Kohonen Feature Mapping through Graphics Hardware

Christian-A. Bohn

Department of Visualization and Media Systems Design
German National Research Center for Information Technology
Sankt Augustin, Germany
http://viswiz.gmd.de/bohn

Abstract

This work describes the utilization of the inherent parallelism of commonly available hardware graphics accelerators for the realization of the Kohonen feature map. The result is an essential reduction of computing time compared to standard software implementations.

Keywords. Kohonen feature map, computer graphics, hardware, *OpenGL* , frame buffer.

1 Introduction

The *Kohonen feature map* (KFM) [3] is a particular kind of an *artificial neural network* (ANN) model, which consists of one layer of n-dimensional *units* (*neurons*). They are fully connected with the network input. Additionally, there exist lateral connections through which a topological structure is imposed. For the standard model, the topology is a regular two-dimensional map instantiated by connections between each unit and its direct neighbors.

The KFM is used for *unsupervised learning* tasks [2]. Through n-dimensional training samples, the units organize in a way that they match the distribution of samples in their n-dimensional input space — *reference units* are placed which can be seen like *representatives* for particular regions. Sample agglomerations are designated as *clusters* and the according reference units are suitable for *classification* tasks. Through embedding the reference units into a topology, the KFM offers outstanding facilities for the structural analysis of data of arbitrary dimension (*dimensionality reduction* [2]). Nowadays, the KFM is one of the most applied neural network models.

The inherent parallel nature of the KFM has led to the basic idea of this work of utilizing the parallel architecture of commonly available computer graphics hardware accelerators. The efficiency of graphics hardware comes from the fact that it is commonly detached from the main computing unit. It "lives" in its own environment and processes its own fast memory, whereas the main application software runs independently on a general purpose CPU connected with the computer's main memory. This separation allows for an efficient development of both parts independently.

This work adapts the general KFM algorithm to be executed by graphics library function calls which are commonly implemented directly as part of the hardware. Each location in the graphics memory (*frame buffer*) stands for a point (*pixel*) on the computer screen. Processing is characterized by, on one hand, memory management functions like transferring or initializing memory blocks, and, on the other hand, by complex functions like drawing two-dimensional geometric objects. Due to the demands for more advanced graphics features, in the last decade, the principle functionality of graphics hardware has widely been extended in a way that it is also suitable to run algorithms which do not come directly from this field.

This work compiles the KFM scheme, but besides, it should generally initiate thinking of applying graphics hardware to non-graphics applications, since actually, graphics acceleration techniques are obviously the main focus of common hardware development. Moreover, graphics hardware is usually available on the most computer systems.

2 Kohonen feature map

Consider a set $\mathcal{A}$ of k units c_i, $i = 1 \dots k$ with a weight vector $\mathbf{w}_i \in \mathbb{R}^n$ attached to each of them. The units are organized in a two-dimensional regular map. The coordinates of the c_i concerning this map are denoted as vectors $\mathbf{r}_i \in \mathbb{R}^2$.

Given an input sample $\xi \in \mathbb{R}^n$, learning is accomplished by a two-step process.

1. Search for the *best matching unit* (BMU) c_b with a weight vector $\mathbf{w}_b$ which is most similar to the input ξ, i.e., which has the Euclidian distance $d = \|\xi - \mathbf{w}_b\|$, such that

$$\|\mathbf{w}_b - \xi\| \leq \|\mathbf{w}_i - \xi\|, \forall c_i \in \mathcal{A} \qquad (1)$$

   holds.

2. Move all units $c_i \in \mathcal{A}$ regarding a distance function *of the lateral connections* $\Omega_\Delta : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$, $\Omega_\Delta(\mathbf{r}_b, \mathbf{r}_i) = h(\|\mathbf{r}_b - \mathbf{r}_i\|)$ according to

$$\mathbf{w}_i{}^{\text{new}} = \mathbf{w}_i{}^{\text{old}} - \epsilon \cdot \Omega_\Delta(\mathbf{r}_b, \mathbf{r}_i) \cdot (\mathbf{w}_i{}^{\text{old}} - \xi). \quad (2)$$

   $h : \mathbb{R} \to \mathbb{R}$ is commonly chosen a Gaussian radial basis function with an expansion parameter Δ. The BMU attracts the surrounding units according to the definition of Ω_Δ and their distances. The learning strength ϵ and Δ are successively diminished during the iteration process, such that the training effects become smaller, and finally, the network converges to a fixed topology.

3 Realization

3.1 Graphics memory structure

The screen pixels are usually organized in a two-dimensional coordinate system on which the graphics library function calls are based. This regular structure is adopted as the KFM topology — each point on the screen is related to a certain unit c_i and its screen coordinates are taken as the unit's position $\mathbf{r}_i$ on the map.

Each pixel is instantiated by a vector of four components regarding the colors red, green, and blue, and a transparency component named *alpha value*. Whereas the resulting pixel color is composed (mentally) from the color channels, the alpha component is used by the graphics hardware for transparency effects only. For the purposes of this work, the library interprets the four components consistently like a four-dimensional vector. Each pixel of the screen is defined being one of the weight vectors $\mathbf{w}_i \in \mathbb{R}^4$ of a unit c_i at the map position $\mathbf{r}_i$.

Consider some example graphics operations and their effects concerning the KFM interpretation. Opening a screen window of 1000×1000 pixels, concurrently allocates a four-dimensional KFM with one million of units. Clearing the screen, sets the components of all vectors $\mathbf{w}_i \in \mathcal{A}$ to zero. Drawing a horizontal red line on top of the screen sets the first components $w_{j,0}$ of those $\mathbf{w}_j : \; j = 1 \ldots k$ to the value 1, whose second component $r_{j,1}$ of their map position $\mathbf{r}_j$ equals 1 (the top line contains those units whose y-coordinate equals 1).

3.2 Graphics library

Programming the graphics hardware is very similar to writing code in assembly language. Graphics commands are function calls which operate on the screen memory and on few internal stacks. In the following, first, some principal facilities of the used graphics library — *OpenGL*® — are outlined, then, the KFM algorithm from section 2 is translated to *OpenGL* calls.[1]

Using the widely spread *OpenGL* guarantees that as many hardware facilities as possible are utilized, and that the program can be implemented on almost all recent computer systems. Nevertheless, some (yet) *SiliconGraphics* specific commands are used (ending with `EXT`), but these features will probably be added to the standard of *OpenGL* in the near future.

Blending

The *OpenGL* basic drawing scheme can be generalized by moving pixel data from a *source* to a *destination*. Here, sources may be a constant four-dimensional *drawing color*, or even complete rectangular regions of the screen memory.

Instead of just overwriting the destination with the source pixel, *OpenGL* provides the generalized *blending* facility which allows for operations like weighted summation or multiplication of the components before setting the destination pixel. From the graphical point of view, blending denotes the softening up of two color values — transparency effects can be mimicked in that way. Blending is determined by the library call `glBlendFunc` with two arguments defining the term which is multiplied with the source and the destination pixel, respectively.

Color matrix

While blending allows only for the combination of the four pixel components *separately*, the applica-

[1] To keep this work comprehensive, it is focused on a small subset of *OpenGL* features. See [5] for further information. Even explanations concerning the graphics hardware architecture (section 3.1) are limited to the needs of this work.

tion of the *color matrix* provides the capability of combining single color components between each other. Each time before a pixel (vector $\in \mathbb{R}^4$) is written to the destination, it is multiplied with the 4×4 color matrix. Thus, exchanging components, as well as a weighted summation between the components are possible.

3.3 Translation

Some general operations needed at several places in the algorithm from section 2 are described. Their principle functionality is explained — detailed comments concerning the implementation like, for example, the storage of intermediate results in the frame buffer, are omitted.

Loading the frame buffer

As few data exchange as possible should arise between the main memory and the screen buffer. Thus, once, the KFM vectors are created in the screen memory, the whole algorithm is executed "on the screen", before the results are transferred back into the main memory. `glReadPixels` and `glDrawPixels` are the according *OpenGL* function calls (see [5, 4] for further descriptions).

Subtraction

Given a particular region in the frame buffer, subtraction of a constant is accomplished by enabling blending, and then, drawing a geometrical object on the frame buffer, or transferring buffer data. Blending arguments should be `(GL_ZERO, GL_ONE_MINUS_CONSTANT_COLOR_EXT)` or `(GL_ZERO, GL_ONE_MINUS_SRC_COLOR)`, respectively. Subtraction of a constant is required in equations (1) and (2) for subtracting the input sample from the actual unit weight vectors. Subtraction of a frame buffer segment is needed in equation (2) to calculate the resulting $\mathbf{w}_i^{\text{new}}$. It is executed by the routine `glCopyPixels` which copies frame buffer data, pixel by pixel, from one screen location to a second.

Multiplication

Multiplication by a constant is realized through the blending arguments `(GL_ZERO, GL_SRC_COLOR)`, and it is needed for the multiplication of the weight matrix with ϵ in equation (2). For the multiplication of two memory buffer regions, again, `glCopyPixels` is applied. It is required for the quadrature of all pixel components (eq. 3) in equation (1) for calculating the Euclidian distance. In this case, source and destination must be identical memory locations.

Searching the BMU

Searching the BMU is equivalent to looking for the c_i with the smallest squared distance

$$d^2 = \sum_{j=1}^{4} (w_{i,j} - \xi_j)^2, \ \forall i = 1 \ldots k. \tag{3}$$

Given the map of units c_i the related d^2 are calculated by subtraction of the constant vector ξ and a summation of the resulting components. By setting the first line of the color matrix to 1, the components' sum is written into the red (first) channel at the destination. Then, the determination of the biggest red-component of all pixels is accomplished by the *OpenGL* function `glMinmaxEXT`. Unfortunately, `glMinmaxEXT` does not deliver the coordinates ($\mathbf{r}_i$) of the maximum pixel, but only its value. Thus, the following algorithm of subsequent calls of `glMinmaxEXT` is necessary.

1. Calculate and store the maximum value by one application of `glMinmaxEXT`.
2. Split the screen into two partitions and determine (using `glMinmaxEXT`) which one contains the stored result from step 1.
3. If the detected partition contains only one pixel, then the position of the BMU is found, otherwise, set the search screen to the detected partition and go to step 2.

The position of the BMU is found recursively after $\log_2 m$ calls of `glMinmaxEXT`, with m the number of pixels of a screen edge, i.e., the width of the KFM (assuming a square map).

Convolution

One operation is left — the multiplication of the KFM matrix with the function Ω_Δ in equation (2). Suitable for this task is an *OpenGL* facility, called *texturing*. Texturing maps a two-dimensional image of texture values (*texels*) — contained in the texture memory which is an additional buffer similar to the screen buffer, and which can also be taken as source or destination for drawing pixels — into the screen memory.

A function table of h (eq. 2) is pre-calculated and stored as texture. By placing it regarding the position of the BMU and blending it with the weights stored in the frame buffer, the convolution is accomplished.

KFM Resolution	32×32	64×64	128×128	256×256	512×512	1024×1024
Software	11	30	45	187	982	5721
Graphics Hardware	3	36	50	95	284	1077

Table 1: The KFM execution times (in seconds) of a standard software implementation compared with the proposed *OpenGL* realization on *SiliconGraphics InfiniteReality* graphics hardware.

3.4 Limitations

Some limitations due to the special format of the frame buffer memory are mentioned in the following.

Limited number of color components. The described algorithm has been limited to four-dimensional Kohonen maps, due to the limited size of four components of one frame buffer pixel vector. Extensions to higher dimensions can easily be accomplished by enabling a frame buffer which is larger than required for the KFM, and applying several subregions of the window to instantiate further components of the KFM weight matrix. To preserve the efficiency of the whole algorithm it should be regarded that the needed memory should not exceed the available frame buffer size, since otherwise, time consuming pixel transfers are necessary to swap the frame buffer and the CPU main memory.

Quantization effects. Each component of the pixel buffer has a resolution of 8 bits. To diminish quantization effects, the calculated results are scaled implicitly after each operation by applying the color matrix and the `glMinmaxEXT` routine to generate a scaling matrix. The comparison of the results of the proposed realization with the results from a standard software implementation did not expose significant differences. Extending the component resolution can also be accomplished by using several frame buffers whose single vector components are concatenated. Of course, this needs an extension of the *OpenGL* standard arithmetic.

4 Tests and conclusion

The realization of Kohonen feature mapping through the usage of a common graphics hardware accelerator is presented. The algorithm is translated into about 50 *OpenGL* calls which operate on screen memory which stores an adapted representation of the KFM. The algorithm profits from the parallel hardware implementation of the *OpenGL* functionality, and increases execution speed of the KFM essentially compared to a common software implementation.

Table 1 shows the comparison of a standard implementation on a *SiliconGraphics* workstation (*MIPS* R10000 processor, 195 MHz) with the described implementation as *OpenGL* code which is executed on a *SiliconGraphics InfiniteReality* graphics board. For large KFM dimensions, accelerations of up to 500 % are accomplished.

References

[1] Andrew S. Glassner. *Principles of Digital Image Synthesis.* Morgan Kaufmann, San Francisco, CA, 1995.

[2] J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991.

[3] Teuvo Kohonen. *Self-organizing maps.* Springer Verlag, New York, 1997.

[4] Tom McReynolds, editor. *Programming with OpenGL: Advanced Rendering.* ACM, SIGGRAPH, 1996. SIGGRAPH '96 course notes.

[5] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide.* Addison-Wesley, Reading MA, 1993.