computergraphics tübingen

# Cuda Implementation of Advanced Pencil Sketch Filter

Andreas Altergott, Raphael Braun, Stefan Burnicki

August 29, 2014

Practical course:
Massive Parallel Programming SS2014
Computer Graphics

# Cuda Implementation of Advanced Pencil Sketch Filter

Andreas Altergott, Raphael Braun, Stefan Burnicki

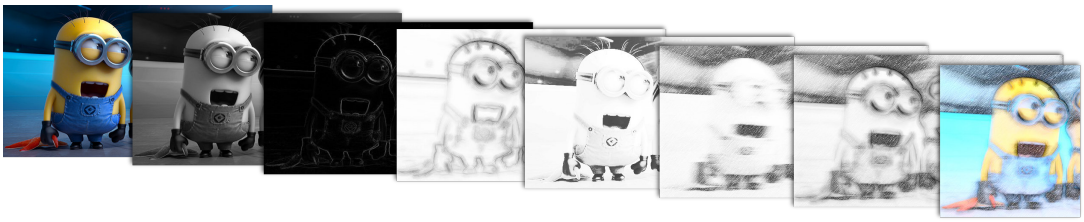

Figure 1: From left to right: natural input image, grayscale, gradient, line filter result, result of histogram matching, hatching texture, combined results, colored result

**Abstract**

This paper describes the Cuda implementation of an advanced pencil sketch filter, based on the work of Cewu Lu, Li Xu and Jiaya Jia [LXJ12]. It explains the used methods of the original paper and discusses the derived implementations for parallel computing. Details of the most complex kernels are explained throughout the paper, as well as some important modifications to improve performance. The result of this work are extremely fast computation times and very pleasing looking pencil sketch images.

## 1 Introduction

Throughout time artist have perfected the art of creating pencil drawings in all thinkable styles, ranging from completely abstract to stunningly photo realistic. However, such skills are rare, and only well trained artists or passionate 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 real world scene in a matter of milliseconds, with accurate proportions and colors, without requiring special skills, the need for pencil sketches seem to be gone. After all, 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.

For this reason designing an image 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 $600 \times 600$ 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 real time 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 benchmark results for various input images and in Section 6 some possible optimization and applications for our implementation are pointed out.

## 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 the original paper they use a parameter learning approach to automatically select those parameters. In this work the filter is kept slightly simpler than the original one, 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 a regular shader pipeline instead of Cuda. Based on the neighborhood information of each pixel it calculates whether 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 gradient magnitude. 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 gradient and then render those strokes using stroke-textures. Our method uses traditional filters to alter the original image, which produces better pencil sketches with respect to the tonal appearance. However, their filter is more versatile, as it allows different painting styles and techniques by changing the stroke-textures and weighting the detail layers.

Another partially GPU based method is described in [BLC+12]. However they concentrate on rendering temporal coherent line drawing animations from a given 3D-scene. To do so, they first find active contours that are then adapted throughout the animation. The result pictures are rendered by parameterizing them 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

As already mentioned, the filter considers *line sketching* and *shading* separately. The following subsections shed some light on how both stages work in detail.

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 all upcoming 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 of objects. 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 big, if that pixel lies directly on a line in $G$ (edge) and if $\mathscr{L}_i$ is following this edge, such that only big values are collected in the convolution. If the pixel doesn't lie on or close to an edge 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. The blurriness comes from the fact that pixels close to edges collect some brightness from the lines that intersect the edge. See Section 6 for more information on that issue.

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 from 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.

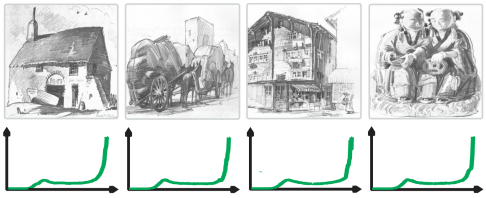

Figure 3: Examples for real pencil sketches and their measured tone distributions. Note: This image was taken from [LXJ12]

[LXJ12] used this observation to create a parametric histogram model for pencil drawings which consists of three functions representing 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 $\sigma_b$, which controls the sharpness of the function. This distribution can be seen on the left in Figure 4.

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 bell shaped peak in the lower 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 $\sigma_d$ and the position with $\mu_d$.

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

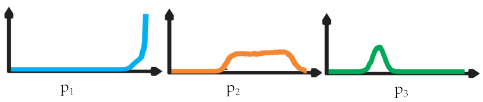


Figure 4: Plots of the three functions $p_i$. Note: picture taken from [LXJ12].

The final tone distribution is now simply composed 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 $\omega_i$ are weighting parameters which can be used to weight the importance of the individual Histogram parts.

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 then used to apply the tone distribution from Equation 6 to the input image. The result $J$ of the Histogram Matching is then used as input for the calculation of 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.

Figure 6: A possible men made pencil hatching pattern $H$.

**Texturing** The filter uses a men made pencil hatching pattern $H$ as shown in Figure 6. To create the correct shading in a pencil sketch, humans repetitively draw on the same position. 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 $H$ $\beta$ 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 this equation for $\beta$ is going to destroy the hatching pattern because $\beta$ 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 $\lambda$ can be used to determine how strong the hatching pattern $H$ will show through in the resulting image.

The hatching texture $T$ is then calculated as pixelwise exponentiation

$$T = H^\beta$$

Figure 7 shows the result for the minion picture.

Figure 7: 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 pixel wise:

$$R = L \cdot T$$

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 with another color space transformation.

In Figure 8 both the grayscale and the colored results are shown.

## 4 GPU Implementation

Figure 9 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 grayscale version of the image from the RGB to YUV result. The grayscale version is used for the gradient calculation, which is then fed into the convolution filter. This finalizes the line drawing step. In preparation

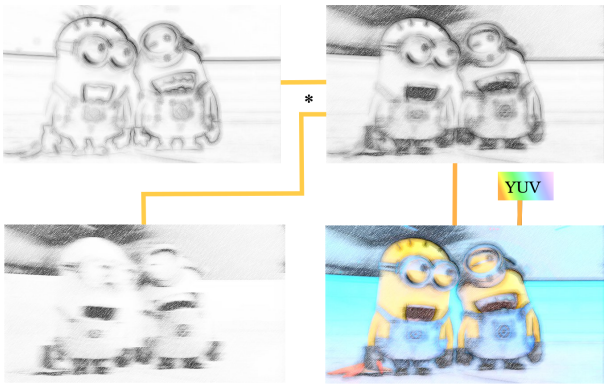


Figure 8: Combination of the line drawing and shading.

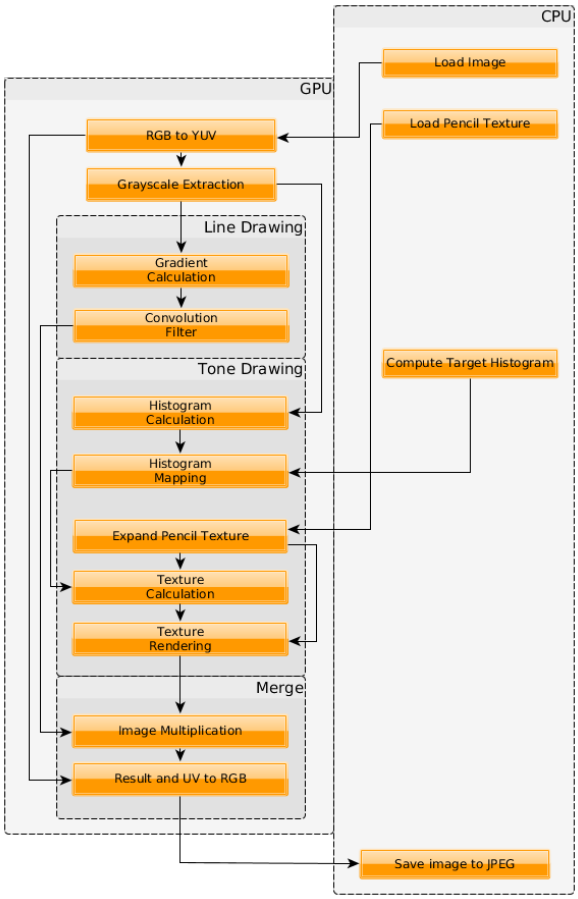


Figure 9: GPU and CPU pipeline of the implementation

for the shading or tone drawing step, the CPU is computing the target histogram using Equation 6. The histogram calculation in the tone drawing step receives the grayscale version as the input image. The result is then passed to the histogram mapping kernel, together with the CPU-computed target histogram. Yet Another kernel expands the hatching pattern image $H$ to image size. The expanded textures and the tonal texture $J$ are used for the texture calculation. The last kernel in the tone drawing section - the texture rendering kernel - computes the final hatching texture for the image. The last two kernels merge line drawing and hatching texture. One kernel combines the results of the line drawing and the tone drawing step and generates the final image, the last kernel adds back the colors if requested and converts the YUV back to RGB. The CPU takes care of saving the image to disk again.

### 4.1 Sketch Filter

Calculating the grayscale image $I_g$ and gradient $G$ on the GPU is no big challenge. However the efficient 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 then has to be selected (see Equation 2). The following paragraphs describe the implementation of the CudaKernel which computes $L$ directly from $G$ given the desired line length, strength and count.

**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 $i$. The rotation angle depends on the line count, as the angle between two lines is $\frac{360°}{\text{line count}}$. All line convolutions for one pixel are calculated by a single thread. The results for a

line is kept as maximum if it is bigger than all its predecessors.

Finally in 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 $\gamma$ 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 to calculate all convolution results. Therefore it makes sense to use shared memory to speed up memory access. Each pixel is calculated by a individual thread, and as we can only use 1024 threads in one block we get blocks of size $32 \times 32$, as shown in Figure 10

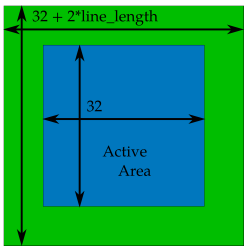


Figure 10: Block dimension (blue) and shared memory dimension (green)

On our device the maximum size of shared memory per block is $48kb$, so we can only store a $109 \times 109$ block of float values. Therefore only line lengths up to 45 pixels are allowed which is more than enough for the purpose of creating pencil sketches.

The data from $G$ is copied by assigning the thread number to the linear indexes of the data. Because there is more data elements than threads, one thread copies multiple data elements. The pattern is depicted in Figure 11. Figure 11.

The coordinate calculations for the line convolution computations are done such that they generate the correct coordinates inside the shared memory block. Due to the coordinate rotations the access patterns are very chaotic, which might lead to unavoidable bank conflicts.

### 4.2 Histogram

The histogram calculation calculates a partial histogram within multiple blocks. This is done in shared memory. Further more every block computes an additional cumulative histogram using a parallel prefix sum. The partial results are being combined at the end by 256 threads. They add the local results from shared memory in to global memory using `atomicAdd`. As our histograms have 256 bins the needed shared memory size per block is only two times 256 Integers, one for the simple histogram and one for the cumulative histogram.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | ... | | | | | | | | ... |
| | | | | | | | | | | | ... |
| | | | ... | $32^2$ -1 | $32^2$ | 1 | 2 | 3 | ... | | ... |
| | | | | | | | | | | | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

Figure 11: Thread number assignment when copying the data from $G$ to shared memory.

### 4.3 Histogram Matching

Histogram matching is used to adjust the tone distribution of the image $I_g$ to match a specific target tone distribution. In this context, the target tone distribution is artificially created using the model in Equation 6. This model creates a histogram which is similar to those of real pencil sketches as explained above.

The cumulative target histogram is created on the CPU by simply applying the parametric model to the values 0 to 255 and summing them up. 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 input image, which was 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 then obtain the tone value of the pixel it belongs to and look up it's cumulative probability value in the source histogram map. Afterwards, the cumulative target histogram is used inversely to find out which tone the value corresponds to in it. The pixel is then set to the value that was looked up. The lookup is done using bi-

nary search. One would think that a binary search is inefficient when multiple threads are executed in a warp and search for different values, but it really isn't. Because the cumulative 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. As 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 is 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:

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

$x$, in our case, is the solution image $\beta$ 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 $\Gamma$ is a diagonal matrix with two diagonals representing the gradient operator, if multiplied to a vectorized image.

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

To do so the input images $\ln(J)$ and $\ln(H)$ are downloaded from the GPU, fed into diagonal matrices from `cusp` and then uploaded again to solve the equation. After that the result is downloaded and uploaded again in our regular data representation for the final composition with the line image $L$.

Matrix $A*$ and vector $b^*$ (Equation 8) could be calculated using `cusp`. Therefore it's necessary to upload $\ln(H)$, $\ln(H)$ and $\Gamma$, 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 only those two objects, because `cusp` needs an output target matrix for each operation. So a lot of unnecessary memory would be needed 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 for $A^*$ in Figure 12. The closed form of $b^*$ is:

$$b^*(i) = \ln(J(i)) \cdot \ln(H(i))$$

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

Figure 12: 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 makes the filter approximately two times faster compared to the GPU method.

## 5 Performance

An important topic for the GPU implementation is of course the performance. The described algorithm was implemented in a program that loads an image from JPEG file, applies the filter to it, and saves it again to file. Profiling this program showed that loading the image, allocating the needed buffers, copying it to GPU, downloading it afterwards, and saving it as JPEG to disk take the most time by far.

As these are factors that aren't under our control, we chose to measure the time it takes to apply the implemented filter, and ignore the mentioned overhead in our measurements. This approach is legitimate, as in time critic applications the filter would be needed to be applied to multiple images. But this would also mean that loading the image and copying it to GPU could be done while the GPU is still busy with the previous image. Also the buffers can be reused, such that their initialization doesn't count for performance measurements.

We measured the time the GPU pipeline needs to execute by repeating the filter 100 times on

the same image to compensate for fluctuation. We chose to do our benchmarks with image resolutions known from common video formats: DVD/PAL (720x576), 720p (1280x720) and FullHD (1920x1080). The filter application to the DVD quality test image took 30 ms, the 720p version took 60ms and the FullHD image took 130ms which are all a lot faster then the original implementation [LXJ12]. Although the implementation is fast enough to be applied in real-time when watching a video in DVD quality, it doesn't make sense to use it as a video filter. The reason for this is that static textures are used for the shading effect which would be repeated in each video frame, such that the shading would look very unnatural.

To determine why our implementation takes so "long" for a FullHD image, we profiled its application to the test image to find bottlenecks. We found out that one step in the "hatch texture calculation" part takes nearly half the time for the whole filter application. The crucial part is to upload matrix $A^*$ to the GPU. If this `cusp` matrix could be directly created on the GPU from the data that is already there, the performance of the filter would roughly be doubled. Unfortunately, we didn't find a good solution to this problem in the scope of this project.

## 6 Future Work

In the scope of our project we didn't yet implement the entire pipeline of the original paper. The quality of our line sketch filter for example could be increased by getting rid of the blurriness, which would require one further processing step that is described in [LXJ12].

As mentioned before, it would be nice if there was a way to construct matrix $A^*$ and vector $b^*$ directly on the GPU with the data that is already there. This might be possible with the use of `cusp`s LinearOperator classes.

Currently the parameters of the filter all have to be set manually. It would be helpful to create some parameter presets, which can be selected and adapted on the fly. Those presets could be created by the user on the fly and saved to disk or learned from real pencil drawings as described in [LXJ12].

Another idea regarding the parameters would be a graphical user interface that immediately reapplies the filter, such that the user could immediately see the response to parameter changes.

If one could find a way to enforce temporal consistency in real time the filter could be used to stylize entire video sequences. It might be worth it to take a look at [LWA+12] for this purpose. With precalculated optical flow it might actually be possible to reach real time performance.

## 7 Conclusion

We used modern technologies for parallel computing on GPUs to implement an image filter to create pencil sketches from real images based on the methods of a published paper. In the course of this, common operations like color mode conversion, gradient calculation, convolutions, histogram creation, and histogram matching have been implemented as parallel working algorithms. Our benchmarks results show that the speed of the image filter would even be sufficient to apply it in real time for videos. This result is clearly an indicator for a successful and efficient GPU implementation of the image filter.

## 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.

[LWA+12] Manuel Lang, Oliver Wang, Tunc Aydin, Aljoscha Smolic, and Markus Gross. Practical temporal consistency for image-based graphics applications. *ACM Trans. Graph.*, 31(4):34:1–34:8, July 2012.

[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.