GPGPU Implementation of Growing Neural Gas: Application to 3D Scene Reconstruction

Sergio Orts, Jose Garcia-Rodriguez

Department of Computing Technology at University of Alicante. Alicante (Spain)

Diego Viejo, Miguel Cazorla, Vicente Morell

Instituto de Investigación en Informática at University of Alicante. Alicante (Spain)

Abstract

Self-organising neural models have the ability to provide a good representation of the input space. In particular the Growing Neural Gas (GNG) is a suitable model because of its flexibility, rapid adaptation and excellent quality of representation. However, this type of learning is time consuming, specially for high-dimensional input data. Since real applications often work under time constraints, it is necessary to adapt the learning process in order to complete it in a predefined time. This paper proposes a Graphics Processing Unit (GPU) parallel implementation of the GNG with Compute Unified Device Architecture (CUDA). In contrast to existing algorithms, the proposed GPU implementation allows the acceleration of the learning process keeping a good quality of representation. Comparative experiments using iterative, parallel and hybrid implementation are carried out to demonstrate the effectiveness of CUDA implementation. The results show that GNG learning with proposed implementation achieves a speed-up of 6x com-

Email addresses: sorts@dtic.ua.es (Sergio Orts), jgarcia@dtic.ua.es (Jose Garcia-Rodriguez), dviejo@dccia.ua.es (Diego Viejo), miguel.cazorla@ua.es (Miguel Cazorla), vmorell@dccia.ua.es (Vicente Morell)

pared with single-threaded CPU implementation. GPU implementation has been also applied to a real application with time constraints: acceleration of 3D scene reconstruction for egomotion, in order to validate the proposal.
Keywords: Growing Neural Gas, Parallel computing, GPU, CUDA, Multicore, 3D Reconstruction, Egomotion

1. Introduction

Unsupervised classification is also known as data clustering and is defined as the problem of finding homogeneous groups of data points in a given multidimensional data set. Each of these groups is called a cluster and defined as a region where the density of data points is locally higher than in others regions. Another objective of unsupervised learning can be described as topology learning: which given a high-dimensional data distribution, consists in finding a topological structure that closely reflects the topology of the data distribution [1].

Self-organising models [2] place their neurons so that their positions within the network and connectivity between different neurons are optimized to match the spatial distribution of activations. As a result of this optimization, existing relationships within the input space will be reproduced as spatial relationships among activated neurons. In particular, Growing Neural Gas [3] is an incremental model able to learn the important topological relations in a given set of input vectors by means of a simple hebb-like learning rule.

Growing models have been widely used in recent years in many applications due to their attributes of growth, flexibility, rapid adaptation, and excellent quality of representation of the input space. These features convert the neural networks in a suitable model to solve problems that deal with dynamic input data. Related

works about computer vision and man-machine interaction like image compression [4], segmentation and representation of objects [5, 6, 7], objects tracking [8, 9, 10], gestures recognition [11, 12, 13], or 3D reconstruction [14, 15, 16] have been developed in the last years.

However, in many cases these applications present temporal constraints, that is why it is necessary to look for mechanisms that accelerate the learning process. In order to accomplish the acceleration of the neural network learning algorithm, a redesign of the sequential algorithm executed onto the CPU to exploit the parallelism offered by the GPU has been carried out. Current GPUs have a large number of processors that can be used for general purpose computing. The GPU is specifically appropiate to solve computationally intensive problems that can be expressed as data parallel computations [17, 18]. However, implementation on GPU requires the redesign of the algorithms focused and adapted to its architecture. In addition, the programming of these devices has different restrictions such as the need for high occupancy in each processor in order to hide latencies produced by memory access, management and synchronization of different threads running simultaneously, the proper use of the hierarchy of memories, and other considerations. Researchers have already successfully applied GPU computing to problems that were traditionally addressed by the CPU [19, 20, 17].

The GPU implementation used in this work is based on NVIDIA's CUDA architecture [21], which is supported by most current NVIDIA graphics chips. Supercomputers that currently lead the world ranking combine the use of a large number of CPUs with a high number of GPUs.

Neural networks have been used successfully in previous works to reduce the dimensionality of 3D input data maintaining a good topology preservation [15],

[22], [23] and [16].

In particular, 3D scene reconstruction is a time consuming task, that is fundamental in most mobile robotic systems [24, 25, 26, 27, 28, 29, 30]. However, most of these works do not deal with real time restrictions.

To validate our work we have applied our GNG accelerated implementation to the extraction and model of features from 3D raw data [31, 32, 33, 34]. Moreover, using this method, apart from accelerating the routine, we achieve two other advantages: a complexity reduction (when comparing with raw data) and an improvement of speed-up without decreasing the quality of the representation obtained.

The rest of the paper is organized as follows: Section 2 describes basic concepts of GPGPU architecture and CUDA software. Section 3 provides a description of the topology learning algorithm of the GNG, and how the algorithm is fitted onto GPGPU architecture. Section 4 presents some experiments and results of the parallel implementation running onto a GPU compared with the single-threaded and multi-threaded CPU versions. Finally, section 5 presents a real application with time constraints to validate our implementation, followed by our main conclusions and future work.

2. GPGPU architecture

A CUDA compatible GPU is organized in a set of multiprocessors as shown in figure 1 [35]. These multiprocessors called Streaming Multiprocessors (SMs) are highly parallel at thread level. However, the number of multiprocessors varies depending on the generation of the GPU. Each SM consists of a series of Streaming Processors (SPs) that share the control logic and cache memory. Each of these

SPs can be launched in parallel with a huge amount of threads. For instance, the GT400 chip family supports up to 1024 threads per SM, with 480 SPs distributed between 15 SMs. The GT400 chip is capable of performing a computing power of 1,5 teraflops, launching a total of 15,360 threads simultaneosly. The current GPUs have up to 12 GBytes of DRAM, referenced in figure 1 as global memory. The global memory is used and shared by all the multiprocessors, but it has a high latency.

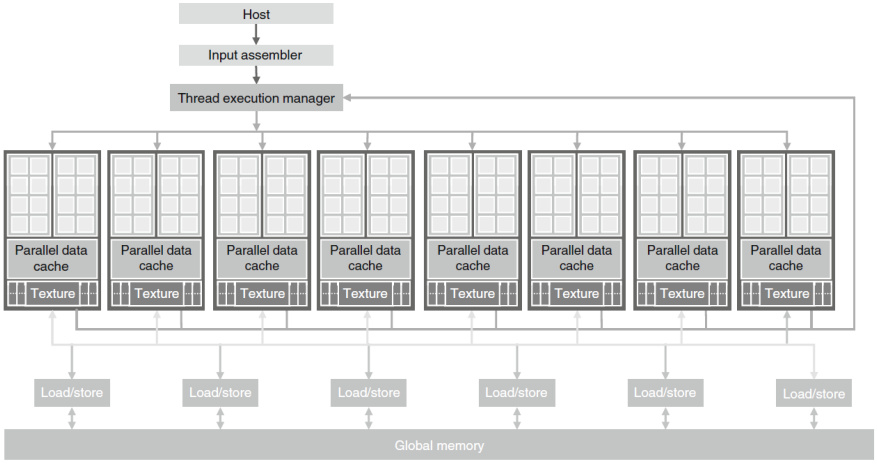

Figure 1: CUDA compatible GPU Architecture.

CUDA architecture reflects a SIMT model: Single Instruction, Multiple Threads. These threads are executed simultaneously working onto large data in parallel. Each of them runs a copy of the kernel[1] on the GPU and uses local indexes to be identified.

Threads are grouped into blocks to be executed. Each of these blocks is allocated on a single multiprocessor, enabling the execution of several blocks within a

[1]Piece of code that is executed on the GPU.

multiprocessor. The number of blocks that are executed depends on the resources available on the multiprocessor, scheduled by a system of priority queues. Within each of these blocks, the threads are grouped into sets of 32 units in order to carry out a fully parallel execution onto processors. Each set of 32 threads is called $warp$. In the architecture, there are certain restrictions on the maximum number of blocks, $warps$ and threads on each multiprocessor, but it varies depending on the generation and model of graphic cards. In addition, these parameters are set for each execution of a kernel to get the maximum occupancy of hardware resources and obtain the best performance. Experiments section shows how to fit these parameters to execute our GPU implementation.

CUDA architecture has also a memory hierarchy. Different types of memory can be found: constant, texture, global, shared and local registries. The shared memory is useful to implement caches. Texture and constant memory are used to reduce the computational cost avoiding global memory access which has high latencies.

In the last years, a large number of applications have used GPUs to speed up the processing of neural networks algorithms [36, 37, 38, 39, 40, 41] applied to various computer vision problems such as the representation and tracking of objects in scenes [42], face representation and tracking [43] or pose estimation [44].

3. GNG implementation using GPUs

From the Neural Gas model [45] and Growing Cell Structures [46], Fritzke developed the Growing Neural Gas model [3], with no predefined topology of union between neurons, in which from an initial number of neurons, new ones are

added (figure 2).

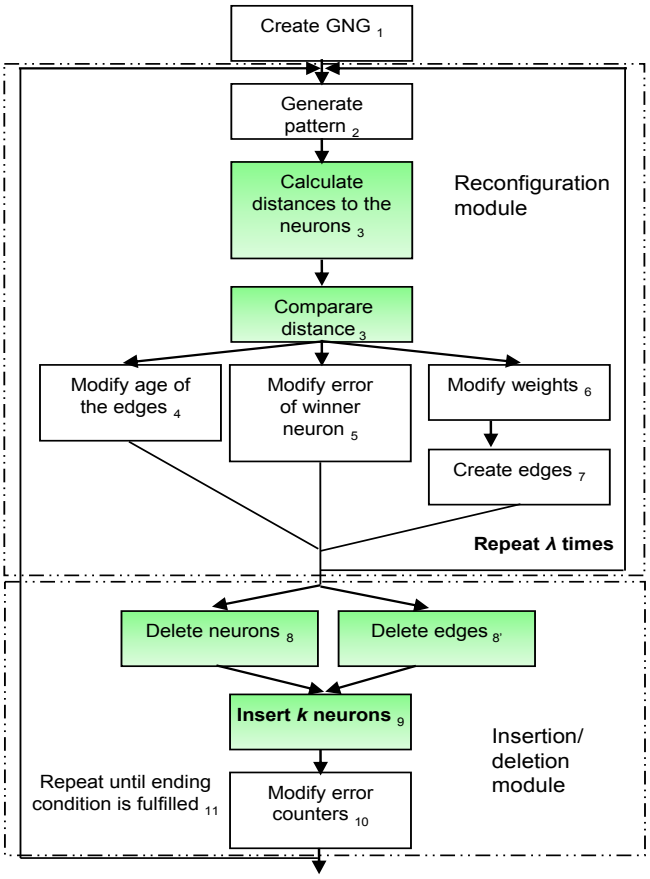

Figure 2: GNG learning algorithm remarking the parallel stages.

GNG learning algorithm has a high computational cost, we propose a method to accelerate it using GPUs and taking advantage of the many-core architecture provided by these devices, as well as their parallelism at the instruction level. GPUs are a specialized hardware for computationally intensive high-level parallelism that uses a higher number of transistors to process data and less for flow control or management of the cache, unlike in CPUs. We have used the architecture and the programming tools (language, compiler, development environment,

debugger, libraries, etc) provided by NVIDIA to exploit their hardware parallelism.

3.1. GNG Algorithm

GNG is an unsupervised incremental clustering algorithm that given some input distribution in R^n, incrementally creates a graph, or network of nodes, where each node in the graph has a position in R^n. The model can be used for vector quantization by finding the code-vectors in clusters. These code-vectors are represented by the reference vectors (the position) of the nodes. It can also be used for finding topological structures that closely reflects the structure of the input distribution. GNG learning is a dynamic algorithm in the sense that if the input distribution slightly changes over time, it is able to adapt, moving the nodes to the new input space.

Starting with two nodes, the algorithm constructs a graph in which nodes are considered neighbours if they are connected by an edge. The neighbourhood information is maintained throughout the execution by a variant of competitive Hebbian learning (CHL).

The graph generated by CHL creates an "induced Delaunay triangulation" that is a sub-graph of the Delaunay triangulation corresponding to the set of nodes. The induced Delaunay triangulation optimally preserves the topology in a very general sense [47]. CHL is an essential component of the GNG algorithm since it is used to process the local adaptation of nodes and insertion of new ones.

The network is specified as:

- A set N of nodes (neurons). Each neuron $c \in N$ has its associated reference vector $w_c \in R^n$. The reference vectors can be regarded as positions in the input space of their corresponding neurons.

- A set of edges (connections) between pairs of neurons. These connections are not weighted, and its purpose is to define the topological structure. An edge aging scheme is used to remove connections that are invalid due to the motion of the neurons during the adaptation process.

GNG uses parameters that are constant with time. Furthermore, it is not necessary to decide a priori the number of nodes to use since nodes are added incrementally during execution. Insertion of new nodes stops when an user defined performance criteria is fulfilled or when a maximum network size has been reached.

The adaptation of the network to the input space vectors is produced in step 6. The insertion of connections (step 4) between the winning neuron and the second closest to the input signal provides the topological relationship between the neurons (figure 2).

The elimination of connections (step 8) removes the edges that are no longer part of that topology. This is done by removing the connections between neurons that are no longer near or that have other neurons located closer, so that the age of these connections exceeds a threshold.

The accumulation of the error (step 5) can identify those areas of the input space of vectors where it is necessary to increase the number of neurons to improve the mapping.

3.2. Estimating the upper bound of acceleration factor

After presenting the different stages of the algorithm, and before tackling the parallel implementation of these stages, it is necessary to know what percentage of instructions is executed at each step in respect to the total number. In order to achieve this, we use a profiler so that depending on the values of the parameters

with which we have adjusted the algorithm (number of neurons and number of input patterns) we obtain the percentage of instructions executed at each stage.

It can be seen that most of the execution time of the algorithm is consumed in the winning neurons search stage, which also calculates the Euclidean distances.

Table 1 shows the percentage of instructions occupied at each stage of the algorithm for different values of the number of neurons N and input patterns λ. It is also shown how stage 3 increases its percentage in respect to the total, when N and λ are increased.

Neurons	Patterns	Stage 2	Stage 3	Stage 4,5,6,7	Stage 8	Stage 9
1000	500	1,8	**73,30**	15	1,2	0,8
5000	500	0,7	**88,80**	5,8	0,9	1
10000	500	0,4	**93,20**	3,3	0,6	0,8
20000	500	0,3	**97,60**	1,9	0,5	0,9
1000	1000	1,8	**69,60**	21,3	0,6	0,3
5000	1000	0,7	**90**	5,6	0,5	0,5
10000	1000	0,4	**94,3**	3,2	0,3	0,4
20000	1000	0,3	**96,5**	1,9	0,2	0,4

Table 1: Percentage of executed instructions at each stage of GNG presented in figure 2

Once this information has been obtained we apply different metrics of parallel computing to estimate which would be the overall maximum acceleration that we can obtain assuming that we can accelerate these stages by a factor S. The metrics used are widely known: Amdahl's Law [48] and other performance metrics of parallel computing [49, 50, 51].

In particular, we focus our study on the modern version of Amdahl's Law,

which states that if a fraction f is accelerated by a factor S, the overall acceleration is:

$$Speedup(f, S) = \frac{1}{(1 - f) + \frac{f}{S}} \quad (1)$$

This is the equation that better estimates the theoretical maximum acceleration that can be obtained using parallel implementations on GPUs as it applies the achieved improvement on a fraction of the code instead of applying it to the total number of cores. The number of cores can be used to measure the acceleration in the case of the execution onto a single GPU core, but in our case and in most of the cases, the acceleration that we get is related to the execution onto one CPU core. So S is defined as the speed-up obtained in respect to a fraction of the CPU code.

As shown in table 2, applying the Amdahl's law, we can estimate the maximum acceleration we could get in the algorithm after accelerating a fraction of the algorithm by a factor S. Other implicit latencies exist in the architecture that will be discussed in the following sections. The acceleration of the winning neurons search and Euclidean distance stages offers the highest overall acceleration.

In the experiments section, real values for speed-up of winning neuron search stage will be obtained. Then we will apply Amdahl's law again to compare theoretical values with real overall speed-up obtained using the GNG algorithm. Thereby we will be able to measure how much time is consumed by other latencies like data transfers or device initialization and what is the speed-up upper bound for GNG algorithm.

3.3. GPU Implementation

In order to accelerate the GNG algorithm on GPUs using CUDA, it is necessary to redesign it so that it fits within the GPU architecture. Many of the oper-

Neurons	Patterns λ	s	p	Overall speed-up
1000	500	0,28	0,73	**3,29**
5000	500	0,11	0,89	**6,39**
10000	500	0,07	0,93	**8,73**
20000	500	0,02	0,98	**13,74**
1000	1000	0,30	0,67	**2,95**
5000	1000	0,1	0,9	**6,89**
10000	1000	0,06	0,94	**9,60**
20000	1000	0,04	0,97	**12,01**

Table 2: Overall maximum acceleration estimated using Amdahl's law and assuming a factor 20 of speed-up in respect to a fraction p of the algorithm

ations performed in the GNG algorithm can be parallelized because they act on all the neurons of the network simultaneously. That is possible because there is no direct dependence between neurons at the operational level. However, there exists a dependence in the adjustment of the network, which makes necessary the synchronization of various parallel execution operations each iteration. Figure 2 describes GNG algorithm steps that have been accelerated onto the GPU using kernels.

3.3.1. Euclidean distance calculation

The first stage of the algorithm that has been accelerated is the calculation of Euclidean distances performed at each iteration. This stage calculates the Euclidean distance between a random pattern and each of the neurons. This task may take place in parallel by running the calculation of each neuron distance onto as many threads as neurons the network contains. It is possible to calculate more

than one distance per thread, but this is efficient only for large vectors where the number of blocks executed on the GPU is also very high.

3.3.2. Parallel reduction

The second task parallelized was the search of the winning neuron: the one with the lowest Euclidean distance to the pattern generated and the second closest. For this search, we use a parallel reduction technique described in [52]. This technique accelerates parallel operations such as the search for the minimum in large data sets. For our work, we modified the original algorithm, so that with a single reduction we not only obtained the minimum, but also the two smallest values of the entire data set. This new version has been called $2MinParallelReduction$. Figure 3 shows how Parallel Reduction can be described as a binary tree where at the end of the $\log_2(n)$ steps we obtain the final result of the operation onto a set of N elements.

3.3.3. Complexity

To perform the calculation of the complexity of this approach comparing it with the sequential version that has a complexity of $O(N)$, it should be noted that, in parallel processing, we identified three types of complexity: complexity in the number of execution steps, complexity of the work performed and time complexity. These complexities in the parallel reduction algorithm are:

- The execution steps complexity is $O(\log_2(N))$ since it is necessary to perform $\log_2(N)$ iterations to reach the final result. Also within each execution step s, $\frac{N}{2^s}$ operations are performed.
- The complexity of the work performed is: If for $N = 2d$ elements, $\sum_{i=1}^{S} 2^{(d-i)} =$

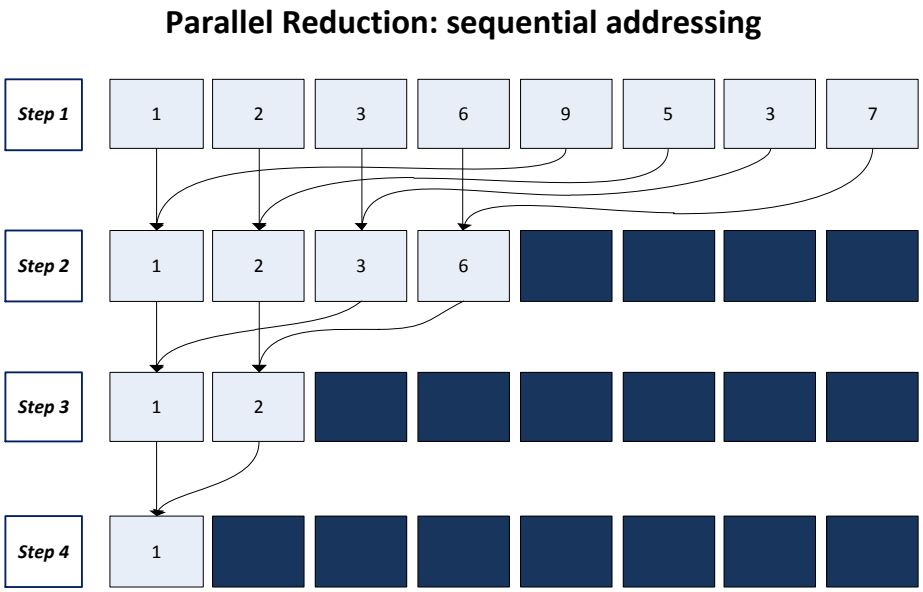

Figure 3: Example Parallel Reduction Algorithm execution.

$N-1$ operations are performed, the complexity of the work done is $O(N)$, where S is the total number of steps.

- The time complexity is $O(N/P + \log_2(N))$, where P is the number of processors.

Therefore, since in each block t threads are executed, each of them processing each element of the set N, we have a number of threads equal to the number of elements. Considering this to calculate the time complexity of each of these threads as processors, the time complexity is reduced to $\log_2(N)$ in respect to the complexity $O(N)$ in the sequential version.

Despite this difference in complexity between the parallel and the sequential versions, the preparation and execution of programs on the GPU involves a time penalty, as well as the GPU memory transfer of data processing that causes a new penalty that begins to be compensated from a number X of elements to be processed. This issue also affects the cost of the operation to be performed onto the data.

3.3.4. Other optimizations

To speed-up the remaining steps we have followed the same strategy used during the first phase. Each thread is responsible in different cases to perform an operation on a neuron: check edges connections age and in the case a certain threshold was exceded delete them, update local error of the neuron or adjust neuron weights. At the stage of finding the neuron with maximum error the strategy followed was the same as the one used in finding the winning neuron, but in this case the reduction is looking only for the neuron with the highest error.

Regardless of the parallelism of the algorithm, we have followed some good practices on the CUDA architecture to get more performance. First, the use of the constant memory to store the neural network parameters ϵ_w, ϵ_n, α, γ, a_{max}. By storing these parameters in this memory, the access is faster than working with values stored in the global memory.

Our GNG implementation onto GPU architecture is also limited by the memory bandwidth available. In the experiments section we show specification reports for each CUDA capable device used and its memory bandwidth. However, this bandwidth is only achievable under highly idealized memory access patterns. It does, however, provide us with an upper limit of memory performance. Nevertheless, some memory access patterns, like moving data from the global memory

into shared memories and registers, provide better coalesced access. The shared memory within each multiprocessor has been used to get the highest advantage of memory bandwidth. So that, it acts as a cache to avoid frequent access to global memory in operations with neurons and allows the threads to achieve coalesced reads when accessing neurons data.

For instance, a GNG network composed of 20,000 neurons and auxiliary structures requires only 17 megabytes. Therefore, GPU implementation in terms of size does not present problems as current GPU devices have enough memory to store it.

3.3.5. Minimizing transfers approach

Memory transfers between CPU and GPU are the main bottleneck to obtain speed-up, so these transfers have been avoided as much as possible. Initial versions of the algorithm failed to obtain performance over the CPU version because the complete neural network was copied from GPU memory to CPU memory and vice versa for each input pattern generated. This penalty, introduced due to the bottleneck of the transfer through the PCI-Express bus, was so high that we did not improve CPU version. After careful consideration of the flow of execution, we decided to move the inner loop of pattern generation to the GPU, although some tasks are not parallelizable and have to be run on a single GPU thread.

The workflow of our first approach using CUDA is shown in figure 4. First, GNG network is created in the CPU and CUDA device is initialized. Then, the necessary space is allocated in the GPU memory to perform processing. Once the GNG network structure has been copied to the GPU memory, the learning algorithm begins: first, a random input pattern is generated and the Euclidean distance is calculated from each of the neurons. Second, these distances are calculated in

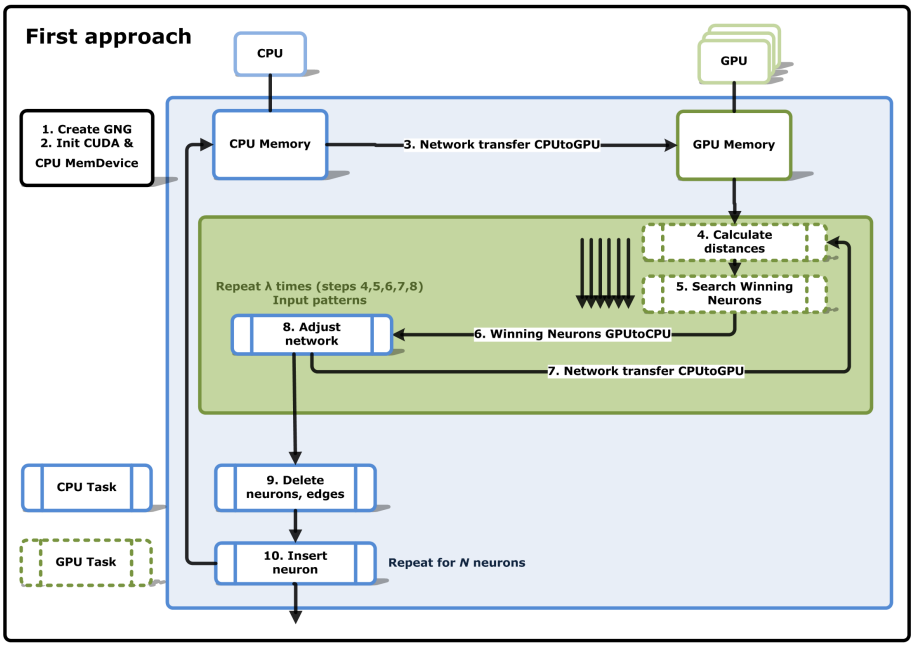

Figure 4: First approach, CUDA workflow.

parallel taking advantage of the massively parallel computing on the GPU and, the two neurons with the lowest distance (winning neurons) are also obtained using a parallel reduction. Then, the indexes of winning neurons are copied to the CPU memory and adjustment is performed (sequential). This step is repeated λ times. This first approach did not obtain improvement regarding of the CPU version because the entire neural network were copied from the CPU memory to the GPU λ times at each new neuron insertion, including significant latencies.

Figure 5 shows the approach used to avoid this large number of transfers between GPU and CPU memory. The inner loop has been moved to the GPU, so it is not necessary to copy the network structure back to the memory of the CPU and make the adjustment. In this case, the adjustment is performed in a single thread onto the GPU because the task set is sequential and can not be parallelized.

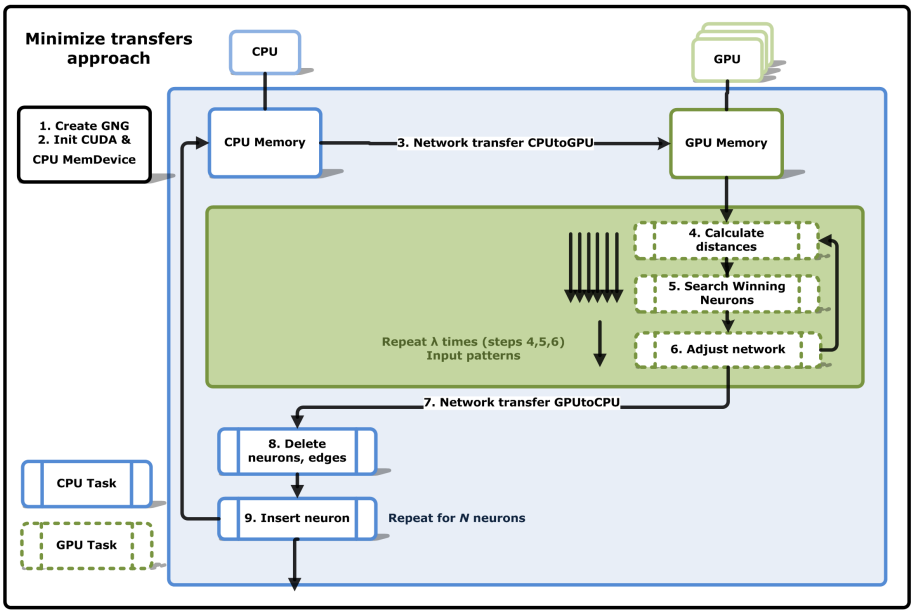

Figure 5: Approach that minimize transfers. CUDA workflow.

Performing this task in a single thread in the GPU is better due to the high latency of transfers between GPU and CPU. It is also clear that reducing the number of memory transactions from the device memory results in a significant increase of the processing throughput.

Figure 6 shows that, for 500 patterns, the percentage of time spent in the execution of the algorithm for memory transfers between CPU and GPU is drastically reduced. Thus we can increase the number of input patterns without increasing the number of transfers between memories.

The use of CUDA in this algorithm provides better performance for a large number of neurons due to the time needed to prepare some specific guidelines for the architecture implementation as kernels execution or GPU memory allocation. Performing these operations on small vectors of 50-500 neurons is almost

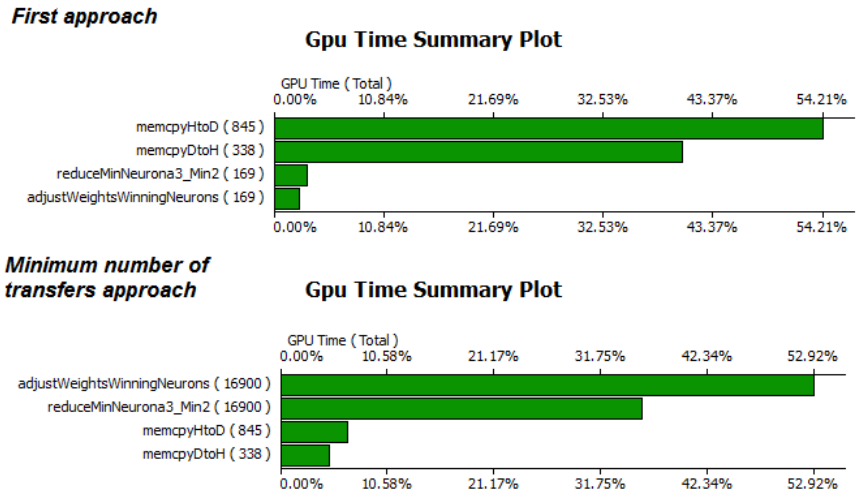

Figure 6: Percentage of time spent in execution of the algorithm for memory transfers.

immediate on the CPU, while the GPU cannot hide these inherent latencies in the architecture if a large number of neurons is not reached. Therefore, we considered the idea of applying hybrid techniques according to the restriction that the GNG is an incremental network that initially works with a small number of neurons, which grows progressively. This hybrid technique begins by running the GNG onto the CPU, but when it is detected that the runtime of the sequential version is higher than the runtime of the parallelized one, the network is copied to GPU memory and the remaining calculation is performed onto the GPU.

4. Experiments

The accelerated version of GNG algorithm has been developed and tested on a machine with an Intel Core i3 540 3.07Ghz and different CUDA capable devices. Table 3 shows different models that we have used and their features.

The multi-core CPU implementation of the GNG algorithm has been developed using Intel Threading Building Blocks (TBB) library [53], taking advantage of the multi-core processor capabilities and avoiding the existing overhead [54]. The number of threads used in the multi-core CPU implementation is the maximum defined in the specifications of Intel i3 540 processor.

Device Model	Capability	SMs	cores per SM	Global Mem	Bandwidth Mem
Quadro 2000	2.1	4	192	1 GB	41.6 GB/s
GeForce GTX 480	2.0	15	480	1.5 GB	177.4 GB/s
Tesla C2070	2.0	14	448	6 GB	144 GB/s

Table 3: CUDA capable devices used in experiments

First, we have performed some experiments to obtain the best parameters to launch kernels in our application, obtaining the maximum occupancy of CUDA multiprocessors. Once we obtained the best parameters we used them to test the *2minimum Parallel Reduction* implementation obtaining different performance depending on the graphic card used and its number of cores. Finally, we have done some experiments to obtain the speed-up regarding single threaded and multi-threaded CPU versions.

4.1. Number of threads per block

As mentioned in section 2, threads are organized into blocks to carry out their execution onto multiprocessors. Depending on the application developed, a different number of threads per block should be used to obtain the best performance. We tested different kernels running on the NVIDIA GTX 480 with different numbers of threads per block. The better performance is obtained when using a number of

threads between 128 and 256 to perform this test (figure 7). This is because using these parameters, we obtain maximum occupancy of CUDA multiprocessors. These results are directly applied to the other cards since the number of threads per block depends on the kind of application designed.

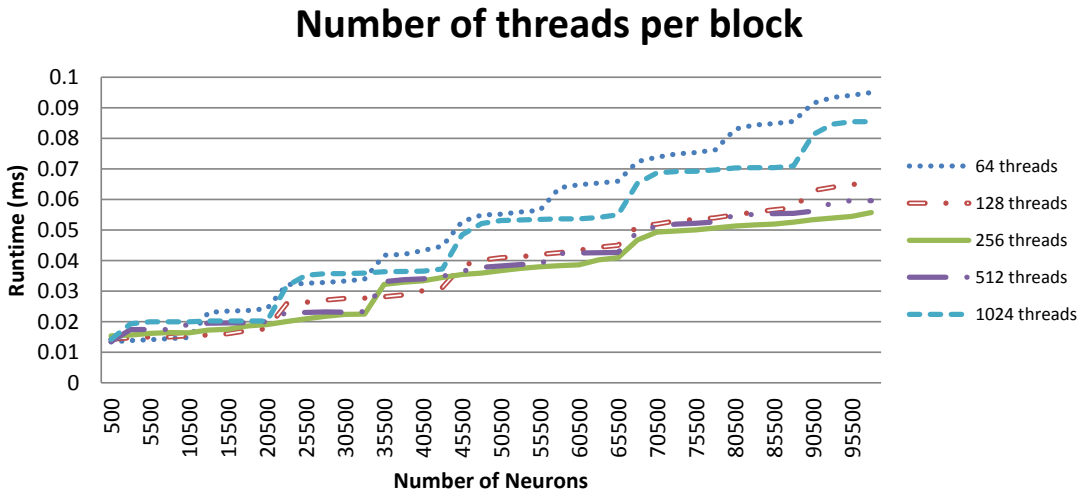

Figure 7: Execution time depending on the number of threads per block.

4.2. Speed-up 2 Min Parallel Reduction

We have made some experiments of *2Min Parallel Reduction* implementation with different graphics boards using 256 threads per block configuration for kernels launch. We obtained a speed-up factor up to 43x faster regarding a single-core CPU and 40x faster regarding multi-core CPU, in the task of taking adjustments of the network with a number of neurons close to 100k. As we can see in figure 8 (bottom), the speed-up factor depends on the device on which we execute the algorithm and the number of cores it has. Figure 8 (top) shows the evolution of the execution time in sequential Reduction operation regarding the parallel version. It

can be also appreciated how it improves the acceleration provided by the parallel version as the number of elements grows.

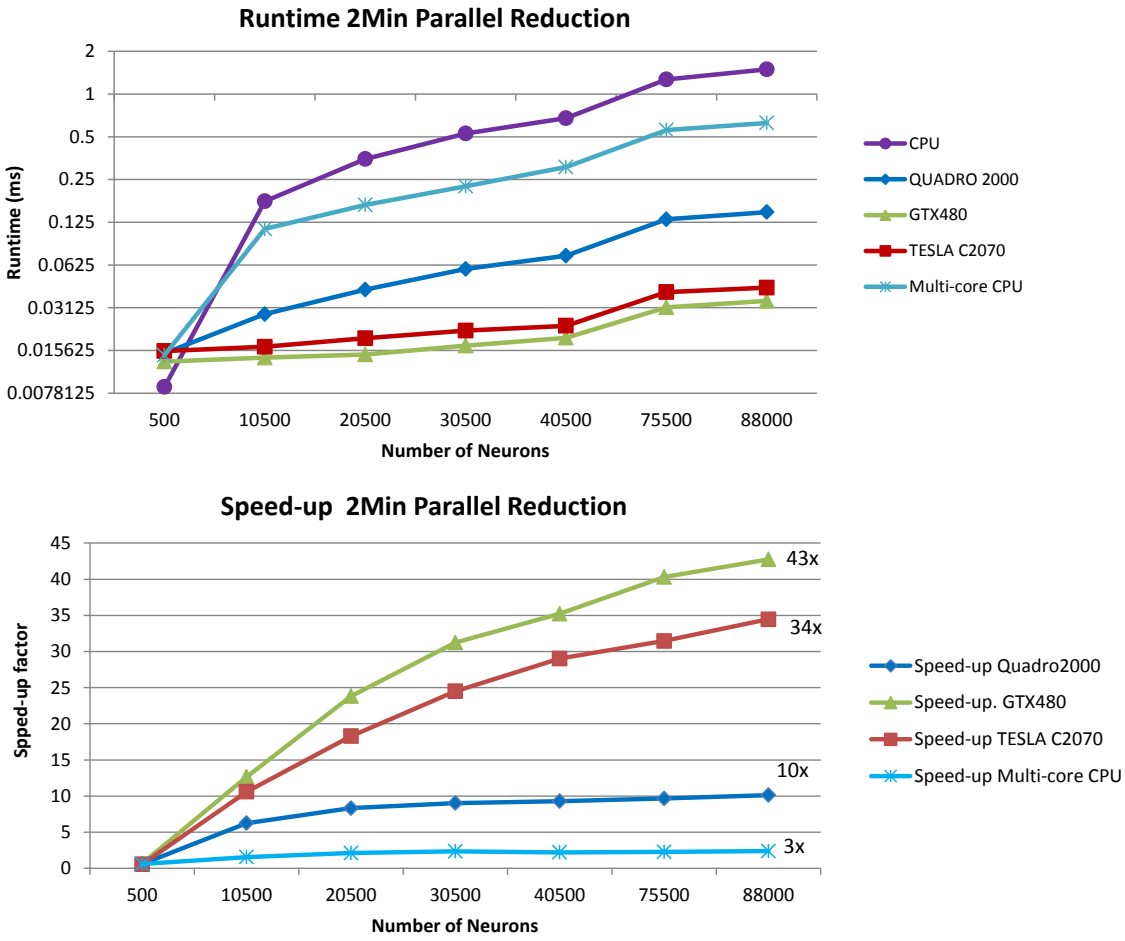

Figure 8: Speed-up of Parallel Reduction with different graphic cards.

Best results were obtained with the most powerful graphic card tested, in our case the NVIDIA GTX 480 with 480 cores and 1,5 GBytes of Memory. Using conventional GPUs such as the Quadro 2000 model, which are found in desktop computers, we can also obtain a good speed-up, 10x-15x faster than CPU.

If we apply these values to Amdahl's law that we previously analyzed, replacing speed-up factor of fraction f by these values, we can estimate what will be the upper limit of speed-up we can obtain for applying the whole GNG algorithm. Thereby, the current acceleration will be compared with the calculated upper limit and we can extract what percentage of time is consumed by other latencies implied in the CUDA architecture.

Table 4 shows overall speed-up obtained for different parameters of GNG using the speed-up we obtained accelerating stage 3 of algorithm.

Neurons	Patterns λ	s	p	Speed-up	Overall speed-up
1000	500	0,27	0,73	0,78	**0,83**
5000	500	0,11	0,89	2,95	**2,42**
10000	500	0,07	0,93	5,61	**4,27**
20000	500	0,02	0,98	10,60	**8,62**
1000	1000	0,30	0,70	0,69	**0,761**
5000	1000	0,1	0,9	2,9	**2,44**
10000	1000	0,05	0,94	5,65	**4,47**
20000	1000	0,03	0,96	10,68	**7,98**

Table 4: Theoretical overall speed-up obtained for GNG algorithm using speed-up obtained in stage 3. Device GTX 480.

4.3. 2D representation and 3D reconstruction

To test our parallel version of the GNG algorithm, we have done experiments using the GNG for 2D representation and 3D reconstruction. To solve the problem of 3D reconstruction, the number of neurons necessary to adapt the input space is

high which benefits the use of the GPU. Therefore, an increase in speed over the CPU version can be achieved.

Based on a previous work [55], it has been chosen a number of neurons N of 1000 / 5000 / 10000 / 200000 and a number of input patterns λ of 500/1000. Other parameters have been also fixed based on our previous experience: $\epsilon_w = 0.1$, $\epsilon_n = 0.001$, $\alpha = 0.5$, $\gamma = 0.95$, $a_{max} = 250$.

GNG learning speed-up factor

Figure 9 shows an experiment using GNG to reconstruct a 3D object with 20000 neurons and 1000 input patterns where CPU solution takes more and more time as the number of neurons in the network grows. However, the parallel CUDA version increases the size of the array of neurons without degrading significantly the performance. For a number of 20k neurons, we obtain 6x speed-up factor using a NVIDIA GTX 480 GPU. This speed-up is lower than the theoretical overall speed-up that we estimated in the previous section. This is due to the implicit latencies of the GPU architecture. Table 5 shows differences between the theoretical overall speed-up and the obtained overall speed-up.

Neurons	Patterns	Theoretical overall speed-up	Real overall speed-up
1000	1000	0,76	0,68
5000	1000	2,44	2,36
10000	1000	4,47	4,23
20000	1000	7,98	6,11

Table 5: Theoretical overall speed-up and obtained overall speed-up using the device GTX480.

In figure 9, it can also be appreciated that the CPU version is faster during the first iterations, so a hybrid version would be faster than separate CPU and

GPU versions. Multi-core CPU implementation is also slower during the first iterations compared with single-core CPU due to the existing overhead caused by the management of threads and by the subdivision of the problem.

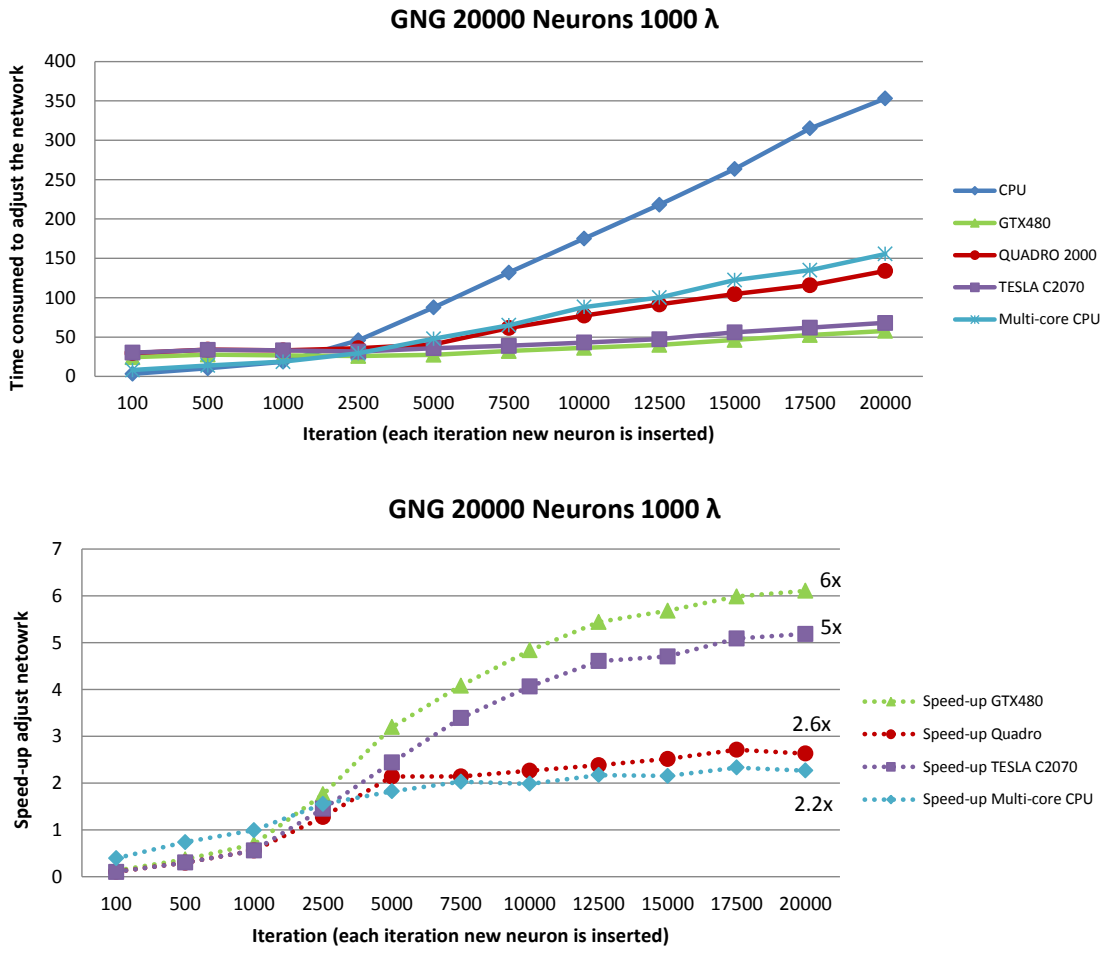

Figure 9: Example of GPU and CPU GNG runtime, and speed-up for different devices.

4.4. GNG hybrid version

As we discussed in the previous experiments, GPU version has low performance in the first iterations of the learning algorithm, where the GPU can not hide the latencies due to the small number of processing elements. To achieve even bigger acceleration of the GNG algorithm, we propose the use of the CPU in the first iterations of the algorithm, and then start processing data in the GPU only when there is an acceleration regarding CPU, thus achieving a bigger overall acceleration of the algorithm (see figure 10). To determine the number of neurons necessary to start computing at GPU we have analyzed in detail the execution times for each new insertion, and concluded that each device, depending on its computing power starts being efficient at a different number of neurons. After several tests, we have determined the threshold at which each device starts accelerating regarding the CPU version. As it can be seen in figure 9 (top), threshold values for different devices are set to 1500, 1700, 2100 for GTX 480, Tesla C2070 and Quadro 2000 models respectively. The hybrid version is proposed as some applications need to operate under time constraints obtaining a solution of a specified quality within certain period of time. In cases when the objective is the disruption of learning due to the application requirements, it is important to insert the maximum number of neurons and perform the maximum number of adjustments to achieve the highest quality in a limited time. The hybrid version ensures a maximum performance in this kind of applications using the computational capabilities of the CPU or the GPU depending on the situation. For example, our proposal has been validated on 3D scene reconstruction performed by a mobile robot. In this application it is necessary to obtain a reconstruction of the scene in a limited time.

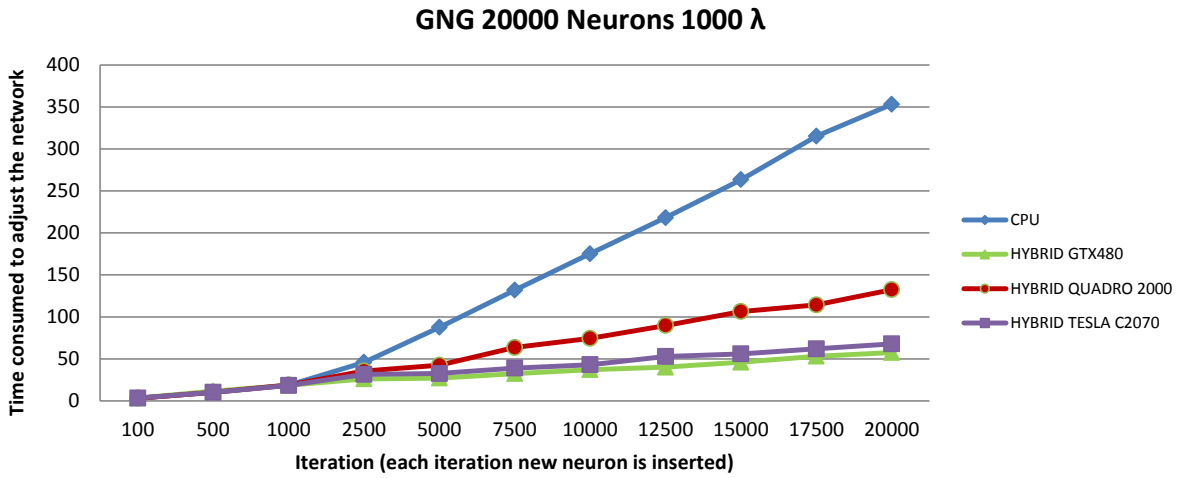

Figure 10: Example of CPU and Hybrid GNG runtime for different devices.

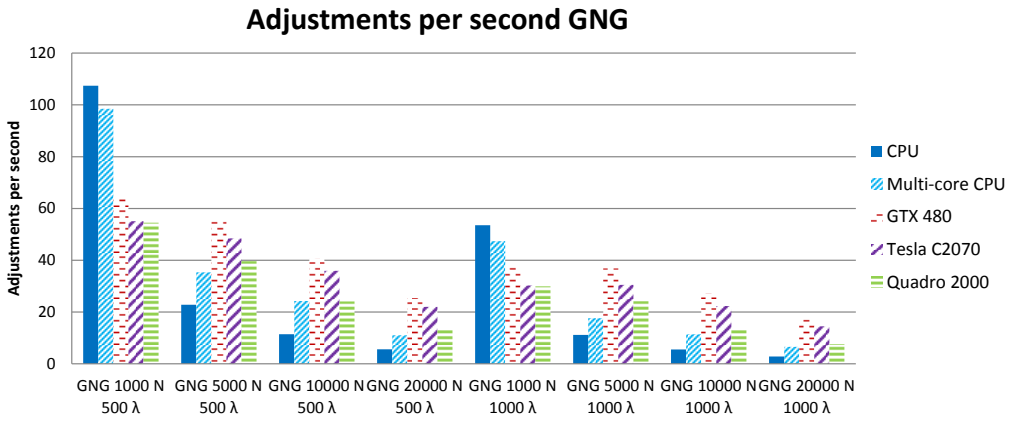

Figure 11: Rate of adjustments per second performed by different GPU devices and CPU.

4.5. Rate of adjustments per second

We have performed several experiments that show how the accelerated version of the GNG is not only capable of learning faster than CPU, bu also obtains more adjustments per second than the single-threaded and multi-threaded CPU

implementations. For instance, after learning a network of 20000 neurons we can perform 17 adjustments per second using the GPU while the single-core CPU gets 2.8 adjustments per second and the multi-core CPU gets 8 adjustments per second. This means that GPU implementation can obtain a good topological representation with time constraints. Figure 11 shows the different adjustments rates per second performed by different GPU devices compared to CPU. It is also shown that when increasing the number of neurons in the CPU, it can not handle a high rate of adjustments per second.

4.6. Discussion

From the experiments described above we can conclude that the number of threads per block that best fits in our implementation is 256 due to the following reasons: First, the amount of computation the algorithm performs in parallel. Second, the number of resources that each device has and finally the use that we have made of shared memories and registries. It is also demonstrated that in comparison to CPU implementation, the $2MinParallelReduction$ achieves a speed-up of more than 40x to find out a neuron at a minimum distance to the generated input pattern. Theoretical values obtained applying Amdahl's law and its comparison with real values obtained from the experiments indicates that GPGPU architecture has some implicit latencies: initialization time, data transfers time, memory access time, etc.

Experiments on the complete GNG algorithm showed that using the GPU, small networks under-utilize the device, since only one or a few multiprocessors are used. Our implementation has a better performance for large networks than for small ones. To get better results for small networks we propose a hybrid implementation. These results show that GNG learning with the proposed hybrid

implementation achieves a speed-up 6 times higher than the single threaded CPU implementation.

Finally, it is shown how our GPU implementation can process up to 17 adjustments of the network per second while single threaded CPU implementation only can manage 2.8, getting a speed-up factor of more than 6 times.

5. Accelerating 6DoF egomotion using GNG

In this section, we show an application where the use of the accelerated GNG improves its solution. The main goal of this application is to perform six degrees of freedom (6DoF) pose registration in semi-structured environments, i.e., man-made indoor and outdoor environments. This registration can provide a good starting point for Simultaneous Location and Mapping (SLAM). We use the method proposed in [31]. This method is developed for managing 3D point sets collected by any kind of sensor. For our experiments, we have used data from an infrared time-of-flight camera SR4000, but in [31] there are examples of this method applied to other 3D devices, like a sweeping unit with a 2D laser Sick and a Digiclops stereo camera, mounted on a mobile robot. We are also interested in dealing with outliers, i.e., environments with people or non-modeled objects. This task is hard to overcome as classic algorithms, like ICP and its variants, are very sensitive to outliers. Furthermore, we do not use odometry information. Finally, the huge amount of data makes necessary the acceleration of the overall process in order to obtain the results in real time.

We briefly describe the method proposed in [31] to manage 3D data and to use it for 6DoF egomotion calculation. GNG produces a Delaunay Triangulation which can be used as a representation of the points neighbourhood. GNG can be

applied directly to 3D data. Figure 12 shows the result of applying GNG to 3D points from a SR4000.

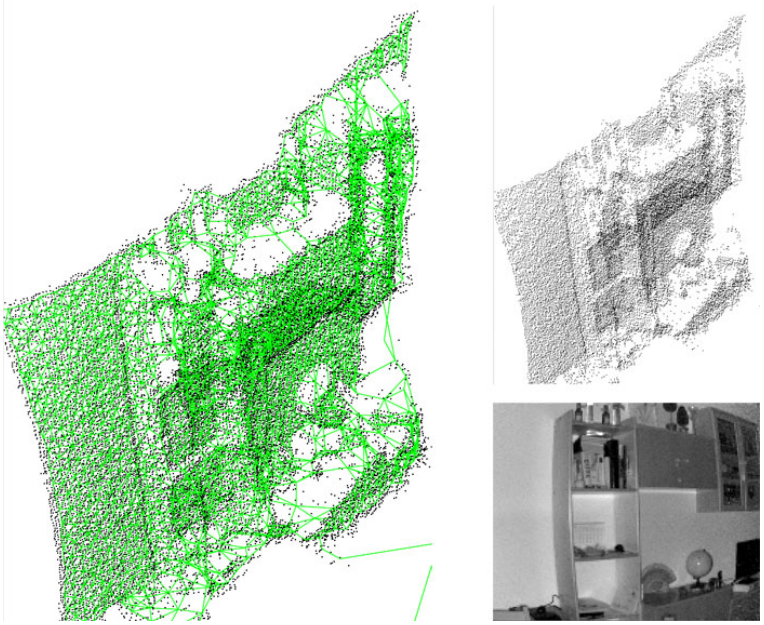

Figure 12: Applying GNG to SR4000 data.

On the other hand, in [31] a feature extraction process is applied to the raw 3D data in order to obtain a complexity reduction. These features are planar patches which are models representing surfaces from the 3D data. This feature extraction method is based on neighbour searching. We can improve and accelerate the neighbour searching using the GNG structure as it produces a more detailed and accurate planar patches descriptions. Figure 13 shows planar patches extraction from a 3D image obtained by a SR4000 camera. The right image shows the results of combining GNG with the features extraction procedure. It can be compared with the left image in which no GNG has been used. The more number of planar patches we have, the more accurate result we obtain.

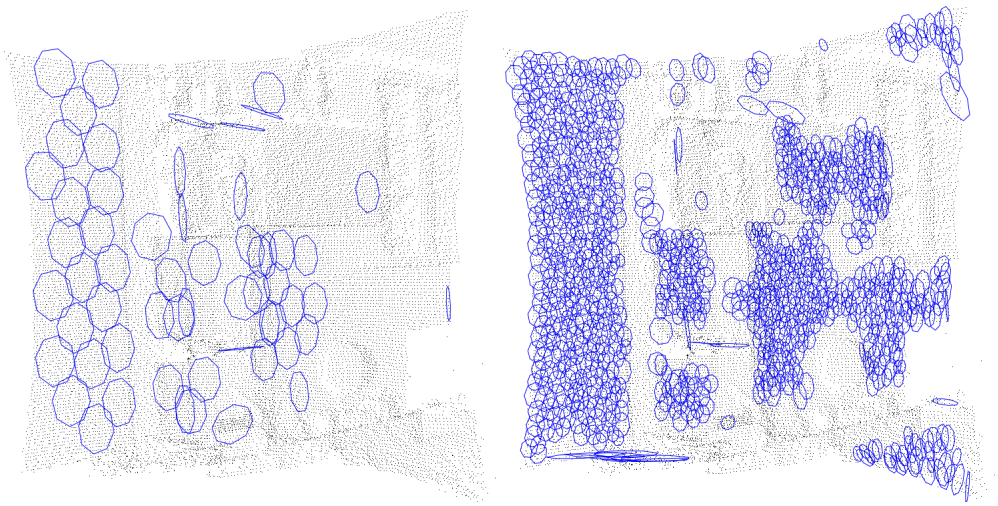

Figure 13: Left, planar patches extracted from SR4000 camera. Right, use of GNG to improve planar patches extraction.

For this reason, we would like to use these models to achieve further mobile robot applications in real 3D environments. The basic idea is to take advantage of the extra knowledge that can be found in 3D models such as surfaces and its orientations. This information is introduced in a modified version of an ICP-like algorithm in order to reduce the outliers incidence in the results. ICP [56] is widely used for geometric alignment of a pair of three-dimensional points sets. From an initial approximate transformation, ICP iterates the next three steps until convergence is achieved: first, closest points between sets are stated; then, best fitting transformation is computed from paired points; finally, transformation is applied. In mobile robotics area, the initial transformation usually comes from odometry data.

Nevertheless, our approach does not need an initial approximate transformation as ICP based methods do. We can use the global model structure to recover

the correct transformation. This feature is useful for those situations where no odometry is available, or it is not accurate enough, such as legged robots. In our case, we exploit both the information given by the normal vector of the planar patches and its geometric position. Whereas original ICP computes both orientation and position at each iteration of the algorithm, we can take an advantage of the knowledge about planar patches orientation for decoupling the computation of rotation and translation. We first register the orientation of planar patches sets and when the two planar patches sets are aligned we address the translation registration.

Figure 14: Planar based 6DoF egomotion results. Left image shows map building results without using GNG while the results shown on the right are obtained after computing a GNG mesh.

In figure 14, we show an example of 3D map building using this 6DoF egomo-

tion approach. For this experiment, 100 3D images from a 5 meter range SR4000 camera were used. The image on the left shows a 3D view of the reconstructed environment using 6DoF egomotion from planar patches. In the right image, the same scene is reconstructed but GNG was used to improve feature extraction. While in the first experiment the registration of the sequence was almost impossible, in the second one the reconstruction was reasonably good. Computing time for obtaining planar patches descriptions after applying GNG is almost the same as without GNG and is about 300 ms per image. Application of GPU acceleration provides a lower reconstruction time per each data acquisition, 50 ms for an adjustment of a neural network composed by 20.000 neurons and 1000 λ input patterns as it can be seen in figure 11. This makes our system suitable to deal with time constraints.

6. Conclusions and future work

This paper proposes the modification and acceleration of the GNG algorithm in order to obtain a more efficient version suitable for operations with time constraints. As demonstrated in the experiments, the runtime of sequential GNG algorithm grows with the number of neurons as the network increases. In contrast, in the parallel version implemented onto GPU architecture, as we increase the number of neurons, we obtain a greater acceleration over the sequential version. Experimental results show that the GPU implementation significantly reduces learning time compared with single-threaded and multi-threaded CPU implementations for GNG.

GNG algorithm can be accelerated using the GPU and allows better performance than the CPU implementations. It has also been demonstrated how 3D

scene reconstruction for mobile robotics can be accelerated using GPUs in order to deal with time constraints.

The parallel solution implemented on GPU can be still improved carefully analyzing all aspects offered by the CUDA architecture and making a better use of them: multiprocessors occupancy, memory hierarchy use, transfer between CPU and GPU memory, and other.

Further work will include other improvements on the GPU implementation: generating random patterns using GPU and using multi-GPU computation to improve performance and to manage several neural networks learning different features simultaneously. More applications of the accelerated GNG will be studied in the future.

Acknowledgments

This work has been supported by grant DPI2009-07144 from Ministerio de Ciencia e Innovacion of the Spanish Government, by the University of Alicante projects GRE09-16 and GRE10-35, and Valencian Government project GV/2011/034. Experiments were made possible with a generous donation of hardware from NVDIA.

References

[1] S. Furao, O. Hasegawa, An incremental network for on-line unsupervised classification and topology learning, Neural Netw. 19 (2006) 90–106.

[2] T. Kohonen, Self-Organising Maps, Springer-Verlag, 2001.

[3] B. Fritzke, A Growing Neural Gas Network Learns Topologies, Vol. 7, MIT Press, 1995, pp. 625–632.

[4] J. Garcia-Rodriguez, F. Florez-Revuelta, J. M. Garcia-Chamizo, Image compression using growing neural gas, in: Proc. Int. Joint Conf. Neural Networks IJCNN 2007, 2007, pp. 366–370.

[5] F. Florez-Revuelta, J. M. Garcia-Chamizo, J. Garcia-Rodriguez, A. Hernandez-Saez, Representation of 2d objects with a topology preserving network, in: J. M. I. Quereda, L. Micó (Eds.), PRIS, ICEIS Press, 2002, pp. 267–276.

[6] J. Rivera-Rovelo, S. Herold, E. Bayro-Corrochano, Object segmentation using growing neural gas and generalized gradient vector flow in the geometric algebra framework, in: J. Martnez-Trinidad, J. Carrasco Ochoa, J. Kittler (Eds.), Progress in Pattern Recognition, Image Analysis and Applications, Vol. 4225 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 306–315, 10.1007/11892755-31.

[7] Y. Wu, Q. Liu, T. S. Huang, An adaptive self-organizing color segmentation algorithm with application to robust real-time human hand localization, in: in Proc. of Asian Conference on Computer Vision, 2000, pp. 1106–1111.

[8] A. Angelopoulou, A. Psarrou, J. Garcia-Rodriguez, Robust modelling and tracking of nonrigid objects using active-gng, in: Proc. IEEE 11th Int. Conf. Computer Vision ICCV 2007, 2007, pp. 1–7.

[9] X. Cao, P. N. Suganthan, Hierarchical overlapped growing neural gas networks with applications to video shot detection and motion characterization, in: Proc. Int. Joint Conf. Neural Networks IJCNN '02, Vol. 2, 2002, pp. 1069–1074.

[10] H. Frezza-Buet, Following non-stationary distributions by controlling the vector quantization accuracy of a growing neural gas network, Neurocomput. 71 (2008) 1191–1202.

[11] H.-J. Boehme, A. Brakensiek, U.-D. Braumann, M. Krabbes, H.-M. Gross, Neural networks for gesture-based remote control of a mobile robot, in: Proc. IEEE Int. Joint Conf. IEEE World Congress Computational Intelligence Neural Networks, Vol. 1, 1998, pp. 372–377.

[12] F. Florez, J. M. Garcia, J. Garcia, A. Hernandez, Hand gesture recognition following the dynamics of a topology-preserving network, in: Proc. Fifth IEEE Int Automatic Face and Gesture Recognition Conf, 2002, pp. 318–323.

[13] J. Garcia-Rodriguez, A. Angelopoulou, A. Psarrou, Growing neural gas (gng): A soft competitive learning method for 2d hand modelling, IEICE - Trans. Inf. Syst. E89-D (2006) 2124–2131.

[14] A.-M. Cretu, E. M. Petriu, P. Payeur, Evaluation of growing neural gas networks for selective 3d scanning, in: Proc. Int. Workshop Robotic and Sensors Environments ROSE 2008, 2008, pp. 108–113.

[15] R. do Rego, A. Araujo, F. de Lima Neto, Growing self-organizing maps for surface reconstruction from unstructured point clouds, in: Neural Networks, 2007. IJCNN 2007, 2007, pp. 1900 –1905.

[16] Y. Holdstein, A. Fischer, Three-dimensional surface reconstruction using meshing growing neural gas (mgng), Vis. Comput. 24 (2008) 295–302.

[17] W.-m. W. Hwu, GPU Computing Gems Emerald Edition, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[18] J. Nickolls, W. J. Dally, The gpu computing era, IEEE Micro 30 (2010) 56–69.

[19] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore gpus, in: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IPDPS '09, 2009, pp. 1–10.

[20] D. R. Horn, J. Sugerman, M. Houston, P. Hanrahan, Interactive k-d tree gpu raytracing, in: Proceedings of the 2007 symposium on Interactive 3D graphics and games, I3D '07, 2007, pp. 167–174.

[21] CUDA Programming Guide, version 3.2, (2010).

[22] I. Ivrissimtzis, W.-K. Jeong, H.-P. Seidel, Using growing cell structures for surface reconstruction, in: Shape Modeling International, 2003, pp. 78 – 86.

[23] M. I. Fanany, I. Kumazawa, A neural network for recovering 3d shape from erroneous and few depth maps of shaded images, Pattern Recogn. Lett. 25 (2004) 377–389.

[24] S. May, D. Droeschel, D. Holz, S. Fuchs, E. Malis, A. Nüchter, J. Hertzberg, Three-dimensional mapping with time-of-flight cameras, J. Field Robot. 26 (2009) 934–965.

[25] D. Hähnel, W. Burgard, S. Thrun, Learning compact 3D models of indoor and outdoor environments with a mobile robot, Robotics and Autonomous SystemsTo appear.

[26] G. Grisetti, S. Grzonka, C. Stachniss, P. Pfaff, W. Burgard, Efficient estimation of accurate maximum likelihood maps in 3d, in: In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS, 2007.

[27] J. W. Weingarten, G. Gruener, R. Siegwart, Probabilistic plane fitting in 3d and an application to robotic mapping., in: ICRA, IEEE, 2004, pp. 927–932.

[28] R. B. Rusu, Z. C. Marton, N. Blodow, M. Beetz, Learning informative point classes for the acquisition of object model maps, 2008, pp. 643–650.

[29] B. Steder, G. Grisetti, M. V. Loock, W. Burgard, Robust on-line model-based object detection from range images., in: IROS, IEEE, 2009, pp. 4739–4744.

[30] B. Steder, G. Grisetti, W. Burgard, Robust place recognition for 3d range data based on point features., in: ICRA, IEEE, 2010, pp. 1400–1405.

[31] D. Viejo, J. Garcia, M. Cazorla, D. Gil, M. Johnsson, Using 3d gng-based reconstruction for 6dof egomotion, in: Neural Networks (IJCNN), The 2011 International Joint Conference on, 2011, pp. 1042 –1048.

[32] I. Villaverde, M. Graña, An improved evolutionary approach for egomotion estimation with a 3d tof camera, in: J. Mira, J. Ferrndez, J. lvarez, F. de la Paz, F. Toledo (Eds.), Bioinspired Applications in Artificial and Natural Computation, Vol. 5602 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2009, pp. 390–398.

[33] R. Muñoz Salinas, E. Aguirre, M. Garcia-Silvente, A. Gonzalez, A multiple object tracking approach that combines colour and depth information using a confidence measure, Pattern Recognition Letters 29 (10) (2008) 1504 – 1514.

[34] R. Katz, J. Nieto, E. Nebot, Unsupervised classification of dynamic obstacles in urban environments, J. Field Robot. 27 (2010) 450–472.

[35] D. B. Kirk, W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[36] H. Jang, A. Park, K. Jung, Neural network implementation using cuda and openmp, in: Proc. DICTA '08.Digital Image Computing: Techniques and Applications, 2008, pp. 155–161.

[37] K. Oh, GPU implementation of neural networks, Pattern Recognition 37 (6) (2004) 1311–1314.

[38] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, A. Veidenbaum, Efficient simulation of large-scale spiking neural networks using cuda graphics processors, in: Proc. Int. Joint Conf. Neural Networks IJCNN 2009, 2009, pp. 2145–2152.

[39] C.-F. Juang, T.-C. Chen, W.-Y. Cheng, Speedup of implementing fuzzy neural networks with high-dimensional inputs through parallel processing on graphic processing units, IEEE T. Fuzzy Systems 19 (4) (2011) 717–728.

[40] J. Garcia-Rodriguez, A. Angelopoulou, V. Morell, S. Orts, A. Psarrou, J. M. Garcia-Chamizo, Fast image representation with gpu-based growing neural gas, in: IWANN (2), 2011, pp. 58–65.

[41] J. Igarashi, O. Shouno, T. Fukai, H. Tsujino, 2011 special issue: Real-time simulation of a spiking neural network model of the basal ganglia circuitry

using general purpose computing on graphics processing units, Neural Netw. 24 (2011) 950–960.

[42] R. Uetz, S. Behnke, Large-scale object recognition with cuda-accelerated hierarchical neural networks, in: Proc. IEEE Int. Conf. Intelligent Computing and Intelligent Systems ICIS 2009, Vol. 1, 2009, pp. 536–541.

[43] F. Nasse, C. Thurau, G. Fink, Face detection using gpu-based convolutional neural networks, in: X. Jiang, N. Petkov (Eds.), Computer Analysis of Images and Patterns, Vol. 5702 of LNCS, Springer, 2009, pp. 83–90.

[44] S. Oh, K. Jung, View-point insensitive human pose recognition using neural network and cuda, World Academy of Science 60.

[45] T. M. Martinetz, S. G. Berkovich, K. J. Schulten, 'neural-gas' network for vector quantization and its application to time-series prediction 4 (4) (1993) 558–569.

[46] B. Fritzke, Growing cell structures - a self-organizing network for unsupervised and supervised learning, Neural Networks 7 (1993) 1441–1460.

[47] T. Martinetz, Competitive Hebbian Learning Rule Forms Perfectly Topology Preserving Maps, in: S. Gielen, B. Kappen (Eds.), Proc. ICANN'93, Int. Conf. on Artificial Neural Networks, Springer, London, UK, 1993, pp. 427–434.

[48] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), ACM, New York, NY, USA, 1967, pp. 483–485.

[49] J. L. Gustafson, Reevaluating amdahl's law, Communications of the ACM 31 (1988) 532–533.

[50] X.-H. Sun, J. L. Gustafson, Computer benchmarks, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 1993, Ch. Toward a better parallel performance metric, pp. 23–39.

[51] M. D. Hill, M. R. Marty, Amdahls law in the multicore era, IEEE COMPUTER.

[52] M. Harris, Optimizing parallel reduction in cuda, NVIDIA Dev. Technology.

[53] Intel, INTEL THREADING BUILDING BLOCKS 4.0 OPEN SOURCE. http://threadingbuildingblocks.org/. (2012).

[54] A. Bhattacharjee, G. Contreras, M. Martonosi, Parallelization libraries: Characterizing and reducing overheads, ACM Trans. Archit. Code Optim. 8 (1) (2011) 5:1–5:29. doi:10.1145/1952998.1953003.
URL http://doi.acm.org/10.1145/1952998.1953003

[55] J. Garcia-Rodriguez, A. Angelopoulou, J. Garcia-Chamizo, A. Psarrou, S. Orts-Escolano, V. Morell-Gimenez, Fast autonomous growing neural gas, in: Neural Networks (IJCNN), The 2011 International Joint Conference on, 2011, pp. 725 –732.

[56] P. Besl, N. McKay, A method for registration of 3-d shapes, IEEE Trans. on Pattern Analysis and Machine Intelligence 14 (2) (1992) 239–256.