Spiking Neurons on GPUs

Fabrice Bernhard
Renaud Keriven

Research Report 05-15
November 2005

CERTIS - ENPC
http://www.enpc.fr/certis
Ecole Nationale des Ponts et Chaussées
77455 Marne - Paris - France

Spiking Neurons on GPUs

Fabrice Bernhard and Renaud Keriven

Odyssee Project - INRIA/ENS/ENPC
45 rue d'Ulm, 75005 Paris, France,
Fabrice.Bernhard@polytechnique.org

Abstract. Simulating large networks of spiking neurons is a very common task in the areas of Neuroinformatics and Computational Neurosciences. These simulations are time-consuming but also often intrinsically parallel. The recent advent of powerful and programmable graphic cards seems to be a pertinent solution to the problem: they offer a cheap but efficient possibility to serve as very fast co-processors for the parallel computing that spiking neural networks need. We describe our implementation of three different problems on such a card: two image-segmentation algorithms using spiking neural networks and one multi-purpose spiking neural-network simulator. Using these examples we show the benefits, the challenges and the limits of such an implementation.

1 Introduction

1.1 Motivations

Considering the power of recent graphic cards in parallel computation, and the possibility to program them, it is no wonder that so many people have tried to divert them from their initial purpose. Simulations of spiking neural networks, very common in neuroinformatics, are highly parallel and are therefore one of the many tasks that could benefit from such cards.

However, the only reference to neural networks on graphic cards we have found was Konrad Pietras' implementation of a perceptron for lighting purposes [1]. Perceptrons are very different from spiking neural networks, and do not serve at all the same pupose. This work is therefore far from what interested us, which we now describe here.

We will briefly introduce spiking neural networks and how they differ from artificial neural networks, and then give details on the implementation of three different problems using such networks: two image-segmentation algorithms and a general-purpose spiking neural network simulator. We will try to show the benefits, the challenges and the limits of such a task and give our conclusion on its pertinence.

1.2 Spiking neural networks

Frank Rosenblatt's perceptron [2] in the late fifties is the first famous attempt to simulate the brain on a computer. Although able to learn and useful in some

simple classification tasks, it is limited, as Minsky and Papert's work showed [3]. More generally, artifical neural networks are too far from a biologically plausible model to interest scientists in neuroinformatics and computational neurosciences.

Research in those domains is oriented towards spiking neural networks, for which models also take into account the internal dynamic of each neuron: the membrane potential, the firing rate, the amount of ions released at the synapses, etc. A very simple but common and already powerful model is for example Lapique's integrate-and-fire neuron [4] which describes the neuron's membrane as a leaky condensator. This is the one we are going to use in the three following implementations.

Spiking neural networks' dynamics are however harder to describe mathematically than artificial neural networks'. Simulation is therefore an essential part of their study, but these are also slower: the pertinent time-scale would be roughly for example the average duration between spikes, which can be as long as a hundred cycles of a timestep simulation, a lot more than for an artificial neural network, whose weights are updated at every cycle. Fortunately, since these networks try to mimick the brain's behaviour, where neurons all work independently from one another, they are intrisically parallel.

1.3 Graphics Processing Unit

The primary goal of 3D graphic cards is to replace the CPU in all the calculations needed to show 3-dimensional objects to the screen: projection of the objects, texturing, lighting, etc.

One of the tasks that graphic cards accelerate very significantly is the calculations of the screen's pixels' colour. After having projected the different objects on the sceen according to the position of the viewer, a program (the fragment shader) is run in parallel on all pixels to decide their colour, taking into account objects' textures, lighting, fog, etc. This is accelerated by up to 24 pipelines and is the step that we will divert to serve as a powerful co-processor for our spiking neural networks' simulations.

Basically, we will map a neuron to a pixel and replace the fragment program by one that will calculate the updated state of our neurons' variables. By repeating this multiple times, we will have our timestep simulations. A very good guide on how to efficiently do this can be found on Dominik Göddeke's page [5].

Our technical choice was to use OpenGl to interface with the graphic card, the FrameBufferObject class [6] to render to textures and NVidia's Cg to write the fragment programs. It seemed to us to be the most flexible and efficient solution at the time of the experiments.

2 Implementation of three different algorithms using spiking neural networks

2.1 Synchrony and Desynchrony in Integrate-and-Fire Oscillators, S.R. Campbell, D.L. Wang, and C. Jayaprakash [7].

Quick description of the algorithm This first algorithm is a segmentation of a picture based on a locally-excitatory globally-inhibitory spiking neural network. For each pixel of the picture to segment, there is a spiking neuron associated to it, verifying a Lapicque-like differential equation:

$$\frac{dV}{dt} = -V + I \tag{1}$$

I is defined at the beginning so that neurons in homogeneous regions spike spontaneously, ie $I > \theta$ for them, where θ is the threshold of the neuron. Every neuron is connected in a excitatory manner to all its local neighbours (up, down, left and right) if they have similar colour. If a neuron emits a spike, the potential of its similar neighbours is increased. Every neuron is also globally connected to all the others in an inhibitory manner. The result is that neurons coding a segment of homogeneous colour will excite themselves locally and after a short time will synchronize their spikes. Neurons of different segments will inhibit themselves through the global inhibition and stay desynchronized. Segments of homogeneous colour will therefore appear after a short time by just looking at the spiking times of every neurons: all neurons of a same segment will spike together while different segments will spike at different times.

Implementation on the GPU All arrays like V and I are put into textures. The evolution of the potential can be implemented as a euler-timestep, but here in that simple case, we can also observe that:

$$V(t) = I(1 - e^{-t}) \Rightarrow V(t + dt) = I + e^{-dt}\,(V(t) - I) \tag{2}$$

The first challenge appears with the spiking. In a graphic card, we have a CREW (concurrent reading, exclusive writing) mechanism: we can read in parallel anywhere in the textures (a process also called "gather" [9]), but we can write only in one place: the destination pixel ("scatter" is not possible). Therefore we cannot, when a neuron fires, update the connected neuron's potential, as we would simply do in sequential mode. To solve this problem we convert the scatter problem to a gather problem by introducing a new array, E, coding the excitatory state of a neuron. At every step, every neuron reads the excitatory state of its connected neurons. If it is equal to 1, we add the appropriate excitation. This method is fast enough for a few number of connections, since reading in a texture is very fast on graphic cards.

For the global inhibition however, this method would mean that every neuron must read at each pass from every other neuron. This would become very time-consuming. Since the global inhibition here just depends on the number

of neurons having fired at the precedent pass we just need to determine that number. The first idea we had was to implement it through a reduction: For a $n \times n$ square, we calculate

$$E(x,y) = E(x,y)+E(x+2^i,y)+E(x,y+2^i)+E(x+2^i,y+2^i), i \in [|0, log_2(n)|]. \tag{3}$$

We therefore have after $log_2(n)$ cycles $E(0,0) = \sum E(x,y)$. The second idea was to use the hardware acceleration of mipmapping to perform that same reduction. For a texture of size n, the mipmap is a pyramid of $log_2(n)$ textures of sides $\frac{n}{2^i}, i \in [|1, log_2(n)|]$ which pixels are the average of the source texture. Therefore the last texture is one pixel large and contains the average of all points of the source texture, in our case: $\frac{1}{n^2}\sum E(x,y)$. However this is not very precise since it is done on 8 bits and is not very flexible. The solution found was to use a trick specific to graphic cards: there is a counter that is able to count the number of pixels written in a pass: the Occlusion Query. We therefore just need to run a pass on the graphic card which writes the pixel (x,y) only if $E(x,y) = 1$, else it exits the fragment program with the CG command `discard()` and then count the number of pixels written with the occlusion query.

Fig. 1. The original picture followed by the different steps of its segmentation by the algorithm of Campbell, Wang and Jayaprakash

Results Thanks to the occlusion query trick we were able to put the whole algorithm on the GPU and gain a significant increase in speed: it is about 7 times faster on a 6800 Ultra and 10 times faster with a 7800 256MB than running solely on a Xeon 2,8 GHz with 1 Gb RAM. The results are identical. The visualisation is also made easy. In figure 1 we represent in red the membrane potential, in green the excitement state. Blue is just there for visualisation reasons: it is set to 1 when the neuron spikes and then decreases exponentially. Neurons of the same segment, who spiked at the same time, have therefore the same blue.

Finally it is interesting to remark that with this algorithm using spiking neurons we are able to see connexity, since connex neurons are synchronized. This is something that a simple perceptron cannot do and shows the potential of spiking neural networks. More details can be found in [10].

2.2 Image Segmentation by Networks of Spiking Neurons, J.M. Buhmann, T. Lange, and U. Ramacher [8]

Quick description of the algorithm This second algorithm is a segmentation algorithm based on a histogram clustering method: we divide the picture in $N \times N$ little rectangles for which we calculate the histogram of luminosity, for M grey levels. The segmentation then tries to assign to a same segment blocs of similar luminosity histogram. For that we use a spiking neural network designed to roughly minimize the objective function

$$H = \sum_{1}^{N \times N} D^{KL}(h_i, \bar{h}_{s(i)}) \tag{4}$$

where D^{KL} is a distance, h_i the histogram of block i and $\bar{h}_{s(i)}$ the average histogram of the segment to which block i belongs. We are therefore looking for segments where the member blocs are the more homegeneous possible.

The spiking neural network is composed of $K \times N \times N$ neurons, where K is the a priori known number of segments: for each bloc (i, j), we will have K neurons that will compete in a winner-take-all mechanism to choose which segment the bloc belongs to. The neurons are modeled by the simple differential equation:

$$\frac{dV}{dt} = -\frac{V}{\rho} \tag{5}$$

The neuron (i, j, k) receives input spikes with weights $\forall m \in [|1, M|], w_{k,m}$ and with probability $h_{i,j}(m)$: the relative importance of the mth grey level in the histogram of block (i, j). All K neurons of bloc (i, j) have inhibitory connections between them to implement the winner-take-all mechanism. There are also connections between contiguous blocs to smooth the segmentation: neurons coding for the same segment in direct neighbours have excitatory connections while neurons coding for different segments have inhibitory connections, so that direct neighbours tend to belong to the same segment if similar enough.

Knowing $\bar{h}_k$, the average histogram of blocs in segment k, the weights are updated at every cycle with the following rule:

$$\frac{d}{dt} w_{k,m} = \alpha \left(exp\left(w^{max} - w_{k,m}(t)\right) \bar{h}_{k,m} - 1 \right) \tag{6}$$

Implementation on GPU This algorithm is divided in four steps:

- We first update with a pass the weights $w_{k,m}$.
- We then update in a pass $V_{i,j,k}$ taking into accounts the input spikes and the neighbour's connections.
- We calculate in a third pass $s(i, j)$: the segment to which bloc (i,j) belongs, defined by the k for which neuron (i,j,k) fires most.
- Finally we calculate the average histogram of segment k $\bar{h}_k$ on the CPU, since it is unfortunately not possible to do it efficiently on the graphic card.

Of course, implementing this is not as straightforward on the GPU as it would be on CPU. First of all we need random numbers to implement noise, used to avoid staying in local minima, and also to send the input spikes with probability $h_{i,j}(m)$. We did this by filling a giant texture with random numbers, but it is clearly not an optimal solution since the period of this pseudo-random generator is equal to the size of the random-filled texture, a number that cannot be much bigger than the size of our network.

We stored our three-dimensional variable: $V(i,j,k)$ of size (N,N,K), in a $(N \times K, N)$ texture. However we notice that we now need to have $N \times K \leq 4096$.

To avoid border effects we chose to add a null border to the texture containing V, which does not imply more calculation since we can tell the card to render only the inner part.

We are not always able to transform conditional branches into multiplications by 0 or 1 to avoid slow conditional branching. However we have observed that there exists conditional arithmetic operators in the assembly language, used if there is just a simple arithmetic operation inside the conditional branch. This enables us to make multiple tests without significant speed loss.

The last part of the algorithm is not efficiently parallelisable so we leave it on the CPU. However the transfer of the data from the GPU to the CPU and back is a costly operation. We therefore try to do that step only once every 100 steps.

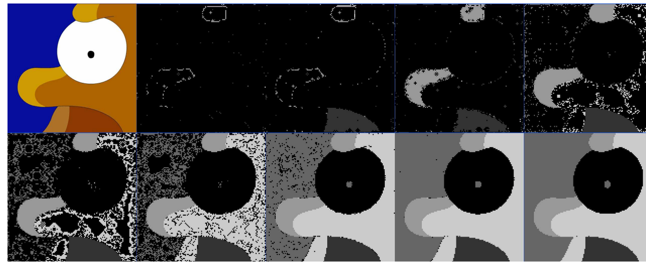

Fig. 2. The original picture followed by the different steps of its segmentation by the algorithm of Buhmann and Lange

Results We manage to get once again a significant acceleration: 5 times faster on a 6800 Ultra and 10 times faster on a 7800 256MB, on the same test configuration as before. The convergence is however a very little slower, due surely to the pseudo-random texture which is clearly not as good as a real pseudo-random generator, and the last part, on the CPU, not executed at every step. It is interesting to note that this last part is considered by the authors as the weakness of their algorithm in terms of biological plausibility. The results are visualised by projecting the outcome of the third pass: $s(i,j)$, see for example figure 2.

2.3 Generalisation to an easily modifiable network for use in neuroinformatics

The implementation of the two precedent algorithms on GPU being convincing, we decided to make a more general simulator, that would be flexible enough to be adapted to different networks used in neuroinformatics.

The main new challenge here was to provide an easy and flexible way to define the connections between neurons. The idea proposed for M connections per neuron in a $N \times N$ network was to store them in a $(N\sqrt{M/4}, N\sqrt{M/4})$ RGBA (4 colour components) texture, with a connection (two coordinates) in each colour component. This is possible since our coordinates fit in 16 bits and Cg provides a function to store two 16 bits variables in a 32 bits component: `pack_2half(half2 (x,y))` Using squares of side $\sqrt{M/4}$ for each neuron's connections is motivated by the fact that textures' width and height are both limited to 4096. Very large or long rectangles are therefore not possible.

At each neuron's update we will then go through the table of connections with two nested 'for' loops. Something which is possible only in the latest generation cards. There is a limitation here in the 6800 Ultra, that each loop has to be smaller than 256, but that is not a real problem since we will face another limitation before: the size of the texture. They are limited to 4096×4096, and that number is still theoretical since in many cards such a texture would fill the entire memory on its own. But roughly, we could imagine with a 3584×3584 neighbours' texture, for example:

- $896 \times 896 = 802816$ neurons with 64 neighbours each
- $512 \times 512 = 262144$ neurons with 196 neighbours each
- $128 \times 128 = 16384$ neurons with 3136 neighbours each

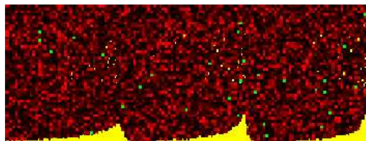

Fig. 3. Detail of the simulation of an excitatory spiking neural network, with tracking of some neurons' spiking times (yellow dots) and a global spiking histogram

However, the more connections there are, the more texture lookups are done. The observation was that for M big, the time needed increased linearly, limiting the size of M. In our experiments, we tried a 896×896 large network with 64 neighbours for each neuron. We had a pass rate of about 22 per second on a 7800 256MB, which was more than 20 times faster than a CPU-only implementation. One interesting reason for such a gain is that our model also included electrical synpases, which means that neurons adjust a little to the potential of their

connections at every pass. And this is not cache-friendly, therefore favouring even more the GPU, with its faster memory. In figure 3 you can see this basic excitatory network with random but constant connections, with the red value representing V, to which we added the tracking of some neurons' spikes (the yellow pixels) and a histogram of the global spiking activity.

3 Conclusion

We have clearly seen a significant speed increase, (between 5 and 20 times faster) in all the algorithms implemented, thanks to the inherently parallel way of functioning of neural networks. But there are other practical reasons that make the GPU very interesting: it is cheap compared to clusters, more flexible than a hardware implementaton, has a bright future since GPUs' speed increases faster than CPUs' speed [9] and there is always the possibility to leave parts of the algorithms on the CPU if this is more efficient.

Different limitations exist, either due to the parallel nature of the calculations or to the fact that graphic cards are still designed more towards 3-D graphics rendering than general purpose calculations. The most important ones being: the impossibility to "scatter", therefore imposing neurons to listen for a possible incoming spike at every pass, slow conditional branching, the absence of a random-generator function and the limited size of the textures. But we managed to bypass more or less all these problems in our implementations using different tricks. Some of these limitations might even disappear in future generations' card, making, in our opinion, the future of simulations of spiking neural networks on GPU very interesting.

References

1. K. Pietras. GPU-based multi-layer perceptron as efficient method for approximation complex light models in per-vertex lighting. http://stud.ics.p.lodz.pl/˜keyei/lab/atmoseng/index.html, 2005
2. F. Rosenblatt Principles of neural dynamics. Spartan Books, New York, 1962
3. M.L. Minsky, and S.A. Papert Perceptrons. MIT Press, 1969
4. L.Lapicque Recherches quantitatives sur l'excitation electrique des nerfs traitée comme une polarisation. J Physiol Pathol Gen 9:620-635, 1907
5. Dominik Göddeke. GPGPU::Basic Math Tutorial. http://www.mathematik.uni-dortmund.de/˜goeddeke/gpgpu/tutorial.html, 2005
6. Framebuffer Object (FBO) Class. http://www.gpgpu.org/developer/
7. S.R. Campbell, D.L. Wang, and C.Jayaprakash. Synchrony and Desynchrony in Integrate-and-Fire Oscillators. *Neural Computation 11*, pages 1595–1619, 1999.
8. J.M. Buhmann, T. Lange, and U. Ramacher. Image Segmentation by Networks of Spiking Neurons. *Neural Computation 17*, pages 1010–1031, 2005.
9. J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *EuroGraphics 2005*, 2005.
10. D.L. Wang. On connectedness: A Solution Based on Oscillatory Correlation. *Neural Computation 12*, pages 131–139, 2000.