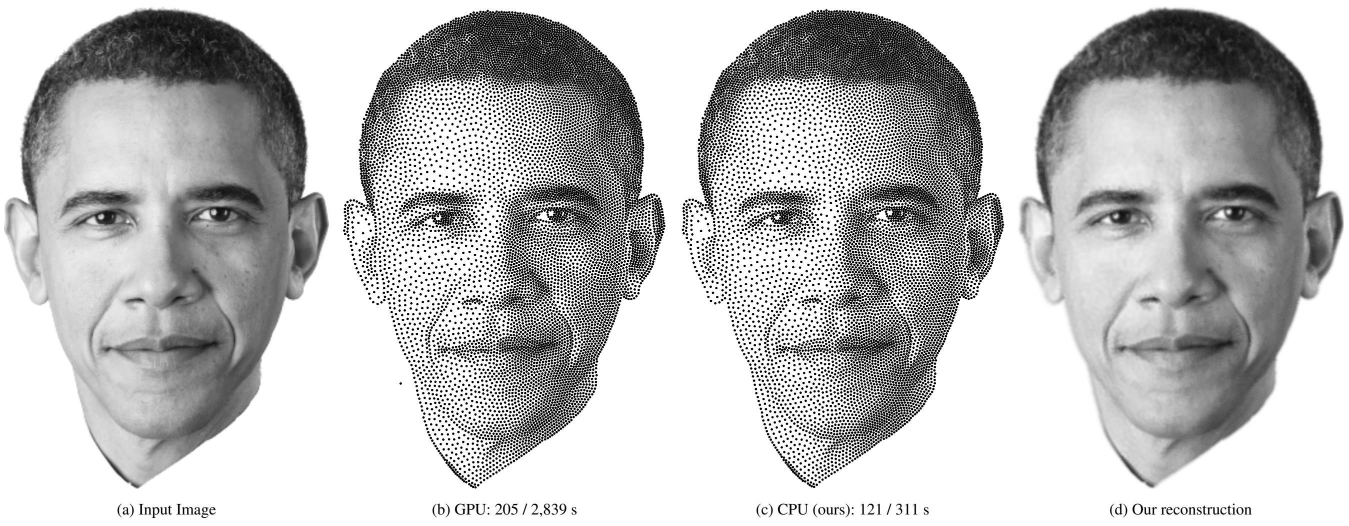
EG UK Computer Graphics & Visual Computing (2024)
A. Slingsby and D. Hunter (Editors)

Serial Gaussian-Blue-Noise Stippling

Abdalla G. M. Ahmed [†]

Khartoum, Sudan



(a) Input Image (b) GPU: 205 / 2,839 s (c) CPU (ours): 121 / 311 s (d) Our reconstruction

Figure 1: *Stippling of the input image in (a) using (b) the original GPU-based GBN algorithm of Ahmed et al. [ARW22] and (c) our CPU-based modified algorithm, both using 10K points and 10K iterations, comparing the quality and consumed time in seconds for float/double precision, respectively. In (d) we see a reconstruction of the image from the points in (c).*

Abstract

We adapt the adaptive Gaussian Blue Noise (GBN) algorithm to iterate serially over the points, one by one, thus enabling its implementation on CPU. Towards that end, we propose an alternative kernel shaping model. Our implementation model is simpler and has a linear time complexity, replacing the quadratic complexity of the original model.

1. Introduction

Stippling is a black-and-white reproduction technique for grayscale images. Unlike its close relative, halftoning, that uses a low-resolution grid of pixels, stippling allows arbitrary placement of the small dots, usually disc-shaped, that constitute the image, resulting in a far higher quality for the same count of picture elements.

Computer-generated stippling was introduced by Deussen et al. [DHVOS00] to imitate hand-crafted stipplings in artworks and illustrations. Many algorithms have been proposed since then. The Gaussian Blue Noise (GBN) algorithm of Ahmed et al. [ARW22] represents the current state of the art. Figure 1(b, c) demonstrates GBN stippling, along with a reconstructed image in (d). It offers a tone-reproduction quality comparable to the original grayscale input image for the same information bit budget. For example, the shown stipplings use only 62% of information bits compared to the input image resolution, and reproduce almost an identical image at a distant view. Noting the disjointedness and well-spacing of the stipple points, along with this visual reproduction quality, Ahmed [Ahm23] re-introduced stippling as an image encoding technique that is well-suited for printing/engraving grayscale images on different non-electronic media, including stones, leather, and wood, among others demonstrated in his poster.

The adaptive GBN algorithm presented by Ahmed et

† abdalla_gafar@hotmail.com

DOI: 10.2312/cgvc.20241226

al. [ARW22] is inherently parallel, updating all the points simultaneously in each iteration. While parallelization is usually taken as a pro, lack of serialization may arguably be considered a limiting factor of GBN, preventing its portability to CPU-only devices. In this paper, we present our serial-based implementation of GBN. To achieve that goal, we developed an alternative kernel-shaping algorithm that supports serial computation.

2. Background

In uniform GBN optimization, a Gaussian kernel

$$g(\mathbf{x}) = e^{-\frac{\|\mathbf{x}-\mathbf{x}_i\|^2}{2\sigma^2}} \quad (1)$$

is placed on each sample point, and the placement of the points is optimized so that the sum of the Gaussians is smooth and uniform all over the domain. For the adaptive case used in stippling, the idea is similar, but static negative kernels are placed on the pixels of the given density map, and the goal is to make the absolute sum of the pixel and sample point kernels close to zero all over the domain. There is, however, an important difference introduced in this case, adopting from Fattal [Fat11], that the kernels are shaped

$$g(\mathbf{x}) = ae^{-\frac{a\|\mathbf{x}-\mathbf{x}_i\|^2}{2\sigma^2}} \quad (2)$$

in accordance with the local density to cope with the effective local spatial frequency content, making them narrower and taller in darker parts containing more samples, and shorter and flatter in lighter parts with fewer samples. This calls for an algorithm to compute the shaping factor a for each kernel. Ahmed et al. [ARW22] use the logic in Algorithm 1. The intuition is to make each kernel maintain the same $1 : 2\pi\sigma^2$ peak-to-sum ratio as found in uniform distribution. We note, however, that this calls for an iterative process to recompute each kernel in accordance with the updated information of the others. This process is concealed in the authors' code by interleaving it with the gradient-descent location updating logic that computes the optimal placement of the points. The other concern is that this process might diverge, flattening all kernels, as we have actually observed a few times. This is solved by including the normalization at the end of Algorithm 1 to force all the kernels to stay within a nominal shape. It is this normalization that dictates parallelization in their original code, since all the relative peak-to-sum ratios must be available, and changing the shaping factor of any point affects all the other points.

3. Our Treatment

The first and most significant modification we introduce is a different algorithm for computing the shaping factors: we shape each kernel σ proportional to the nearest neighbor as summarized in Algorithm 2 Not only has the logic become far simpler, but we find that our modified model is far more intuitive. The key idea is that the nearest neighbor readily gives an estimate of the area covered by the point, and the kernel width simply expands or shrinks accordingly — while maintaining the same mass as suggested by Fattal [Fat11].

Algorithm 1: Original kernel shaping algorithm in GBN, reproduced from Ahmed et al. [ARW22].

```
Input  : A nominal kernel width $\sigma$ and list $\{\mathbf{x}_k\}_{k=1}^N$ of
         kernel centers.
Output: An optimized list $\{a_k\}_{k=1}^N$ of kernel amplitudes.
1 Initialize all amplitudes assuming a uniform density:
    $a_k \leftarrow 1$;
2 for I-iterations do
3     Compute accumulated density at each point:
        $d_k = \sum_{l \neq k} a_l \exp\left(-a_l \frac{\|\mathbf{x}_k-\mathbf{x}_l\|^2}{2\sigma^2}\right)$;
4     Set all amplitudes to respective densities: $a_k = d_k$;
5     Normalize $\{a_k\}$ so that they average to 1;
```

Algorithm 2: Our alternative kernel shaping algorithm.

```
Input  : A nominal kernel width $\sigma$ and list $\{\mathbf{x}_k\}_{k=1}^N$ of
         kernel centers.
Output: An optimized list $\{a_k\}_{k=1}^N$ of kernel amplitudes.
1 Compute nominal area per point $u \leftarrow$ "domain area" /
    "number of points";
2 foreach point $p_i$ do
3     Find distance $d_i$ to nearest neighbor point;
4     Set $a_i = u/d_i^2$;
```

3.1. Our Implementation

We tested this logic empirically, and it produced very similar results to the original algorithm, indicating the success of our model, and opening the door for our serial implementation. Algorithm 3 shows the main steps of our implementation.

Algorithm 3: Our serial implementation.

```
Input  : 1, A density map
         2. An initial list $\{\mathbf{x}_k\}_{k=1}^N$ of point locations.
Output: An optimized list of point locations in accordance
         with the density map.
1 Initialize a list $a$ of shaping factors using Algorithm 2;
2 Initialize a list $b$ of nearest neighbors of each point;
3 for I-iterations do
4     foreach point $p_i$ do
5         compute a new location as per adaptive GBN
            optimization [ARW22], and move the point;
6         Find the new nearest point of $p_i$, and update $a_i$ and
            possibly $b_i$ if changed;
7         Find if the moved point $p_i$ has come closer to
            another point $p_j$ than its nearest neighbor, and
            update $a_j$ for that point;
8         Find if $p_i$ was the nearest neighbor to another point
            $p_j$ and moved away, then search for nearest
            neighbor of $p_j$ and update $a_j$ and $b_j$;
```

3.2. Data Structures

We maintain a list of neighbors within the effective neighborhood of each point, beyond which the Gaussian tail is negligible, as decided by machine precision. Note that the mutual kernel shape between two points is the average of their kernel shapes; hence, a narrow kernel may interact with a very distant wider kernel. We maintain a simple array structure for this list and recompute the whole list if any point moves more than half the distance to its initial nearest point. Such a full list update is needed only 28 times in the 10K iterations of Figure 1(c), and the span between updates increases as the points continue to slow down with iterations.

Through this idea, we managed to confine the quadratic time complexity to the list updating, reducing the core algorithm to a linear time complexity, where a point typically maintains an average of 60/90 neighbors for float/double real point coordinates, respectively. The advantage of this list becomes more profound if we note that the list has to be iterated twice per point, once to compute the new location, and then to update the nearest neighbors and the associated shaping factors. At a cost of extra coding complexity, it is possible to maintain two lists, since the nearest neighbors stay within the first ring of Voronoi neighbors, and do not extend to 6 or 9 sigma as needed in position optimization.

4. Results

Here, we briefly discuss different aspects of our implementation.

4.1. Quality

We have not seen any noticeable quality difference compared to the latest code of Ahmed [Ahm23], cf. Figure 1(b, c), nor between the CPU and GPU implementation of our variant.

4.2. Speed

Comparing timing between CPU and GPU is rather tricky, since there are many parameters and models. For an abstract comparison of time complexity, our implementation maintains linear time complexity for the core computation, as mentioned above, and quadratic complexity for the less-frequent list updates, in contrast to the quadratic complexity throughout the GPU implementation. The numbers beneath Figure 1(b, c) show typical real-time performance on the same machine, namely a contemporary laptop with an Intel Core i7-10510U CPU and an NVIDIA GeForce MX250 GPU with 384 cores. Notably, the CPU implementation utilizes only one of the 8 available cores in this machine, which suggests that our implementation is possibly more economical for patch processing and/or low-end resources.

4.3. Memory

The space complexity of our implementation is linear in the number of points and is negligible for typical numbers of stippling points, especially since it is CPU-based.

4.4. Precision

One interesting observation is that double precision computation on GPU is significantly more costly than on CPU, as can be seen in Figure 1(b, c). Thus, our implementation enables feasible double precision computation, which becomes more significant as the number of points increases. We observed an improvement in quality with double precision computations compared to float, though at the cost of longer processing time.

4.5. Reconstruction

As with the original GBN, the kernel shaping algorithm is readily adaptable to a reconstruction algorithm. Our modified model offers a simpler, faster, and more stable reconstruction algorithm, since the shaping of each kernel can be computed independently of the other shaping factors. Figure 1(d) shows an example reconstruction of an image from the sample points in (c).

4.6. Coding Complexity

Our serial implementation is admittedly lengthier and more code-complex than the reference GPU implementation, as we have to maintain a few auxiliary tables. Increased coding complexity naturally leads to a higher likelihood of mistakes, and we actually spent considerable time using the incomplete algorithm before realizing the missing final step in Algorithm 3. Therefore, we have included our documented code in the supplementary materials to save readers time and help them avoid similar errors. In addition to the details discussed here, the code includes other documented facilities for initializing the point distributions, exporting the output, and video-recording the optimization process by pipelining to FFMPEG, for example.

On the positive side, our implementation model is more flexible and offers trade-offs between coding complexity and speed. Additionally, it may serve as valuable training material for students.

5. Conclusion

Through a different intuition of kernel shaping logic, we managed to make GBN stippling ready for serial CPU implementation, and we developed an actual optimized implementation that should help increase the accessibility of GBN-quality stippling, making it available to more low-cost computers and mobile devices.

References

[Ahm23] AHMED A. G. M.: Image printing on stones, wood, and more. In *ACM SIGGRAPH 2023 Posters* (2023), SIGGRAPH '23, ACM. doi:10.1145/3588028.3603686. 1, 3

[ARW22] AHMED A. G. M., REN J., WONKA P.: Gaussian Blue Noise. *ACM Trans. Graph. 41*, 6 (Nov. 2022). URL: https://doi.org/10.1145/3550454.3555519. 1, 2

[DHVOS00] DEUSSEN O., HILLER S., VAN OVERVELD C., STROTHOTTE T.: Floating Points: A Method for Computing Stipple Drawings. *Computer Graphics Forum* (2000). doi:10.1111/1467-8659.00396. 1

[Fat11] FATTAL R.: Blue-noise point sampling using kernel density model. *ACM Trans. Graph. 30*, 4 (2011). doi:10.1145/2010324.1964943. 2