computergraphics tübingen

Cuda Implementation of Advanced Pencil Sketch Filter

Andreas Altergott, Raphael Braun, Stefan Burnicki

August 27, 2014

Practical course:
Massive Parallel Programming SS2014
Computer Graphics

Cuda Implementation of Advanced Pencil Sketch Filter

Andreas Altergott, Raphael Braun, Stefan Burnicki

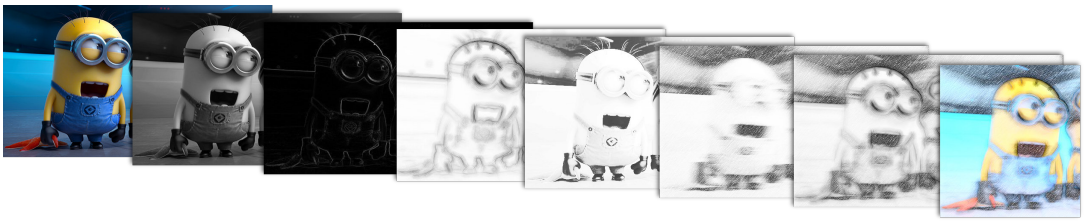

Figure 1: From left to right: natural input image, grayscale, gradient, line filter result, result of histogram matching, hatching texture, combined results, colored result

Abstract

Anyone?

1 Introduction

Creating pencil sketches have a very long history. Throughout time artist have perfected the art of creating pencil drawings in all thinkable styles, ranging from completely abstract to stunningly photorealistic. However such skills are rare, and only well trained artists or passionated hobby artists are able to actually create pleasing looking images using nothing but the eye, a paper and a pencil. Today, in a time where cameras can be used to capture a realworld scene in a matter of seconds, with accurate proportions and colors and no skills, the need for pencil sketches seem to be gone. However, photographs do not convey the same emotions that sketches do. There is a certain timeless flair to pencil sketches that makes them just nice to look at.

Therefore designing an pencil sketch filter, which produces a pencil sketch from a regular photograph is an exciting task. The filter described in [LXJ12] is such a filter, which consists of two stages: sketching the outlines and drawing the shading. In the sketch stage the filter calculates several line convolutions and the shading stage solves a huge linear system of equations. Both tasks are computationally rather expensive and according to the authors their implementation of the filter takes 2 seconds to run on a 600×600 image. Figure 1 shows input, intermediate results and the final result of the filter.

This report presents our efforts to implement the entire filter on the GPU using Cuda, to gain realtime results. The structure of the report is as follows: The filter itself is described in Section 3, then in Section 4 we describe our implementation using Cuda. Section 5 shows our benchmarking results for various input images and Section 6 some possible optimizations and applications for our implementation.

2 Previous Work

This work is based on [LXJ12], as this paper describes the filter that is implemented. The filter requires quite some parameters. In [LXJ12] they use a parameter lerning approach to automatically select those parameters. In this work the filter is kept slightly simpler than in the origin paper, e.g. the line filter is slightly simplified and the filter parameters are left as user controllable options.

Another GPU-sketch-filter is described in [LSF10], though it uses the a regular shader-pipeline. Based on the neighbourhood informations of each pixel it calculates weather to place a stroke at this point or not. The strokes are then rendered as stroke-textures. The strokes are made in three different detail layers based on the gradientmagnitude. Furthermore it is possible to filter animations using optical flow. The greatest difference to the filter that we implement is that

they calculate each individual pencil or brush stroke based on the image and then render those strokes using stroke-textures, and our method uses traditional filters to alter the original image. Their filter is quite versatile, as it allows different painting stiles by changing the stroke-textures and weighting the detail layers.

A partially GPU based method is described in [BLC+12]. However they concentrate on rendering temporal coherent line drawing animations from a given 3D-scene. First they find active contours that are then adapted throughout the animation. The result pictures are rendered by parameterizing the and then add a certain style to the path. While this method yields amazing results for very stylistic animations, it needs 3D input data.

3 Method

The filter considers *line scetching* and *shading* separately. The following subsections shed some light on how both stages work precisely.

The input for the filter is a regular RGB-image I_{rgb}. In the first step a grayscale image I_g is calculated using the Y-channel of the YUV-transformed image I_{yuv}. I_g now serves as input for the future steps.

3.1 Line Sketching

One prominent task of the filter is to create sketchy looking outlines. The main objective here is to mimic freehand sketching, which is typically done by drawing short straight lines aligned to the outlines. The outlines are detected using a simple gradient operator. Then each pixel is assigned to a line direction and finally the pixel is drawn as part of its line.

Gradient Image The Outlines are detected and stored in the image G using gradient magnitudes:

$$G = ((\partial_x G)^2 + (\partial_y G)^2)^{\frac{1}{2}}$$

In the center of Figure 2 you can see how G looks like.

Line Convolution Filter Given just the gradient magnitudes G for each line direction $i \in \{1, \cdots, N\}$ where N is the total number of lines, the convolution between G and each line segment $\mathscr{L}_i$ is calculated.

$$G_i = \mathscr{L}_i * G \tag{1}$$

The value $G_i(p)$ of pixel p will be very big, if that pixel lies directly on a line in G (edge) and if $\mathscr{L}_i$ is following this line, such that only big values are collected in the convolution. If the pixel doesn't lie on or close to a line it can not gather high values and therefore stay dark. Pixels which lie very close to edges can still gather some brightness if the line segment $\mathscr{L}_i$ intersects the edge. This way lines in G which follow the direction of $\mathscr{L}_i$ show up and slightly overshoot in G_i.

Now to actually draw the lines, each pixel selects its maximum value from all G_i:

$$L = \max(\{G_i\}) \quad \forall i \in \{1, \cdots, N\} \tag{2}$$

In a final step L is inverted, such that the lines are dark and the rest is white. The result can be seen on the right side in Figure 2.

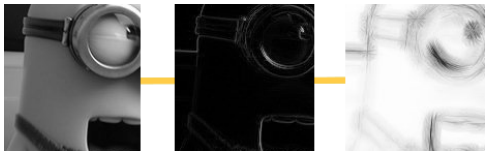

Figure 2: The intermediate results when calculating the line sketches. Left: input image, Center: Gradient image, Right: result L.

3.2 Shading

The other important step in creating a believable pencil sketch image from a natural image is to produce a hatching texture to create the shading. This is done in two steps. First the histogram of the input image I_g is matched to a histogram model that was derived in [LXJ12]. This way the tone distribution is forced to correspond to tone distributions that were measured in real pencil sketch images. Then the image of a given hatching pattern is used to render the hatching texture for the input image.

Histogram Matching Tones in natural images do not follow any specific pattern. In pencil drawings however the tones are basically created only by two basic tones, namely the white paper and the graphite strokes in different strengths. Heavy strokes are used in very dark areas, mid tone strokes are used to produce impression of 3D layered information and in bright areas the paper is just left white. Figure 3 shows the tone distributions of

some real pencil sketches. One can easily see the three regions, the peak in the dark regions, which represent the heavy strokes, the constant distribution in the mid tones, which are used for the layering and very much bright pixels, originating from the white paper, that was just left blank.

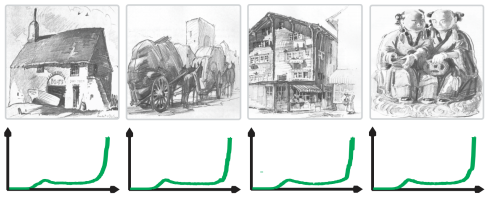

Figure 3: Examples for real pencil sketches and their measured tone distributions. Note: This image was taken from [LXJ12]

[LXJ12] used this observation to create a parametric histogram model for pencil drawings which consists of three functions, which represent those three tone levels:

For the bright part of the histogram they use a Laplacian distribution with a peak at the brightest value. This adds some variation in the bright areas, which originate from slight illumination variances or the use of a eraser.

$$p_1(v) = \frac{255}{\sigma_b} e^{-\frac{255-v}{\sigma_b}} \tag{3}$$

The parameter for this function is just σ_b, which controls the sharpness of the function. This distribution can be seen on the very right of the histograms in Figure 3.

The mid layer is composed of strokes with different pressures and therefore in different gray levels. So the distributions of those gray levels is equally distributed as indicated in the histograms in Figure 3. To represent this part a constant function was chosen to use all those possible gray levels.

$$p_2(v) = \begin{cases} \frac{1}{u_b - u_a} & \text{if } u_d < v \leq u_b \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The controlling parameters for this function are the range boundaries u_d and u_b.

Finally the dark region which shows up as this bell shaped peak in the dark regions in Figure 3 is represented as a Gauss-curve. The position and shape of the dark regions depend on the maximum pressure an artist is using, and the softness of the pencil that is used.

$$p_1(g) = \frac{1}{\sqrt{2\pi}\sigma_d} e^{-\frac{(v-\mu_d)^2}{2\sigma_d^2}} \tag{5}$$

The width of the bell is controlled with the parameter σ_d and the position with μ_d.

Plots for the three functions can be seen in Figure 4.

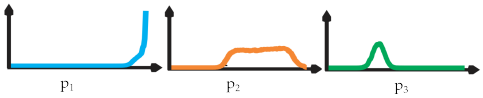

Figure 4: Plots of the three functions p_i. Note: picture taken from [LXJ12].

The final tone distribution is now simply composited out of those three function by creating a weighted sum from p_1, p_2 and p_3:

$$p(v) = \frac{1}{Z} \sum_{i=1}^{3} \omega_i p_i(v) \tag{6}$$

Where Z is a normalization factor to make $\int_0^{255} p(v)dv = 1$ and the ω_i are weighting parameters which can used to weight the importance of the functions.

In [LXJ12] they learned the parameters for those functions from a set of different styled pencil sketches using Maximum Likelihood Estimation. We skipped this part and left those parameters to be controlled by the user.

Histogram Matching is used to apply the tone distribution from Equation 6 to the input image. The result J is then used to calculate the hatching texture. J can be seen on the right side of Figure 5.

Figure 5: Left input. Right: Result of Histogram matching for minions.

Texturing The filter uses a men made pencil hatching pattern H. A human repetitively draws at the same position to generate the correct tone in the hatching texture. A exponential function can be used to simulate this process of placing multiple strokes at the same position: $H(x)^{\beta(x)} \approx J(x)$. This corresponds to drawing H β times at the same position to approximately match the tone that is dictated by our tonal map J. In the logarithmic domain this boils down to $\beta(x) \ln(H(x)) \approx \ln(J(x))$.

Just solving for β in this equation is going to destroy the hatching pattern because β can be calculated for each pixel independently, such that in the end $H^\beta = J$. Therefore a smoothness constraint is introduced:

$$\beta = \arg\min_{\beta} \|\beta \ln(H) - \ln(J)\|_2^2 + \lambda \|\nabla\beta\|_2^2 \quad (7)$$

The smoothness weighting factor λ can be used to determine how strong the hatching pattern H will show through in the result.

The hatching texture T is then calculated as the pixelwise exponentiation

$$T = H^\beta$$

Figure 6 shows the result for the minion.

Figure 6: Left: J. Right: Result of texture rendering T.

Combining Results Finally the results from the line sketching L and the hatching texture T is combined to the finished pencil drawing R by simply multiplying the images pixelwise:

$$R = L \cdot T$$

If desired it is also possible to create colored pencil drawings using the YUV decomposed image I_{yuv} from the beginning and replace the Y channel with R. The resulting RGB-image can then easily be calculated by a color space transformation.

In Figure 7 both, grayscale and colored results are shown.

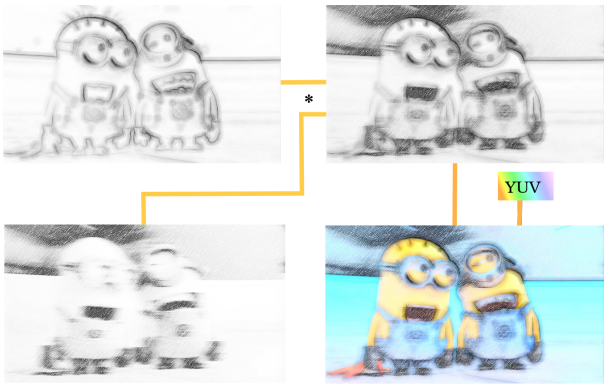

Figure 7: combination of the results from the line drawing and shading.

4 GPU Implementation

Figure 8 displays the GPU and CPU pipeline of the implementation. The first step loads the image, which happens on the CPU. The conversion of the image from RGB to YUV on the GPU is run in parallel with the loading of the pencil textures on the CPU. The next kernel extracts the gray-scale version of the image from the RGB to YUV result. The gray-scale version is used for the gradient calculation, which is then applied to the convolution filter. This accomplishes the line drawing step. In preparation for the tone drawing step, the CPU is computing the target histogram. The histogram calculation in the tone drawing step receives the gray-scale version as the input image. The result is being passed to the histogram mapping, together with the computed target histogram. Another kernel receives the loaded pencil textures and expands them to the image size. The expanded textures and the histogram mapping are being used for the texture calculation. The last kernel in the tone drawing, the texture rendering kernel, computes the result of the tone drawing. The last two kernels represent the merge. One kernel combines the results of the line drawing and the tone drawing step and generates the final image. The last kernel applies the coloring if requested and converts the YUV back to RGB. The CPU takes care of saving the image.

4.1 Sketch Filter

Calculating the grayscale image I_g and Gradient G on the GPU is no big challenge. However the effi-

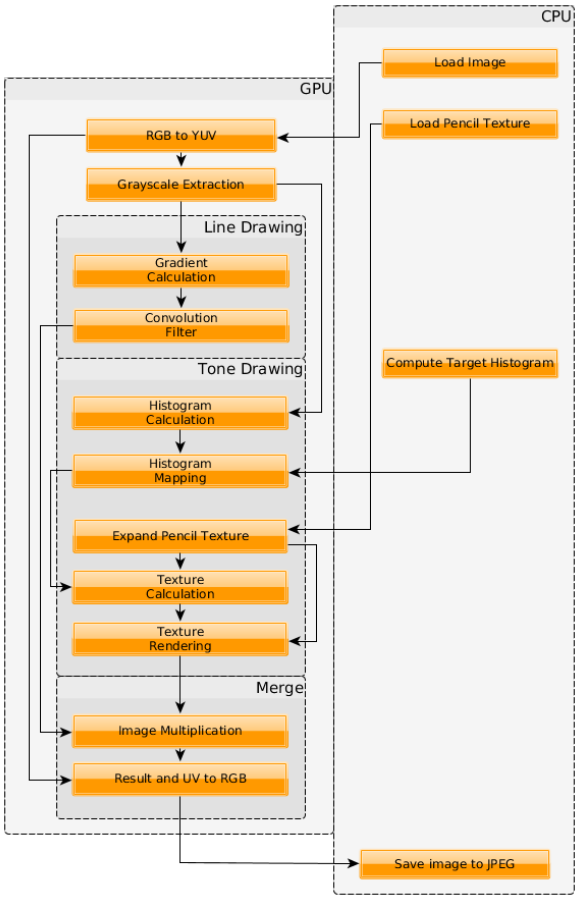

Figure 8: GPU and CPU pipeline of the implementation

cient implementation of the line convolution calculation is more interesting.

To calculate the value $L(x)$, first all $G_i(x)$ have to be computed (see Equation 1) and the maximum from all line convolution results is selected (see Equation 2). The following paragraphs describe the implementation of the Cuda-Kernel, which computes L directly from G given the desired line length and strength.

Compute the line convolutions To compute the convolution result $G_i(x)$ for pixel x and line i all values of G along the line segment $\mathscr{L}_i$ are collected and averaged. The convolution-kernel of the line segment $\mathscr{L}_i$ can be described as

$$k_i(p) = \begin{cases} \frac{1}{\text{line length} \cdot \text{line strength}} & p \text{ is part of } \mathscr{L}_i \\ 0 & \text{otherwise} \end{cases}$$

Collecting the right pixels for the right line is done by iterating over the pixels of a horizontal line with the desired length and strength. This line starts at x. Then the coordinates are rotated to get the pixels for line i. All line convolutions for one pixel is calculated by a single thread. The results for a line is keep as maximum if it is bigger than all its predecessors.

Finally the inversion, a gamma correction and clipping is used to create the final value for $L(x)$:

$$L(x) = \max(255 - \max(G_i)^\gamma, 50)$$

The γ parameter can be used to intensify or weaken the lines.

Shared Memory The same pixel values from G have to be loaded from neighboring quite often. Therefore it makes sens to use shared memory to speed up memory access. As each pixel is calculated by a individual thread, and we can only use 1024 threads in one block we get blocks of size 32×32, as shown in Figure 9

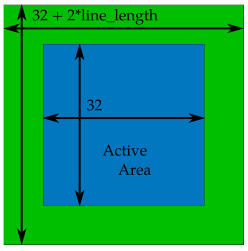

Figure 9: Block and shared memory dimensions

On our device the maximum size of shared memory per block is $48kb$, so we can only store a 109×109 block of float values. Therefore we only allow line lengths up to 45 pixels.

The data from G is copied by assigning the thread number to the linear indexes of the data. As there is more data than threads, one thread copies multiple data elements. The pattern is shown in Figure 10. Figure 10.

The coordinate calculations for the line convolution computations are done such that they generate the correct coordinates for the shared memory

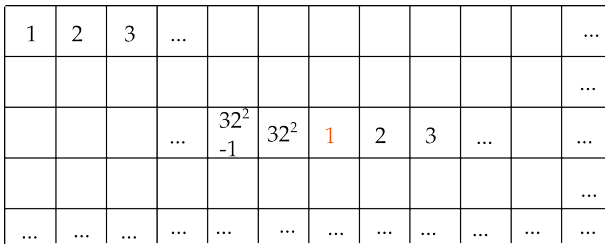

Figure 10: Thread number assignment when copying the data from G to shared memory.

block. Due to the coordinate rotations the access patterns are very chaotic, which might lead to unavoidable bank conflicts.

Coordinates which lie outside the input image must not be included in the copying process and later to calculate the convolution results.

4.2 Histogram

Andreeas

4.3 Histogram Matching

The histogram matching is used to adjust the tone distribution of the grayscale image to match a specific target tone distribution by using their corresponding histogram. In this context, the target tone distribution is artifically created to be similar to those of pencil scetches as explained above.

The cummulative target histogram is created on the CPU by simply applying the the parametric model to the values 0 to 255 and summing it up with the values below. This step is done on the CPU as the target distribution only has to be evaluated 256 times which is negligible effort and can be either be precomputed or done while the GPU is busy with other steps of the pipeline.

Once this histogram is uploaded to the GPU it can be used together with the histogram from the image created in the previous step to adjust the tones of the grayscale image. This is done in a kernel with one thread per image pixel. A thread will the obtain the tone value of the pixel it corresponds to and look up it's cumulative probabilityvalue in the source histogram map. Afterwards, the cummulative target historgam is used in iverse to find out which tone the value corresponds to in it. The pixel is then set to the looked up target tone value.

The inverse lookup in the cummulative target historgam is implemented as a binary search function which is used in the kernel. Although one would think that a binary search is inefficient when multiple threads are executed in a warp and search for different values, it really isn't. As the cummulative target histogram map only consist of 256 values, the binary search will need at maximum $log_2 256 = 8$ loop runs to end up at the correct value. Because the threads are executed in a warp, the branches might get executed in serial. However, there is just one real branching operation inside the loop body, which could double the effort. Still, the total effort negligible in contrast to a linear search through the map with a worst case complexity of 256.

4.4 Texturing

The Equation 7 has the same shape as a Tikhonov regularization and therefore can be solved exactly the same way by solving the following normal equation:

$$\underbrace{(AA^T + \lambda \cdot \Gamma\Gamma^T)}_{A^*} x = \underbrace{A^T b}_{b^*} \tag{8}$$

x is the solution image β in vector shape. A is a diagonal matrix. Each diagonal element corresponds to one pixel in $\ln(H)$. b is the vectorized $\ln(J)$ image and Γ is a diagonal matrix with two diagonals which represents if multiplied to a vectorized image the gradient operator.

The iterative Conjugate Gradient solver from the cuda sparce library (`cusp`) is used to solve the linear system $A^*x = b^*$.

For this the input images $\ln(J)$ and $\ln(H)$ are downloaded from the GPU, feed into diagonal matrices from `cusp` then again uploaded to solve the equation, once more downloaded and uploaded for the final composition with the line image L.

Matrix $A*$ and vector b^* (Equation 8) could be calculated using `cusp`. For this one would upload $\ln(H)$, $\ln(H)$ and Γ, and use the matrix operations included in `cusp` to calculate the matrix-matrix and matrix-vector multiplications. However it is much more effective to construct A^* and b^* on the CPU and upload just those results, because `cusp` needs an output target matrix for each operation. So a lot of unnecessary memory needs to be allocated and used for the calculations on the GPU. Due to the nice structure of the matrices, A^* and b^* can be calculated in $\mathcal{O}(n)$, with n = number of pixel in the input image. This is illustrated in Figure 11.

$$A* = \begin{pmatrix} a & 0 & -\lambda & 0 & 0 & 0 & 0 & \dots & \\ 0 & b & 0 & -\lambda & 0 & 0 & 0 & \dots & \\ -\lambda & 0 & b & 0 & -\lambda & 0 & 0 & \dots & \\ 0 & -\lambda & 0 & b & 0 & -\lambda & 0 & \dots & \\ 0 & 0 & -\lambda & 0 & b & 0 & -\lambda & \ddots & \\ \vdots & & & & \ddots & & & & \\ 0 & 0 & \dots & 0 & -\lambda & 0 & b & 0 & -\lambda \\ 0 & 0 & \dots & 0 & 0 & -\lambda & 0 & b & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & -\lambda & 0 & a \end{pmatrix}$$

Figure 11: Closed form for Matrix A^*. Substitute $a = \lambda + \ln(H(i))^2$ and substitute $b = 2\lambda + \ln(H(i))^2$ with i being the line in the matrix.

According to our tests assembling the Matrix on the CPU gains a total speedup of the program of factor 2.02 compared to the GPU method.

5 Performance

Stefan

6 Future Work

TODO

7 Conclusion

Anyone

References

[BLC+12] Pierre Bénard, Jingwan Lu, Forrester Cole, Adam Finkelstein, and Joëlle Thollot. Active Strokes: Coherent line stylization for animated 3D models. *NPAR 2012, Proceedings of the 10th International Symposium on Non-photorealistic Animation and Rendering*, June 2012.

[LSF10] Jingwan Lu, Pedro V. Sander, and Adam Finkelstein. Interactive painterly stylization of images, videos and 3D animations. In *Proceedings of I3D 2010*, February 2010.

[LXJ12] Cewu Lu, Li Xu, and Jiaya Jia. Combining sketch and tone for pencil drawing production. In Paul Asente and Cindy Grimm, editors, *Expressive*, pages 65–73. ACM, 2012.