Graphics Hardware Implementation of the Parameter-Less Self-organising Map

Alexander Campbell[1], Erik Berglund[2], and Alexander Streit[1]

[1] Faculty of IT
Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
ab.campbell@qut.edu.au
[2] Information Technology and Electrical Engineering
University of Queensland
St. Lucia, QLD. 4072, Australia

Abstract. This paper presents a highly parallel implementation of a new type of Self-Organising Map (SOM) using graphics hardware. The Parameter-Less SOM smoothly adapts to new data while preserving the mapping formed by previous data. It is therefore in principle highly suited for interactive use, however for large data sets the computational requirements are prohibitive. This paper will present an implementation on commodity graphics hardware which uses two forms of parallelism to significantly reduce this barrier. The performance is analysed experimentally and algorithmically. An advantage to using graphics hardware is that visualisation is essentially "free", thus increasing its suitability for interactive exploration of large data sets.

1 Introduction

One of the consequences of the explosion of data collection size and dimensionality is the need for unsupervised approaches to analysis and, in particular, dimensionality reduction. The Self-Organising Map (SOM) is an unsupervised learning technique capable of mapping a high dimensional input space to a lower (generally two-dimensional) output space such that the topology of the input space is preserved in the output space. This allows intuitive exploration of the data in an easily comprehensible 2D map that preserves neighbourhood relations.

For large, high dimensional data sets, or for applications where interactive use of the SOM is required, training times become an issue. This has led to the development of specific hardware implementations such as [1]. However, in order to find widespread application as, for example, an interactive web search tool, custom hardware solutions are obviously infeasible.

Driven primarily by the games industry the speed and programmability of commodity graphics hardware have been developing apace - performance increases outstripping Moore's Law by approximately a factor of 3:1[2]. Over the last few years the graphics processing unit (GPU) has not surprisingly been gaining interest as an inexpensive high performance platform for non-graphics centric computation. General Purpose computation on Graphics Processing Units

M. Gallagher, J. Hogan, and F. Maire (Eds.): IDEAL 2005, LNCS 3578, pp. 343–350, 2005.
© Springer-Verlag Berlin Heidelberg 2005

(GPGPU) is a burgeoning field. GPU is suited particularly to implementations which exploit the parallelism of the graphics rendering pipeline, and which match the single instruction multiple data (SIMD) format at some point in their execution.

A recent development, the Parameter-Less Self-Organising Map (PLSOM) [3, 4], markedly decreases the number of iterations required to get a stable and ordered map. It also has two features which make it highly suited to interactive use: *plasticity preservation* and *memory*. These mean that it handles well being retrained with new data which may be greater than the range of the previously used data (plasticity) or smaller than the range of previous data (memory).

These factors made it an ideal candidate for an efficient parallel implementation, which we present here. We seek to demonstrate the suitability of the PLSOM for parallelisation and graphics hardware implementation, and a significant theoretical and actual performance superiority of this implementation. We have not provided a reference implementation of the standard SOM - for the reasons already given our focus is on the *PL*SOM - however we do discuss briefly how they relate.

We start with the theoretical basis of the Parameter-Less SOM in Section 2, then provide an introduction to using graphics hardware for general purpose computation in Section 3. Section 4 contains the fusion of these elements in terms of implementation and algorithmic complexity. Experimental setup for performance testing and empirical results are presented in Section 5. We conclude by commenting briefly on the potential for sophisticated visualisation and interactive use.

2 The Parameter-Less SOM

The Self-Organising Map [5, 6] is an algorithm for mapping (generally) low-dimensional manifolds in (generally) high-dimensional input spaces. The SOM achieves this through unsupervised training, but one of the major problems have been selecting and tuning annealing schemes, since it must be done empirically in the absence of a firm theoretical basis. There is no need for a learning rate annealing scheme or neighbourhood size annealing schemes with the Parameter-Less SOM. The PLSOM, which is similar to the SOM in structure but differs in adaption algorithm, consists of an array of nodes, N. The nodes are arranged in a grid in output space so that one can calculate the distance between two given nodes. During training, an input in the form of a k-dimensional vector $\mathbf{x}$ is presented to the PLSOM. The winning node at timestep t, $c(t)$, is the node with an associated weight which most closely resembles the input, $c(t)$ is selected using Equation 1.

$$c(t) = \arg\min_i(||\mathbf{x}(t) - \mathbf{w}_i(t)||) \tag{1}$$

where $\mathbf{w}_i(t)$ is the weight vector associated with node i at timestep t. Then the weights are updated using Equations 4-5. The basic idea is to move the weight nodes associated with nodes close to c towards the input $\mathbf{x}$. How much to

move the weight vector of a given node i depends on the distance from i to c (in output space) and the neighbourhood function. The scaling of the neighbourhood function (the neighbourhood size) determines how a node which is far away from c is affected. A small neighbourhood size means relatively few nodes, close to c, are affected while a large neighbourhood function will lead to updates on more nodes further away from c. The weight update is scaled by a variable ϵ which is calculated according to Equations 2 and 3.

$$\epsilon(t) = \frac{||\mathbf{w}_c(t) - \mathbf{x}(t)||}{\rho(t)} \tag{2}$$

where $\rho(t)$ ensures that $\epsilon(t) \leq 1$.

$$\begin{aligned} &\rho(t) = \max(||\mathbf{x}(t) - \mathbf{w}_c(t)||, \rho(t-1)), \\ &\rho(0) = ||\mathbf{x}(0) - \mathbf{w}_c(0)|| \end{aligned} \tag{3}$$

ϵ is used to scale the weight update in two ways; directly, as part of Equation 4 and indirectly as part of Equation 5.

$$\Delta\mathbf{w}_i(t) = \epsilon(t) h_{c,i}(t)[\mathbf{x}(t) - \mathbf{w}_i(t)] \tag{4}$$

where $\Delta\mathbf{w}_i(t)$ is the change in the weight associated with node i at timestep t and $h_{c,i}$ is the neighbourhood function given in Equation 5.

$$h_{c,i}(t) = e^{\frac{-d(i,c)^2}{\Theta(\epsilon(t))^2}} \tag{5}$$

where e is the Euler number, $d(i,c)$ is the Euclidean distance from node i to node c in output space and $\Theta(\epsilon(t))$ is given by Equation 6.

$$\Theta(\epsilon(t)) = \beta\, ln(1 + \epsilon(t)(e-1)) \tag{6}$$

where β is a scaling constant related to the size of the network. For a n-by-m node network one would usually select β according to Equation 7:

$$\beta = \frac{m+n}{2} \tag{7}$$

The PLSOM achieves faster ordering, is independent of input space distribution, eases application and has a firmer theoretical basis.

3 Programmable Graphics Hardware

Commodity graphics hardware is designed primarily for real time approximation of lighting for 3D scenes. In the quest for more realistic approximations, recent graphics hardware has allowed programming of the graphics operations directly using programs that are referred to as *shaders*.

Through the careful construction of graphics commands, we can adapt the hardware to perform calculations, much like a co-processor. The graphics hardware expects polygons that are defined by their edge vertices. Each polygon is

transformed into fragments, which are multi-valued cells. We will refer to the collection of fragments as streams, in keeping with parallel computation terminology. Each fragment is processed by a *fragment shader*, also referred to as a kernel. As part of the hardware process, fragments are written to the framebuffer as pixels, ready for display to the user.

Control over the execution of the process remains with the CPU, since the CPU must instigate any instance of the stream operations. This is achieved by *rendering* a quadrilateral, which passes four vertices that bound the stream output, causing an instance of the kernel to be invoked for each element within the output stream. In our implementation we re-use the contents of the frame buffer as input, which under the architecture of the GPU means passing it to the fragment program as a texture.

The architecture of the graphics hardware is SIMD both as a vector processor and as a stream processor. As a vector processor each instruction can operate on up to four components of a vector simultaneously. As a stream processor the same program, or kernel, is executed for every element in the stream concurrently.

4 GPU Implementation of the PLSOM

There are two strategies to parallelising the self-organising map - vectorisation and partitioning [7] - and these essentially correspond to the two SIMD characteristics just mentioned. Vectorisation is the use of a vector-processor to operate on the (k-dimensional) components of each input in parallel. Partitioning of the map is done to allocate different processors to different sections; on which they can execute identical kernels. So by using graphics hardware we are well positioned to take advantage of both these optimisations. It should be possible to extend the algorithm given below to a standard SOM with very little modification: the only real difference is that way that the neighbourhood is chosen.

4.1 Self-organising Map Kernels

We focus on a single weight update for the whole map, that is to say a single value of t for Equations 1-6. This computation maps itself well to the stream paradigm and requires three separate kernels which we denote *computeDist*<>,*findMin*<> and *updateWeights*<>.

If we extract $d_i = ||\mathbf{x}(t) - \mathbf{w_i}(t)||$ from Equation 1 and formulate it as a single operation for the whole map, we have the distance vector $\mathbf{d} = ||\mathbf{x}(t) - \mathbf{w}(t)||$, and *computeDist*$< \mathbf{x}, \mathbf{w} >$ becomes our first kernel. The weight update is also easily treated as a stream operation using kernel *updateWeights*$< \mathbf{w}, c, \epsilon >$. Provided $c(t)$ is available for Equation 4 this approach should lead to significant speedup. Therefore our main focus is to construct a kernel which can be used to find $\arg\min_i(\mathbf{d})$ in an efficient manner.

Finding the maximum or minimum value of a set in parallel, and with only localised knowledge of state, can be achieved by iterating many local finds on ever smaller sets. This divide-and-conquer style of operation is termed a *reduction*

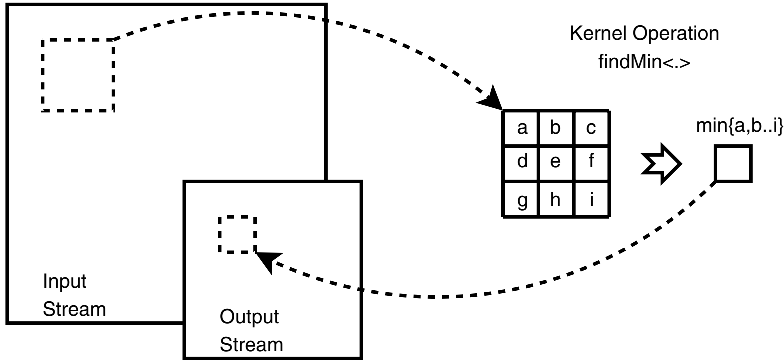

Fig. 1. Reduction Kernels

operation in the GPGPU literature. There are a number of implementation issues to be considered. With each iteration of the kernel operation we modify the coordinates of the quad we render to reduce the bounds of the stream each time. Eventually we are left with a one pixel buffer which is our desired value. Each time the kernel executes, the stream output is copied from its destination into a texture ready for input to the next stream operation. The kernel itself is relatively simple: it gathers a small number of values from the input texture and outputs the minimum value.

Given that our initial set size is N, we only require $q = \lceil log_\eta N \rceil$ iterations, where η is the size of the local sub-sets (and therefore the reduction factor). We term this operation *findMin*$< \mathbf{d}_j >$, where $\mathbf{d}_j$ represents the vector of values in the input stream at iteration j, $j = 1 \ldots q$, ie $\mathbf{d}_1 = \mathbf{d}$ and $|\mathbf{d}_q| = 1$. The nature of this reduction process is expressed in Figure 1.

4.2 Algorithm Analysis

Using these three stream operations leads to a concise algorithm:

1: $\mathbf{d} \leftarrow computeDist < \mathbf{x}, \mathbf{w} >$
2: $q \leftarrow ceil(log_\eta N)$
3: **for all** $j \leftarrow 1, 2, \ldots, q$ **do**
4: $\quad \mathbf{d}_{j+1} \leftarrow findMin < \mathbf{d}_j >$
5: **end for**
6: $c \leftarrow \mathbf{d}_q$
7: $updateWeights < \mathbf{w}, c, \epsilon >$

Assuming a number of processors equal to the number of nodes in the map, we have two standard stream operations - *computeDist*<> and *updateWeights*<> - with constant order time complexity, plus $\lceil log_\eta N \rceil$ iterations of *findMin*<>, which is essentially η compare instructions. This would result in a time complexity $T = \eta \lceil log_\eta N \rceil$ which is $O(logN)$.

However, in reality we have a limited number of processors, these being the twelve parallel pixel shader pipelines on our GeForce 6800 graphics card. With

P processors our time complexity is

$$T(N) = k\left\lceil \frac{N}{P} \right\rceil + \sum_{i=0}^{q-1} \eta \left\lceil \frac{\eta^i}{P} \right\rceil \tag{8}$$

where k is the number of (constant order) instructions in *computeDist* and *updateWeights* combined. This is $O(N/P)$ which demonstrates clearly the extent to which *partitioning* parallelism is exploited. Until some point $N >> P$, we should see roughly $logN$ growth.

4.3 Higher Dimensions

Our treatment so far has ignored the issue of what happens when we have greater than four dimensions. Arbitrary dimensions can be modelled using a 3D texture, with 4 dimensions to each z coordinate. The winning node is now found using a 2-step reduction. First the dimension distance is reduced to a scalar for each node, then the 2D coordinate and scalar distance are reduced as above. In our 3D texture, the x,y dimension are the 2D SOM grid, the z dimension is all the dimensions of that node. The graphics hardware's two SIMD characteristics are both utilised heavily in this situation indicating that similar performance advantages can be expected. Future work will explore this avenue.

5 Training Speed on Various Platforms

In order to give an estimation of the benefit of using programmable graphics hardware for PLSOM training we implemented the same training algorithm on 3 different platforms. In addition to a basic CPU implementation on a desktop and a GPU implementation using an NVIDIA graphics card on that same machine, we also tested a semi-parallelised implementation on a supercomputer.

- Target machine 1: A Coretech desktop with a 3.0 GHz Pentium 4 CPU and 1G RAM. Compiler: MSVC running in debug configuration with no optimisations. Operating system: Windows XP.
  Graphics Hardware: Albatron GeForce 6800 with 12 parallel pipelines.
- Target machine 2: The Queensland Parallel Supercomputing Foundation (QPSF) SGI Altix (64 bit) 3700 Bx2 supercomputer with 64 Intel Itanium 2 (1500Mhz) processors (although we only used 6 for our test) and 121 GB RAM. Compiler: icpc, the Intel parallelising C++ compiler. Optimisation level 3, parallelisation enabled. Operating system: 64-bit GNU/Linux.

For the test we trained an m-by-m node network with 2-dimensional input. The map is presented with 1000 inputs that are randomly generated, uniformly distributed in the unit square. The inputs are presented sequentially. The test program was written in C++ for the two non-GPU versions, a combination of C++ and Cg for the graphics implementation, and the code was functionally

Table 1. Execution Times 1000 Map Updates (Seconds)

number of nodes	Desktop computer	Super computer	Graphics card
729	3.93	0.24	1.67
4096	22.03	0.59	1.71
6561	36.37	1.18	1.72
16384	88.42	2.95	1.72
59049	317.80	11.51	3.32
236196	1278.04	69.26	6.65
531441	-	157.13	14.51
1048576	-	291.67	31.43

identical. Table 1 shows the execution times for 1000 map updates on these platforms.

Figure 2 shows the execution time growth rates of the PLSOM on the supercomputer and the GPU. This observed data indicates an time complexity of $O(N))$ for the GPU and the supercomputer after a critical point, however prior to this the GPU exhibits $logN$ growth as suggested in section 4.2. The parallelism of the graphics card is unable to overcome certain hardware-specific overheads until a certain point, however it is never slower than the desktop implementation in our experiments and it quickly overtakes the supercomputer performance. At

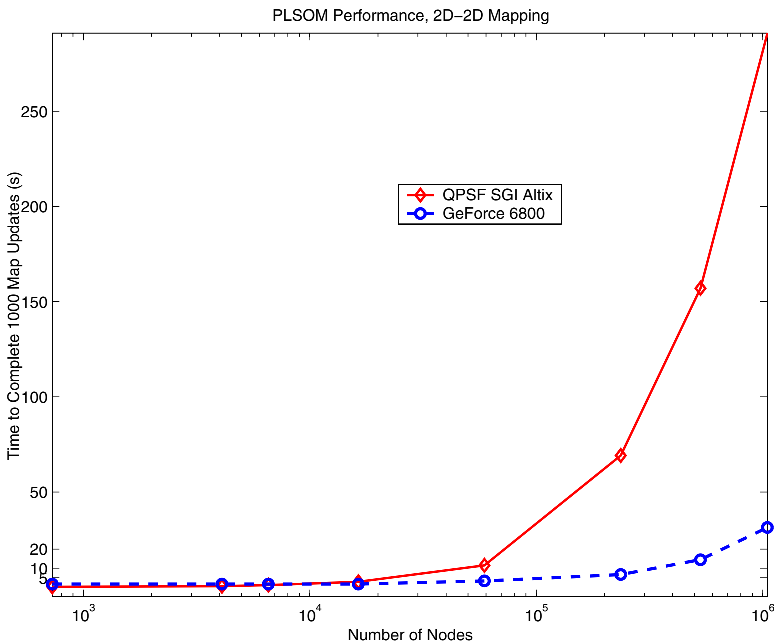

Fig. 2. Execution time growth rates

a map size of $1024 * 1024$ nodes we can see a 90% performance improvement of the GPU over the supercomputer.

This performance superiority cannot be *directly* translated to high dimensional data sets, however as we mentioned in section 4.3 both forms of parallelism would be heavily used in such an implementation and we plan to look at this in future work.

6 Discussion and Conclusion

In this paper we have described an implementation of the Parameter-Less Self Organising Map on commodity graphics hardware and provided an empirical performance analysis. As noted by [7] a desirable application of high speed training and re-training of the self-organising map is interactive analysis of web search results. Given this, inexpensive commodity graphics hardware is an ideal way to provide the computational power required, especially given that while the GPU is busy, the CPU is free for other tasks. Additionally, the visualisation capabilities of graphics hardware are 'on-tap' during the computational process - the execution times for the graphics card included displaying the weight matrix to the screen every ten iterations. This combination of superior visualisation power and what is essentially a tool for visualising higher-dimensional spaces seems synergetic.

References

1. Porrmann, M., Kalte, H., Witkowski, U., Niemann, J.C., Rückert, U.: A dynamically reconfigurable hardware accelerator for self-organizing feature maps. In: SCI 2001 (5th World Multi-Conference on Systemics, Cybernetics and Informatics). (2001) 242–247
2. NVIDIA Corporation: Cg toolkit user's manual: A developer's guide to programmable graphics. (2004)
3. Berglund, E., Sitte, J.: The parameter-less som algorithm. In: ANZIIS 2003. (2003) 159–164
4. Berglund, E., Sitte, J.: The parameter-less self-organizing map algorithm. (ITEE transactions on Neural Networks, accepted. In revisions)
5. Kohonen, T.: The self-organizing map. Proceedings of the IEEE **78** (1990) 1464–1480
6. Ritter, H., Martinetz, T., Schulten, K.: Neural Computation and Self-Organizing Maps - An Introduction. Addison-Wesley publishing company (1992)
7. Rauber, A., Tomsich, P., Merkl, D.: A parallel implementation of the self-organizing map exploiting cache effects: Making the som fit for interactive high-performance data analysis. In: IJCNN 2000 (International Joint Conference on Neural Networks). (2000) 24–27