Compiling the Kohonen Feature Map Into Computer Graphics Hardware

Florian Haar Christian-A. Bohn

Wedel University of Applied Sciences

Abstract

This work shows a novel kind of accelerating implementations of the Kohonen feature map algorithm. The algorithm is adapted to match the functional features of recent general purpose graphics processing hardware so that the newest developments in graphics hardware design can be utilized to run this neural network with a formidable speedup of up to 300%.

Keyword: Computer Graphics, Artificial Neural Networks, Kohonen Feature Map, Unsupervised Learning, GPU Programming, Computer Graphics Hardware

1 Introduction

1.1 Computer Graphics Hardware and Artificial Neural Networks

The non-deterministic nature of usual artificial neural network (ANN) algorithms often leads to excessive execution times even for virtually simple problems. As long as alternate algorithms with lower complexity are not known, in some cases it helps to have a hardware implementation available and to take advantage just from shorter execution times than a general purpose CPU would be capable of. The problem here is that — due to the relatively few applications — there are only few hardware boards available.
Moreover these mostly do not reflect the current state-of-the-art in hardware design.
In contrast — in the computer graphics (CG) area there are millions of users with "need" for graphically elaborated games and the associated computer graphics hardware.
This market drives CG industry to unprecedented short development cycles each of them resulting in a tremendous increase of graphics performance and visualization quality.

While early graphics boards were competent only of managing specialized pixel oriented functions — i.e. filling rectangular areas in display memory — today's graphics processing unit (GPU) function sets have nearly the same functional features as general CPUs. Moreover, they are highly integrated, extremely fast, massively parallel, and are mostly inherited from latest hardware design technologies, and — due to the huge manufacturing quantities — they are cheap.
It is obvious to try to utilize these graphics boards for running algorithms which are not directly connected to image generation. Work which already was accomplished includes the ray tracing approach [1], physical simulation [2], even more graphics unrelated applications like linear algebra [3], robot motion planning [4], cryptography [5], and also neural networks [6].

Overview

This work presents several possibilities to suit the Kohonen feature map (KFM) [7] into graphics hardware giving a significant competitive edge compared to the execution on general purpose hardware. In

the following, we explain our favourable algorithm in detail and refer to alternate implementations we have evaluated. Then we present a performance analysis and discuss results related to the prospective development of graphics hardware.

2 Kohonen Feature Map through Graphics Commands

2.1 The Kohonen Feature Map

The Kohonen feature map is one the most prominent ANN algorithms, maybe because it exposes robust learning characteristics, a fascinating and efficient principle of self organization, and an intuitive kind of visualizing a neural network's training results. In recent decades this resulted in a vast number of applications and thousands of publications.
The Kohonen feature map can be seen as set of n-dimensional reference units (vectors).
A network training loop consists of iteratively presenting n-dimensional sample input vectors to the reference units. Each time the best matching unit (BMU) (the reference unit which is most similar according to a certain distance measure) is modified in a way that it slightly adapts to the input — it is moved in n-dimensional space into the direction of the input. The KFM is organized in such a way that it finally represents the input sample distribution through the positions of its reference cells in n-dimensional space. Now the set can be used for clustering, vector quantization and also dimensionality reduction tasks.
Up to now, the approach is similar to general k-means clustering but the KFM offers another great feature by the fact that reference vectors are organized as a regular map. This creates an additional neighbourhood relation between reference cells and each time the training loop adapts a BMU, also its neighbourhood is adapted by a certain amount.
Thus, the virtually arbitrary map topology is considered additionally while learning, such that the learning sample set can be seen as projected on to the two-dimensional topology. In other words, the KFM approach is capable of recognizing clusters and inheriting a hidden topological information from the learning sample set.
A typical application of the KFM is the clustering of vectors of Fourier coefficients of naturally spoken words in certain time steps. Each resulting "phoneme" then represents a position on the map, and a whole spoken word generates a curve on the map. This curve graphically characterizes a spoken word appropriable in speech recognition applications [8].
Altogether the Kohonen feature map algorithm can be described by a few mathematical terms as follows.
Given a set of m reference units $\vec{c}_k \in \mathrm{R}^n$, k = 1..m, the BMU $\vec{c}_b$ concerning an input training vector $\vec{x} \in \mathrm{R}^n$ is found by

$$\vec{c}_b = \operatorname{argmin} \| \vec{x} - \vec{c}_k \| \ \forall \ k = 1..m.$$

After the BMU is known training of the network is accomplished by

$$\vec{c}_k = \vec{c}_k + \alpha \cdot h_{bk} \cdot (\vec{x} - \vec{c}_k) \ \forall \ k = 1..m \quad (2.1)$$

with α a learning parameter which decreases during network training and h_{bk} a neighbourhood relation concerning the 2D map coordinates $\vec{r}_k \in \mathrm{R}^2$ of a reference unit $\vec{c}_k$ defined like $h_{bk} =$

$$\begin{cases} 1 & if \quad \vec{r}_{b,x} - \vec{r}_{k,x} \le \eta \wedge \vec{r}_{b,y} - \vec{r}_{k,y} \le \eta \\ 0 & else \end{cases}$$

with η defining the size of the neighbourhood. η decreases during learning finally reaching a value of one for a neighbourhood set containing only the direct neighbours of the BMU.

2.2 Principles of Computer Graphics Hardware

"Graphics hardware is an efficient processor of images" [9] — and more generally, graphics hardware efficiently processes several streams of source to a stream of destination pixels from sets of images. The application program selects the source and destination images and the

generation of positions on these images from which the pixel streams are read.
Creating these pixel coordinates is accomplished by mapping (pixel coordinate interpolation) in between geometrical entities (usually triangles).
Roughly speaking, in this approach Fragment Shader Programs [10] define the combination function (mathematical operator) of source (first operand) and destination (second operand) pixel arrays (arrays of reference units).

2.3 Adapting KFM to CG Hardware

The KFM is stored in a RGBA texture map (neural texture map (NTM)) to allow the GPU fast access to the net units. Figure 1 describes the organisation of this map.

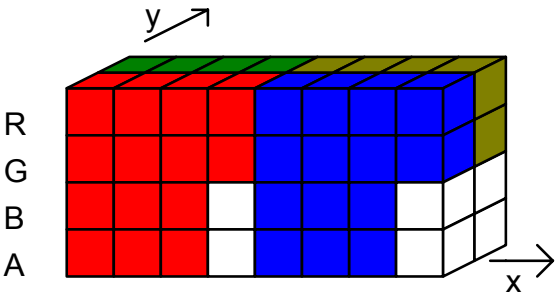

Figure 1: Excerpt from the neural texture map showing four 14 dimensional units. Each unit with its own color distributed over the map's texels. White texel components are filled with zero and do not belong to a unit. The unit's components are arranged in ascending order RGBA-RGBA-RGBA-RG.

Every n-dimensional unit $U=(\mu_1, \mu_2,\ldots,\mu_n)$ occupies $T_{perUnit} = n / 4$ rounded horizontally aligned texels. The unit's components are assigned to a texel $T_i = (\mu_{i*4}, \mu_{i*4+1}, \mu_{i*4+2}, \mu_{i*4+3})$,
with $i = 0, \ldots, T_{PerUnit}-1$.
Empty components arising from unit dimensions which are not multiples of 4 are ignored.

The unit's componental values are restricted to the value range [0..1]. The precision of the value range is based upon the number of bytes per color component.

The input units are stored in a texture map (input texture map (ITM)) to support fast access and for blending purposes. The algorithm generates an ITM for every incoming input unit. The ITM organisation is identical to the NTM organisation except for the fact that all ITM units contain the same values, due to the generation of one ITM for every input unit.

Every described algorithm in this paper is based on an orthographic projection where one logical unit is equivalent to one screen pixel.

The horizontal and vertical dimension concerning the map topology are specified R_x und R_y

2.3.1 Feeding the GPU

If the input units are known before training the KFM, the ITMs can be created in the first place. Otherwise the ITMs must be created at run time. This paper acts on the assumption that the ITMs are created at run time.
We use the render to texture approach because this is the most efficient kind of filling a texture on the local GPU memory [12].

Instead of drawing every input unit's pixel one after another needing $R_x * R_y * T_{PerUnit}$ drawing steps, we draw one component to all units and mask out the non-concerning texel positions through using the stencil buffer technique.

The ITM is generated at runtime drawing $T_{PerUnit}$ rectangles of size $R_x * R_y$ with activated stencil test. The rectangle's color C_i depends on the input unit's components $r = [\lambda_0, \lambda_1, \ldots , \lambda_n]^T$ and can be displayed as $C_i=[\lambda_{i*4}, \lambda_{i*4+1}, \lambda_{i*4+2}, \lambda_{i*4+3}]^T$, with $i = 0, .., T_{PerUnit}-1$. The stencil test must be updated for every rectangle to accept only values equal to i (see figure 3). This allows drawing the component's color only at the associated position in the ITM.

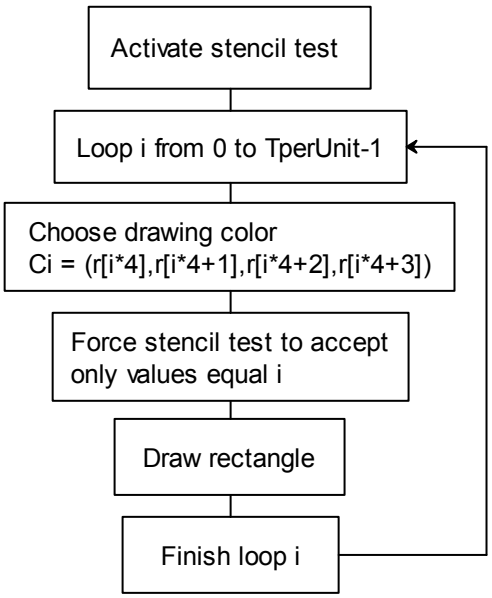

Figure 3: Flow chart of the "Generating the ITM" algorithm. The ITM is generated by drawing $T_{perUnit}$ rectangles with activated stencil test. The rectangle's color depends on the actually drawn input unit's components. Colors are only drawn if the actually drawn component i equals the stencil test values.

2.3.2 Determining the difference between Input and Net Units

Since we are only interested in the comparison of Euclidean distances, we can alternatively compare the sum of the absolute differences of the single unit components avoiding costly calculation of the square and square root.
The sum over a whole vector is calculated by first accumulating the differences over one pixel, i.e. 4 vector components, and then taking the sum of all intermediate results.

Is $\vec{x}$ the input unit and $\vec{r}_{jk}$ the net unit at the position (j, k) in the KFM the Euclidean distance D_{jk} for that unit is calculated:

$$D_{jk}(x, r_{jk}) = \sum_{i=0}^{TperUnit} E_V(x_i, (r_{jk})_i),$$

with $x_i = [\lambda_{i*4}, \lambda_{i*4+1}, \lambda_{i*4+2}, \lambda_{i*4+3}]^T$,

$r_i = [\alpha_{i*4}, \alpha_{i*4+1}, \alpha_{i*4+2}, \alpha_{i*4+3}]^T$,

$j = 0, \ldots, R\_x$ and $k = 0, \ldots, R\_y$.

The following algorithm is executed in a fragment shader (see figure 4). By drawing a rectangle with dimension $R_x * R_y$ the fragment shader is called once for every unit. One execution of the shader calculates the distance for each texel/pixel component and accumulates them locally.
Figure 5 demonstrates the association between the pixels and the NTM units after executing the fragment shader algorithm.

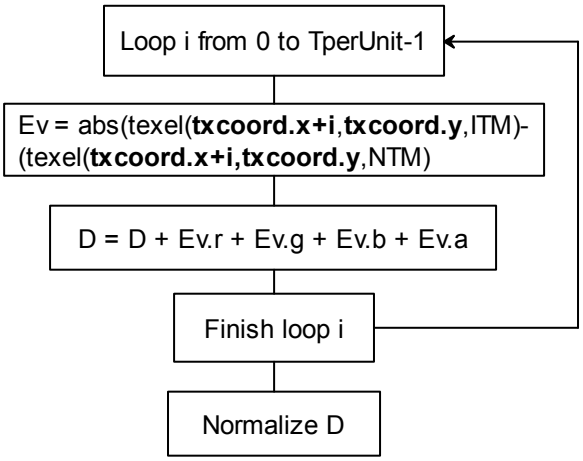

Figure 4: Flow chart of the difference-determining algorithm. The algorithm is executed in a fragment shader. Looping through all pixels of a unit admits the calculating of the separated Euclidean distances. Normalizing D is necessary to stay inside the pixel buffer's values range.

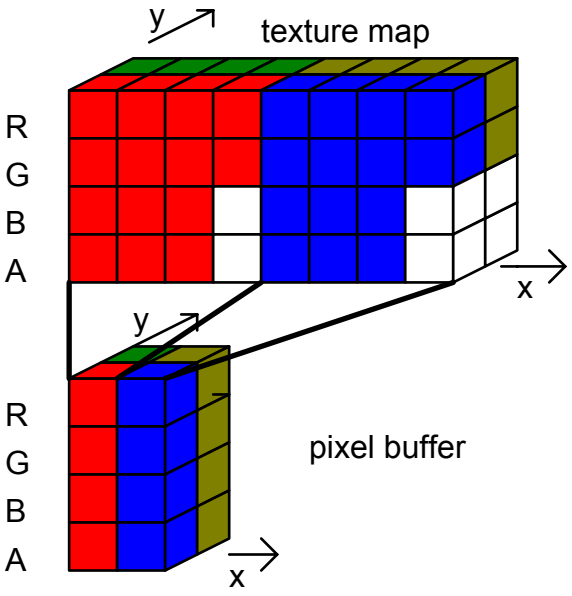

Figure 5: The distance for every unit is saved in one pixel in the pixel buffer after executing the above discussed algorithm.

Alternative solutions. We tested alternative algorithms using blending or the accumulation buffer to subtract the differences and a fragment shader to compute the sum of all differences. Further each algorithm was combined with an accumulation buffer to compute the sum within a CPU loop and not within the fragment shader; normalizing the vectors after each iteration step using the accumulation buffer.

2.3.3 Finding the Best Matching Unit

After calculating the distances between the input and net units it is necessary to find the position of the unit with the smallest distance to the input.
The fastest way to find the minimum value is to write $R_x * R_y$ vertices at the same x,y-coordinate with the distance as negative z-coordinate. Enabling depth test forces the GPU to sort the incoming vertices by their z-coordinate.
Additionally the color of each vertex is set to the position of the actually considered distance in the KFM. So after drawing all vertices the pixel's red and green color components at the previous used x,y-coordinate contain the x- and y-position of the BMU in the KFM. Eventually it is necessary to adjust the value's range of the considered distance position to the color component's value range. Figure 6 shows a flow chart of this algorithm.

Before drawing the vertices the computed distances must be read from the pixel buffer into CPU memory.

Alternative solutions. The biggest problem of this algorithm is the performance loss due to the copy procedure. An alternative solution would be to bind the pixel buffer to a texture; then using a vertex shader to read and draw the distances from the texture. This solution has not been tested because this paper is based on the nVidia Cg 2.0 language which does not support texture access for vertex shader.

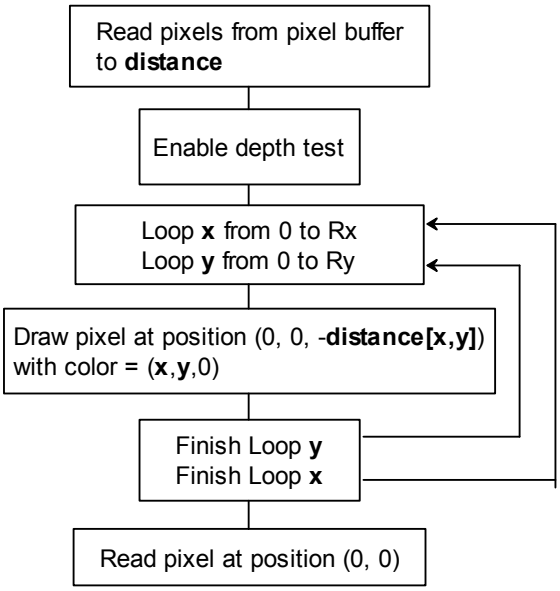

Figure 6: Flow chart of the "Finding the BMU" algorithm. After reading the pixel buffer into CPU memory (distance) a vertex for every distance value is drawn with activated depth test. By adding the considered distance position in the KFM (x,y) as color, the pixel read at the end of the algorithm contains the BMU's position in the KFM in the red and green color components.

2.3.4 Training the Map

Training a unit is achieved through applying formula 2.1 to every texel T_i^{NTM} of the NTM and T_i^{ITM} of the ITM, with i = 0, .., $T_{PerNeuron}$:

$$T_i^{NTM}(t+1)=T_i^{NTM}(t)+\alpha(t) \cdot h_{bk}(t) \cdot [T_i^{ITM}(t)-T_i^{NTM}(t)]$$

Since every T_i^* on the right side of the equation is equivalent with a texture access we convert the formula to reduce texture accesses:

$$T_i^{NTM}(t+1)=$$
$$(1-\alpha(t) \cdot h_{bk}(t))\, T_i^{NTM}(t) + \alpha(t) \cdot h_{bk}(t) \cdot T_i^{ITM}(t).$$

The training is accomplished by texture blending, followed by copying the blended pixel within the neighbourhood into the NTM (see figure 7).
Multiplying the two texture maps with the factors $(1-\alpha(t) \cdot h_{bk}(t))$ and $\alpha(t) \cdot h_{bk}(t)$ happens during drawing, combining the textures' colors multiplicative with the polygon surface color, which has to be set to $(1-\alpha(t)) \cdot h_{bk}(t))$ or $\alpha(t) \cdot h_{bk}(t)$ respectively.

Using blending with an additive blend function and blend factors of one for both texture maps completes the training of the map.
Afterwards the neighbourhood region is copied from the pixel buffer to the same position into the NTM.

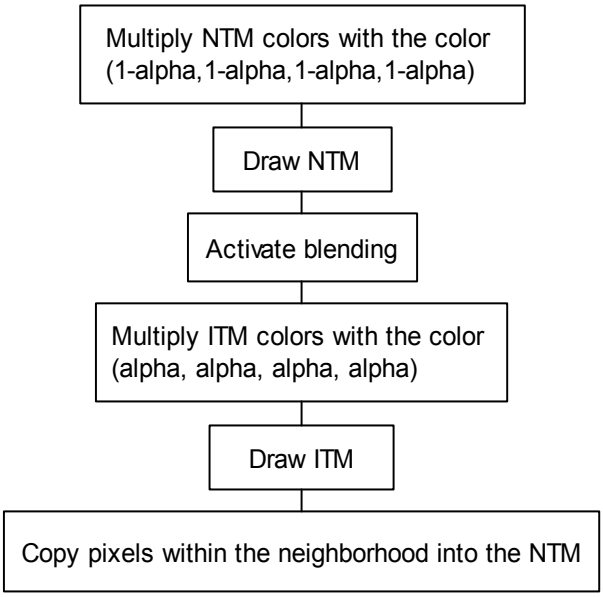

Figure 7: Flow chart of the „Training the Map“ algorithm. Multiplication of the textures' colors with the factors happens during drawing. An additive blend function and blend factors of one are applied to both textures.

Alternative solutions. Training of the map can also be solved using the accumulation buffer or fragment shader. Additionally we have tried to train only the neighbourhood by calculating its texture offset with the CPU prior computing the algorithms.

3. Results

3.1 Performance

We compare execution times for "Determining the difference between Input and Net Units", "Finding the Best Matching Unit" and "Training the map" to alternate solutions mentioned in this work including a pure CPU implementation of the algorithm.

The execution times are levied for 1500 iteration steps for 10*10 40 dimensional units. The algorithms were executed on a Pentium4 2Ghz with nVidia GeForce 6800 and Windows XP. Measurements are accomplished several times and are mean values of the output of the ANSI-C clock() function.
The algorithm names used within the diagrams provide information about the used GPU features (shader = fragment shader, accum = accumulation buffer, blend = blending). A '+' between algorithms indicates serial usage, a '|' parallel usage.

Determining the difference between Input and Net Units

Figure 8 exposes the times for calculating the difference between vectors and it exposes that the pure shader based algorithm clearly shows the best results.

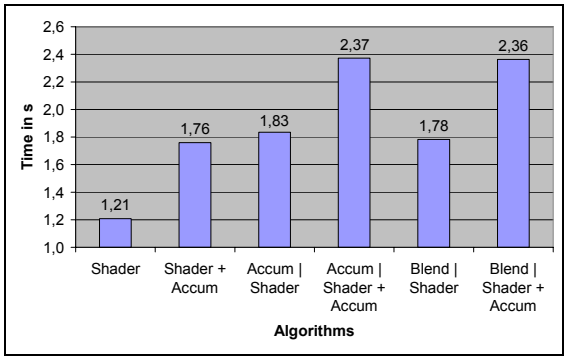

Figure 8: Required times for 1500 iteration steps determining the differences between input units and net units for a KFM with 10*10 40 dimensional units sorted by algorithm. The algorithms' order stays the same for other KFM measures.

The ranking of the algorithms stays the same for any size of the KFMs.
Because of the loop which calculates the sum of the components of each unit, the algorithm becomes slower for bigger units than for more units while keeping the required texture size the same.

The bigger the KFM the smaller the difference between the shader algorithms (shader, shader+accum) and the accumulation buffer and blending algorithms. But the pure shader implementation's speed is not reached for reasonable KFM measures.

Finding the Best Matching Unit

Figure 9 illustrates the required times for finding the BMU. The ranking of the algorithms stays the same for smaller KFMs with the blending algorithm always as fastest algorithm.
These algorithms are also faster when calculated for more units with less dimensions than the other way round expecting the same NTM measure. This happens due to the fewer number of pixels which must be copied into the NTM after adjusting the net units. With a 3x3 neighbourhood a NTM with $T_{PerUnit}$=1 has to update 3*3=9 pixels, while a NTM with $T_{PerUnit}$=2 has to update 3*3*2=18 pixels.

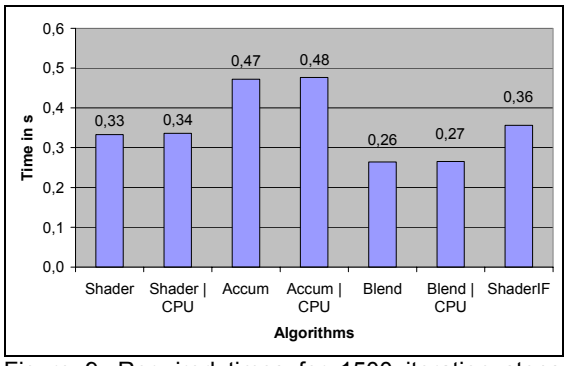

Figure 9: Required times for 1500 iteration steps finding the BMU for a KFM with 10*10 40 dimensional units sorted by algorithm. The algorithms' order stays the same for other KFM measures.

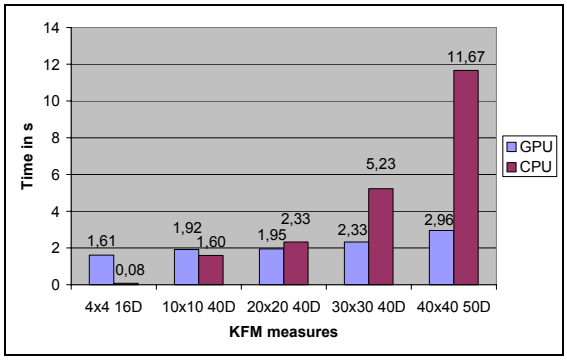

Figure 10: Required times for 1500 iteration steps performing the complete Kohonen algorithm for four different KFM measures using the discussed GPU algorithm and a fast CPU algorithm.

Finally we compare the discussed GPU algorithm with a fast CPU algorithm. Like Figure 10 shows the CPU algorithm is much faster if calculating unusual small KFMs than the GPU algorithm, but the difference shrinks with increasing KFM size. For 20*20 units with 40 components our approach clearly outperforms the CPU implementation, and this proves the idea of our approach.
Generally we can state that the GPU-algorithm runs significantly faster than the usual CPU implementation. Only for cases where the networks become unusual small the communication between GPU and CPU takes the main part of the computation and thus performance shrinks.

4 Summary and Future Work

We presented a novel way for implementing the Kohonen Feature Map algorithm on a common computer graphics acceleration board. Our main idea is to profit from a massively parallel hardware design which CG boards share with the characteristics of the KFM algorithm.

Our results clearly proved our idea. We reached a speed-up of about 300% for a 40x40 KFM with 40-dimensional reference units.

We showed that the performance of our approach significantly depends on the network size. While uncommon small networks are not capable of profiting from using the GPU, normal sized networks run notable faster, and the further increasing network sizes leads to even greater acceleration factors.

Future Work

With a typical development cycle of six months for graphics cards, which mostly leads to completely new functional extensions of actual hardware, the potential of future work is obviously huge.
Our main goal is to reduce the accesses between GPU and CPU as far as possible by migrating the algorithm completely onto the graphics card.

This might be possible soon, since several innovations are already announced by the industry (i.e. “accessing the texture from the vertex shader”).

References

[1] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In SIGGRAPH ’02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, pages 703–712. ACM Press, 2002.

[2] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In HWWS ’03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 92–101. Eurographics Association, 2003.

[3] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. ACM Trans. Graph., 22(3):908–916, 2003.

[4] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In SIGGRAPH ’90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques, pages 327–335. ACM Press, 1990.

[5] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. Pixelflow: the realization. In HWWS ’97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pages 57–68. ACM Press, 1997.

[6] Christian-A. Bohn. Kohonen feature mapping through graphics hardware. In Paul P. Wang, editor, Proc. JCIS’98, volume II, pages 64–67. Association for Intelligent Machinery, Inc, 1998.

[7] T. Kohonen. Self-organized formation of topologically correct feature maps. In J. W. Shavlik and T. G. Dietterich, editors, Readings in Machine Learning, pages 326–336. Kaufmann, San Mateo, CA, 1990.

[8] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. ACM Comput. Surv., 31(3):264–323, 1999.

[9] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In HWWS ’02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 109–118. Eurographics Association, 2002.

[10] Randima Fernando, Mark J. Kilgard. The Cg Tutorial. Addison-Wesley, 2003.

[11] T. Kohonen. Self-Organizing Maps. Springer, 1997.

[12] Chris Wynn. OpenGL Render-to-Texture. nVidia Paper. www.developer.nvidia.com/attach/6725.