATION

A. EXPERIMENTAL SETUP

The experiment is performed on an Ubuntu workstation equipped with an Intel(R) Xeon(R) CPU 5-2650 v4 @ 2.20 GHz, accelerated by two NVIDIA Tesla P100 GPUs. The NVIDIA Tesla P100 has 3,584 CUDA cores and 16 GB of HBM2 memory. The core frequency is up to 1,328 MHz and the floating-point performance is 10.6 TFLOPS. Additional configuration parameters are listed in Table 1.

B. EXPERIMENTAL RESULTS AND ANALYSIS

1) IMAGE VISUAL EFFECT

The test set was obtained from the picture test suite of the pencil drawing generation algorithm. All parameters are consistent with the sequential algorithm [28]. This paper

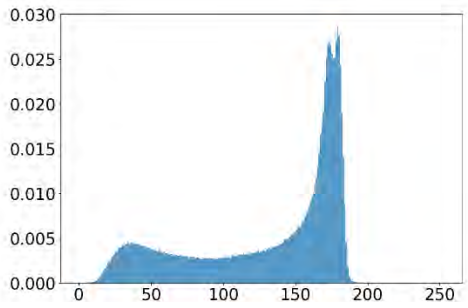


FIGURE 11. Grayscale histogram of a real pencil drawing.

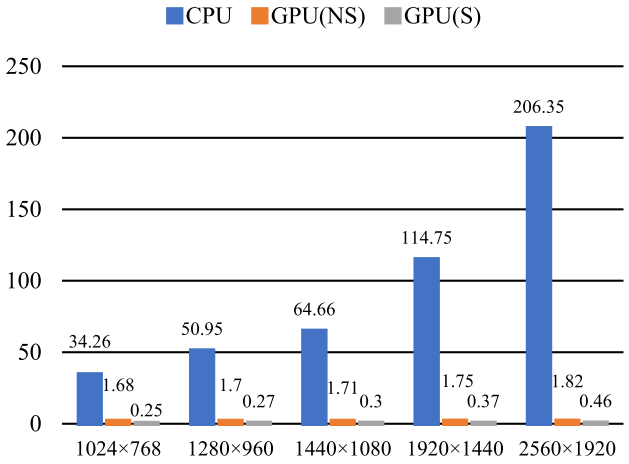


FIGURE 12. Time-costs of the algorithm with CPU, GPU and GPU Stream.

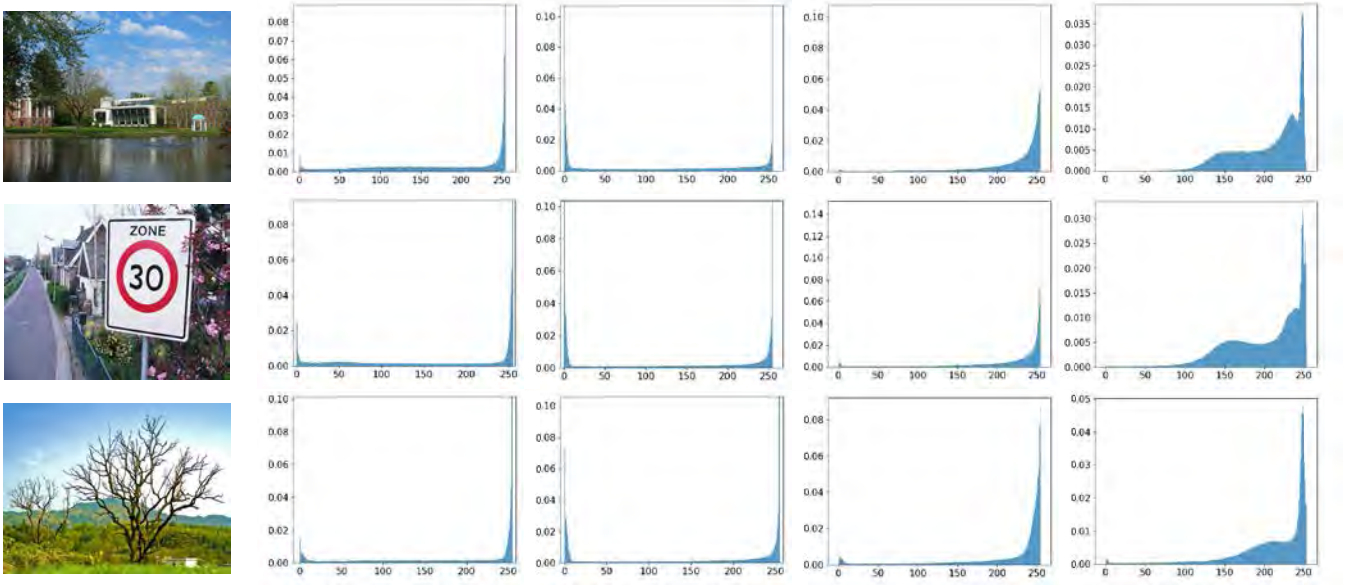
compares and analyzes the pencil drawing software and the LIC algorithm. The comparison result is presented in Figure 13. Compared with the algorithm based on LIC, the lines drawn by the algorithm proposed in this paper are more delicate, the texture of the pencil drawing is clearer, and the result is closer to the effect of pencil drawing. Compared to the AKVIS Sketch, a software based on DIP, the color tone is slightly flatter and does not reflect the shadow effect of pencil painting. The results of the algorithm proposed in this paper are rich in hue, and the algorithm can simulate the stereo effect of pencil painting very well. Compared with Pencil Sketch, another software based on DIP, it has obvious granular sensation and serious distortion in rendering people's face. The proposed algorithm has fine texture and can outline the portraiture well.

Figure 11 is a real grayscale histogram of a hand-drawn sketch. The effect of pencil drawing can be quantitatively analyzed via the analysis of the grayscale histogram. The graph of the established tone distribution model is shown in Figure 14. By comparing the tone distribution maps of the pencil drawings that were generated by LIC, AKVIS Sketch and Pencil Sketch, it is possible to judge how close the pencil drawings generated by the algorithms are to the actual pencil drawing.

The grayscale histograms of the pencil drawings that were produced by various algorithms or software in the previous

(a) original images (b) LIC (c) AKVIS Sketch software (d) Pencil Sketch software (e) the proposed algorithm

FIGURE 13. Comparison with the LIC algorithm, the AKVIS sketch software and the pencil sketch software.



(a) original images (b) LIC (c) AKVIS Sketch (d) Pencil Sketch (e) The proposed algorithm

FIGURE 14. Grayscale histogram comparison.

section were calculated using Matplotlib and their tone distribution maps were plotted. The tone distributions of the results of the previous algorithm and software are shown in Figure 14. According to the figure, the pencil drawing result of the algorithm is closest to the tone distribution model of the actual pencil drawing; therefore, the processing effect is closest to that of the actual pencil drawing. The grayscale histogram results demonstrate that grayscale is distributed in the middle layer in this paper and these pixels are used to describe the shadow structure and stereoscopic effect of the image. There is a maximum when the grayscale is close to 0, which is the line that represents the black outline.

2) SPEEDUP PERFORMANCE

The three sets of pictures in the test set in the experiment were scaled using the image processing software. Each group received images of 5 sizes for the test set of this experiment: 1024×768, 1280×960, 1440×1080, 1920×1440 and 2560×1920. The performances of CPU algorithms, GPU algorithms without CUDA Stream (GPU(NS)), and GPU algorithms with CUDA Stream (GPU(S)) are evaluated on these three sets of images.

The runtime diagrams for the algorithms, which were obtained from a set of pictures, are shown in Figure 12, where the horizontal axis corresponds to the size of the image and the vertical axis to the average time-cost in seconds over 10 runs.

According to the experimental results, the running time of the CPU serial algorithm increases with the image size and exhibits a general trend of geometric growth. When the image area increases from 1440×1080 to 1920×1440, the area of the image is doubled and the computation time is doubled. However, the time-costs of GPU parallel algorithm is relatively stable and slowly growth by the size of the picture within the range of picture sizes that were evaluated. It is because that, by the resolution is increased, the GPU can use more threads to process but the CPU still run in sequentially. Therefore, as the size of the image increases, the performance advantage of the GPU algorithm increases.

The speedup is a general measure of performance improvement via parallel program optimization. It represents the ratio of the running time of the pre-optimization program to the running time after optimization and is calculated via Eq.12.

$$S = \frac{T_s}{T_p} \tag{12}$$

The above three sets of pictures are processed and the average computation times over ten runs are calculated. The average of these results is calculated as the average speedup, which illustrated in Figure 15.

3) COST ANALYSIS

Using NVIDIA's program performance analyzer, namely, NVPROF, the execution information of each function of the program is obtained, including the execution time and the number of calls. The time-cost execution results analysis at 1280×960-resolution image input is presented in Table 2. The percentage of time-cost is shown in Figure 16. Here, memory copy DtoH refers to data copying from GPU memory to CPU memory and memory copy HtoD refers to data copying from CPU memory to GPU memory.

The convolution sub-algorithm occupies a substantial portion of the runtime. The average execution time was $905.75\mu s$ over the 16 times that it was called, which corresponds to 64.49% of the total time. The acceleration of the algorithm is mainly due to the acceleration of conv2D in the kernel

TABLE 2. Time-cost execution results from NVPROF.

Sub-algorithm	Time (%)	Time(ms)	Calls	Avg(μs)
Convolution	64.49	14.492	16	905.75
Memory copy DtoH	19.89	4.4704	6	745.06
Memory copy HtoD	8.50	1.9109	7	272.98
Classification	3.31	0.7435	8	92.949
Median filter	2.79	0.6265	1	626.53
Output	0.40	0.0902	1	90.209
Parameter model	0.28	0.0624	1	62.497
Gradient	0.14	0.0305	1	30.528
Result combination	0.09	0.0211	1	21.121
Histogram	0.09	0.0213	1	21.312

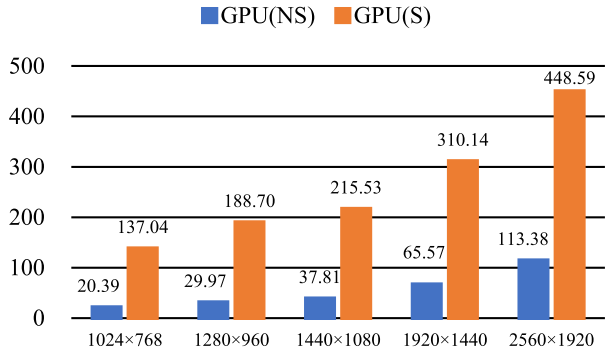
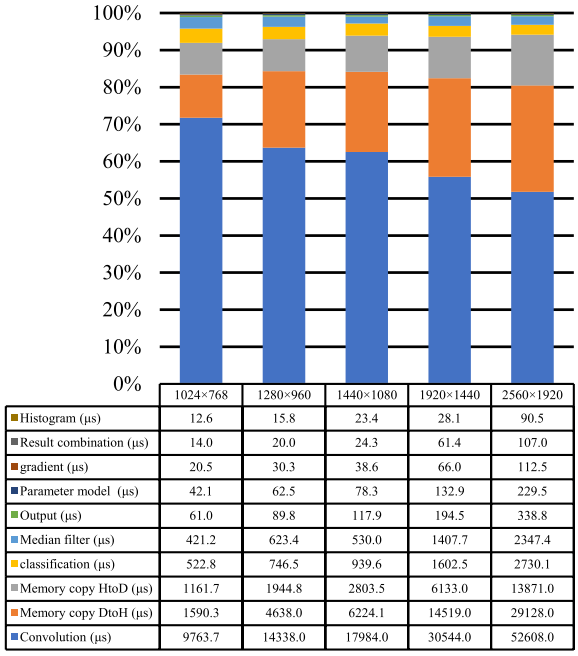


FIGURE 15. Speedups of the algorithm with GPU and GPU Stream.



	1024×768	1280×960	1440×1080	1920×1440	2560×1920
Histogram (μs)	12.6	15.8	23.4	28.1	90.5
Result combination (μs)	14.0	20.0	24.3	61.4	107.0
gradient (μs)	20.5	30.3	38.6	66.0	112.5
Parameter model (μs)	42.1	62.5	78.3	132.9	229.5
Output (μs)	61.0	89.8	117.9	194.5	338.8
Median filter (μs)	421.2	623.4	530.0	1407.7	2347.4
classification (μs)	522.8	746.5	939.6	1602.5	2730.1
Memory copy HtoD (μs)	1161.7	1944.8	2803.5	6133.0	13871.0
Memory copy DtoH (μs)	1590.3	4638.0	6224.1	14519.0	29128.0
Convolution (μs)	9763.7	14338.0	17984.0	30544.0	52608.0

FIGURE 16. Time-cost and percentage of each sub-algorithm.

function of the convolution operation in Sub-algorithm 1. Memory copying takes up a total of 28.39% of the time. However, due to the utilized CUDA streaming technology, this part of the time can be overlapped with the calculation process. In practice, it takes less time than this value. The median filter should have the same average time as the convolution because they have the same data input scale and parallel design. However, because of the simplified design of the sorting algorithm in this paper, the average execution time is reduced by 30.83% for convolution. The average execution time of pixel classification is only about 93μs but occupied 3.31% of the total time because the sub-algorithm is called 8 times in different directions. At last, the rest of sub-algorithms takes up no more than 1% of the total time. In general, the time-cost of all sub-algorithms reach effect within expectations and the most time-consuming part is fully paralleled.

V. RELATED WORK

Currently, with the development of video games and animated movies, non-realistic rendering technology is attracting increasing attention. Many relevant research studies have been reported and many algorithms have been proposed that achieve a satisfactory visual effect [29]–[34]. In [32], Liang et al. proposed a method for automatically stylizing portrait videos that contain small human faces that extends the mask regions with the convolutional neural network (R-CNN) features. The experimental results demonstrated that the method could effectively preserve the small and distinct facial features. In [29], Hiraoka et al. proposed a method for generating oil-film-like images via iterative processing between a bilateral infra-envelope filter and an unsharp mask. The results of the tests offered guidelines for generating oil-film-like images from various color photo images. In [31], Gao et al. proposed a framework for chromatic penciling style generation from wild photographs. According to a series of comparative experiments with previous work, the proposed method more accurately captured the main features of real chromatic pencil drawings and the results had an improved visual appearance. An image can be depicted very beautifully by utilizing these algorithms. However, these algorithms require a long time to complete the painting; hence, it is impossible to perform the task in real time.

With the development of parallel computing technology, especially GPUs, image processing tasks with many computations can be completed quickly. In [35], Royer et al. proposed an adaptive cartoon-like stylization method for color videos in real time. This framework consists of color space conversion, bilateral filtering, color quantization and edge detection. For running in real time, a parallel version was implemented on the GPU, which runs 45 to 180 times faster than the CPU version. In [36], Park et al. proposed a non-photorealistic rendering technique that renders three-dimensional objects and photo images into cartoon and sketch styles. To accelerate the time-consuming segmentation procedure, Park et al. used GPGPU for the parallel computing using the GPU. As a result, it is 5.52 times faster than the original version. In [37], Lu et al. discussed a non-photorealistic real-time virtual sculpting system. Via touch-enabled manipulation, haptic interaction is conducted to form the deformable surfaces and the edge extraction is employed to stress boundaries. The experimental results demonstrated that the parallel realization of the system on GPU guarantees real-time performance.

These results demonstrate that the GPU facilitates the acceleration of NPR. Compared with these studies, this paper focuses on developing an efficient method for rendering pencil paintings and maximizing the parallelism of the algorithm.

VI. CONCLUSION

For performing real-time rendering and accelerating pencil drawing generation, this paper proposes a parallel fast pencil drawing generation algorithm based on GPU. First, the parallelism of the pencil drawing generation algorithm is identified via theoretical analysis. Then, the CUDA programming model is used to design and implement the parallel pencil drawing algorithm. Next, the data access time is reduced by designing an optimized memory access mode. Finally, the running speed of the parallel algorithm is further accelerated via parallel execution of independent sub-tasks using CUDA streaming technology. In addition, the synchronization and atomic operations ensure the high accuracy of the results.

The parallel fast pencil drawing generation algorithm is implemented on the CUDA platform. The results demonstrate that the CUDA-based parallel algorithm can obtain a maximum speedup of 448.59 times on an NVIDIA Tesla P100 graphics card, which represents a significant decrease in execution time. Using an image test set, the fast pencil drawing generation algorithm yields more realistic results than other software and methods. The parallelization of the algorithm based on combining a sketch and tone enables real-time rendering of pencil drawings.

REFERENCES

[1] J. Zhang, R.-Z. Wang, and D. Xu, "Automatic genaration of sketch-like pencil drawing from image," in *Proc. IEEE Int. Conf. Multimedia Expo Workshops (ICMEW)*, Jul. 2017, pp. 261–266.
[2] Q. Kong, Y. Sheng, and G. Zhang, "Hybrid noise for LIC-based pencil hatching simulation," in *Proc. IEEE Int. Conf. Multimedia Expo (ICME)*, Jul. 2018, pp. 1–6.
[3] H. Yang and K. Min, "Simulation of color pencil drawing using LIC," *KSII Trans. Internet Inf. Syst.*, vol. 6, no. 12, pp. 3296–3314, 2012.
[4] M. Hata, M. Toyoura, and X. Mao, "Automatic generation of accentuated pencil drawing with saliency map and LIC," *Vis. Comput.*, vol. 28, nos. 6–8, pp. 657–668, 2012.
[5] H. Yang, Y. Kwon, and K. Min, "A stylized approach for pencil drawing from photographs," *Comput. Graph. Forum*, vol. 31, no. 4, pp. 1471–1480, 2012.
[6] B. Cabral and L. C. Leedom, "Imaging vector fields using line integral convolution," in *Proc. 20th Annu. Conf. Comput. Graph. Interact. Techn.*, 1993, pp. 263–270.
[7] X. Gao, J. Zhou, Z. Chen, and Y. Chen, "Automatic generation of pencil sketch for 2D images," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, 2010, pp. 1018–1021.
[8] M. Hata, M. Toyoura, and X. Mao, "Automatic pencil drawing generation using saliency map," in *Proc. ACM Symp. Appl. Perception*, 2013, p. 119.
[9] C. Zhao, B. Gao, W. Deng, H. Zhang, "A pencil drawing algorithm based on wavelet transform multiscale," in *Proc. Prog. Electromagn. Res. Symp. Fall (PIERS–FALL)*, Nov. 2017, pp. 73–79.
[10] X. Cai and B. Song, "Image-based pencil drawing synthesized using convolutional neural network feature maps," *Mach. Vis. Appl.*, vol. 29, no. 3, pp. 503–512, 2018.
[11] T. Umenhoffer, L. Szirmay-Kalos, L. Szécsi, Z. Lengyel, and G. Marinov, "An image-based method for animated stroke rendering," *Vis. Comput.*, vol. 34, nos. 6–8, pp. 817–827, 2018.
[12] X. Yang, L. Jian, W. Wu, K. Liu, B. Yan, Z. Zhou, and J. Peng, "Implementing real-time RCF-Retinex image enhancement method using CUDA," *J. Real-Time Image Process.*, vol. 16, no. 1, pp. 115–125, 2019.
[13] N. Baek and K. J. Kim, "An artifact detection scheme with CUDA-based image operations," *Cluster Comput.*, vol. 20, no. 1, pp. 749–755, Mar. 2017.
[14] Y. Yuan, X. Yang, W. Wu, H. Li, Y. Liu, and K. Liu, "A fast single-image super-resolution method implemented with CUDA," *J. Real-Time Image Process.*, vol. 16, no. 1, pp. 81–97, 2019.
[15] C. Cuenca, E. González, A. Trujillo, J. Esclarín, L. Mazorra, L. Alvarez, J. A. Martínez-Mera, P. G. Tahoces, and J. M. Carreira, "Fast and accurate circle tracking using active contour models," *J. Real-Time Image Process.*, vol. 14, no. 4, pp. 793–802, 2018.
[16] F. E. Sayadi, M. Chouchene, H. Bahri, R. Khemiri, and M. Atri, "CUDA memory optimisation strategies for motion estimation," *IET Comput. Digit. Techn.*, vol. 13, no. 1, pp. 20–27, 2019.
[17] I. M. Orak and A. Celik, "A parallel algorithm for defect detection of rail and profile in the manufacturing," *J. Fac. Eng. Archit. Gazi Univ.*, vol. 32, no. 2, pp. 439–448, 2017.
[18] A. Y. Chekunov, "Implementation of the fast algorithm for geometrical coding of digital images with the use of CUDA architecture," *Moscow Univ. Math. Bull.*, vol. 73, no. 6, pp. 229–238, 2018.
[19] G. Georgis, G. Lentaris, and D. Reisis, "Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution," *J. Real-Time Image Process.*, vol. 965, no. 1, pp. 1–28, Jul. 2016.
[20] S. Santander-Jimenez, M. A. Vega-Rodríguez, J. Vicente-Viola, and L. Sousa,"Comparative assessment of GPGPU technologies to accelerate objective functions: A case study on parsimony," *J. Parallel Distrib. Comput.*, vol. 126, pp. 67–81, Apr. 2019.
[21] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 4–13, Jan. 2013.
[22] B. Qin, Z. Wu, F. Su, and T. Pang, "GPU-based parallelization algorithm for 2D line integral convolution," in *Proc. Int. Conf. Swarm Intell.*, 2010, pp. 397–404.
[23] D. Lu, "Information-theoretic exploration for texture-based visualization," *J. Vis.*, vol. 20, no. 2, pp. 393–404, 2017.
[24] D. Xie, Y. Zhao, and D. Xu, "An efficient approach for generating pencil filter and its implementation on GPU," in *Proc. 10th IEEE Int. Conf. Comput. Aided Design Comput. Graph.*, Oct. 2007, pp. 185–190.
[25] J. Xiao and L. Yao, "Study on 3D sketch rendering based on GPU," in *Proc. Int. Conf. Image Anal. Signal Process.*, Oct. 2011, pp. 351–354.
[26] X. Zhu, X. Jin, S. Liu, and H. Zhao, "Analytical solutions for sketch-based convolution surface modeling on the GPU," *Vis. Comput.*, vol. 28, no. 11, pp. 1115–1125, 2012.
[27] Z. Chen, Y. Jin, B. Sheng, P. Li, and H. Sun, "Parallel pencil drawing stylization via structure-aware optimization," in *Proc. 31st Int. Conf. Comput. Animation Social Agents*, 2018, pp. 32–37.
[28] C. Lu, L. Xu, and J. Jia, "Combining sketch and tone for pencil drawing production," in *Proc. Symp. Non-Photorealistic Animation Rendering*, 2012, pp. 65–73.
[29] T. Hiraoka and K. Urahama, "Generation of oil-film-like images by bilateral infra-envelope filter," *IEICE Trans. Inf. Syst.*, vol. 99, no. 6, pp. 1724–1728, 2016.
[30] D. Cui, Y. Sheng, and G. Zhang, "Image-based embroidery modeling and rendering," *Comput. Animation Virtual Worlds*, vol. 28, no. 2, 2017, Art. no. e01725.
[31] C. Gao, M. Tang, X. Liang, Z. Su, and C. Zou, "PencilArt: A chromatic penciling style generation framework," *Comput. Graph. Forum*, vol. 37, no. 6, pp. 395–409, 2018.
[32] D. Liang, K. Park, and P. Krompiec, "Facial feature model for a portrait video stylization," *Symmetry*, vol. 10, no. 10, pp. 1–14, 2018.
[33] W. Qian, D. Xu, J. Cao, Z. Guan, and Y. Pu, "Aesthetic Art Simulation for Embroidery Style," *Multimedia Tools Appl.*, vol. 78, no. 1, pp. 995–1016, 2019.
[34] W. Qian, D. Xu, K. Yue, Z. Guan, Y. Pu, and Y. Shi, "Gourd pyrography art simulating based on non-photorealistic rendering," *Multimedia Tools Appl.*, vol. 76, no. 13, pp. 14559–14579, 2017.
[35] Q. Wang, D. Chen, S. Li, Q. Wu, and Q. Zhang, "An adaptive cartoon-like stylization for color video in real time," *Multimedia Tools Appl.*, vol. 76, no. 15, pp. 16767–16782, 2017.

[36] H.-C. Yoon and J.-S. Park, "Non-photorealistic rendering using CUDA-based image segmentation," *KIPS Trans. Softw. Data Eng.*, vol. 4, no. 11, pp. 529–536, 2015.
[37] P. Lu, B. Sheng, S. Luo, X. Jia, and W. Wu, "Image-based non-photorealistic rendering for realtime virtual sculpting," *Multimedia Tools Appl.*, vol. 74, no. 21, pp. 9697–9714, 2015.

JIYAN QIU was born in Jiangxi, China, in 1998. He is currently pursuing the B.S. degree in information management and information system from Northwest A&F University, China. His research interests include parallel computing and computer vision.

BIN LIU was born in Shaanxi, China, in 1981. He received the B.S. degree in computer science and technology from the Shaanxi University of Science and Technology, China, in 2004, the M.Sc. degree in technology with a major in parallel computing and cloud computing from Yunnan University, China, in 2010, and the Ph.D. degree in electronic and information engineering from Xi'an Jiaotong University, China, in 2014.

Since 2018, he has been an Associate Professor with the College of Information Engineering, Northwest A&F University, China. He is also a Postdoctoral Fellow with the College of Mechanical and Electronic Engineering, Northwest A&F University. His research interests include cloud computing, parallel computing, and computer vision. He serves as a Reviewer for IEEE Access, the IEEE Transactions on Computers, and *The Journal of Supercomputing*.

JINRONG HE was born in Gansu, China, in 1984. He received the B.S. degree in information and computing science and technology and the M.Sc. degree in science with a major in computational mathematics from the Wuhan University of Technology, China, in 2007 and 2010, respectively, and the Ph.D. degree in computer software and theory from Wuhan University, China, in 2014.

From 2015 to 2018, he was a Lecturer with the College of Information Engineering, Northwest A&F University, China. He is currently a Lecturer with the College of Mathematics and Computer Science, Yan'an University, China. He is also a Postdoctoral Fellow with the College of Mechanical and Electronic Engineering, Northwest A&F University. His research interests include machine learning and computer vision.

CHAOYANG LIU was born in Henan, China, in 1995. He received the B.E. degree in computer science and technology from Northwest A&F University. His research interests include parallel computing and computer vision.

YUANCHENG LI was born in Henan, China, in 1981. He received the B.S. degree in computer science and technology from the Xi'an University of Posts and Telecommunications, China, in 2004, and the M.Sc. degree in computer software theory and the Ph.D. degree in computer system structure from Xi'an Jiaotong University, China, in 2007 and 2012, respectively.

Since 2012, he has been a Lecturer with the College of Computer Science and Technology, Xi'an University of Science and Technology, China. His research interests include parallel computing and compiler optimization. He serves as a Reviewer for *Concurrency and Computation: Practice and Experience* and the *Journal of the Chinese Institute of Engineers*.