GPU Acceleration and Interactive Visualization for Spatio-Temporal Networks

BY

ANDREA PURGATO
B.S, Politecnico di Milano, Milan, Italy, 2014

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2016

Chicago, Illinois

"Computer Science is no more about computers than astronomy is about telescopes."

Edsger W. Dijkstra, 1970

ACKNOWLEDGMENTS

I would first like to thank my thesis advisors Angus Forbes, Tanya Berger-Wolf, Marco D. Santambriogio for the help given during these last months. Thanks also to EVL, to have provided a great work place where make this thesis possible.

A special thanksgiving goes to the Italian crew here in Chicago, thanks to Andrea, Roberto, Vittorio, Davide, Benedetto, Federico, Arianna, Chiara, Marc and Lorenzo for this amazing year in Chicago. I would also like to thank two special friends that supported me even from the other side of the world, Mario and Anita.

Finally, I must express my very profound gratitude to all my Family members, my father Roberto, my mother Luisa, my sister Elena and my grandmother Rosa, that made this American adventure possible. Thanks to have supported me from Italy every single day.

AP

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF ABBREVIATIONS

GPU	Graphic Processor Unit
CPU	Central Processor Unit
FPGA	Field Programmable Gate Arrays
GPGPU	General-Purpose Computing on Graphic Processor Units
SPT	Spatio-Temporal Network
DN	Dynamic Network
TW	Time Window
PCC	Pearson Correlation Coefficient
SIMT	Single Instruction Multiple Thread
SIMD	Single Instruction Multiple Data
GM	Global Memory
SM	Streaming Multiprocessor
SVG	Scalable Vectorial Graphics
JS	Javascript

SUMMARY

The thesis work presented here focuses on the optimization of the similarity computation among nodes for spatio-temporal networks. The study case in this work is the computation of spatio-temporal networks on brain. The aim of the first part of the work presented is to built a GPU-based computation system able to speed up the networks definition. In the second part of the work we developed an interactive tool to visualize the networks computed using the GPU-based method developed in this first part.

Nowadays, with the increase of computational analysis in sciences such as biology and neuroscience, the computational aspect is the most challenging one. Scientists need tools able to process large amounts of data simultaneously. Previously, scientific computations were performed using clusters of computers, but this type of infrastructure is very expensive and complex to build [1; 2]. The work in this thesis is part of a greater project that has the aim to apply spatio-temporal networks inference techniques to perform network analysis studies in different fields. One of the problems of spatio-temporal network applications is the computational time, and it becomes impractical to keep developing studies when it takes a long time to analyze and compute the results. We aim to help the researchers to get results in a reasonable amount of time, so they can be focused more on their studies. There are many ways to speed up the computation process, one of these is to exploit the paradigm of parallel computation using Graphic Processor Unit (GPU). GPUs represent a low-cost solution to this problem since the level of parallelization they have is really high and GPUs hardware match exactly the requirements for the problem addressed. With the first part of this thesis work we present a GPU solution and we evaluate the performance of spatio-temporal networks definition.

The second part of the thesis work uses the results computed in the first part to propose an interactive visualization tool for the spatio-temporal networks computed. With a decrease in computation time, it is easier to achieve a comparison between the results. The proposed visualization tool allows for the changing of parameters at run-time and visualizing the differences in the resulting networks. This process is made possible thanks to a great work of pre-computation of all the data that must to be visualized using the GPU-based system developed. This second part of the work goals to improve the understanding of the results computed by the algorithm. In this study, we focus on brain network computation and visualization.

We mentioned that this thesis work is composed by two different parts. One of them is used to compute the similarity measure needed to define the DN (explained in Chapter 4). While, the other aims to visualize interactively the DN computed, leaving one degree of freedom to the final user (explained in Chapter 5). One of the goal of this work is the creation of a pipeline that goes from the collected data to the visualization of the result, which in our case are the networks computed previously. In fact the two piece of software developed are part of the pipeline that defines the framework we present. This framework want to avoid the dependability problem given by that the different software are not able to communicate among each others.

CHAPTER 1

INTRODUCTION

Parallel computing is the new answer to time-consuming simulations and computations. It is no longer an exotic thought, but a real possibility. The needs of the computing world in the last years have changed radically, more and more different sciences now require high speed computation of massive amount of data, from the classical physics and mechanics to new sciences like data science and computational finance [3; 4].

This particular trend changes the way to think about computation in the computer world. From the first computers (early 1980s) the big companies tried to increase the computation power of computer Central Processor Unit (CPU). This trend changes when the CPU producers were not able to increase the performances in a way similar to the first years, fact derived from reaching of the limitations imposed by the architecture and by the physical characteristics of the processors. So the concept of high speed computation moved into a new direction: parallelize the computation using new architectures which are completely different from the previous. In this change an important role has been played by Graphic Processor Unit (GPU), that present a different architecture concept for the computation parallelization [5].

Social network analysis is an example of science field where the computational problem is the bottleneck in the studies that are performed to analyze interaction among entities. One of the new models applied to social studies is Spatio-Temporal Network (STN). This model adds the spatial embedding and time aspect to normal graphs to represent the dynamics of the interaction over the time. STNs could be applied to many different contexts which involve interactions. Sociology is the science where this technique born, but other different sciences like biology started to use in their studies this new model. Our work presents a GPU based approach to speed up the STNs inference process. The acceleration provided by the hardware aims to increase the level of the analysis performed on STNs. Moreover, we also present a meaningful visualization of the networks inferred. This work can be applied to any science that want to use STNs model. In fact, the network computation is completely independent by the context of application.

STNs model adds spatial embedding and, especially, the time aspect as new dimension of the problem. While the spatial embedding often simplifies network visualizations (the nodes come with pre-defined coordinates), the time aspect makes the creation of a meaningful visualization challenging, in fact the network visualization should make clear the shape changes of the STNs over the time. Therefore, researchers should be able to see more than one time consecutive instant of the network. Shi et al. [6] discuss about three main aspects that makes the dynamic network visualization challenging. The first is *Visual Metaphor*, the adding of the time introduce a third dimension to the solutions space, and plotting three dimensions data is not always easy. Then there is the *Scalability* problem, due to the third dimension the size of the graph grows significantly increasing clutters of the layout computation. The last challenging aspect is the *Suitability for Analysis*, which means that is difficult to provide an efficient tool with a good visual assistance for human analysis.

Biology is one of the fields that in the last few years have expressed interest in STNs. The application of STN model to the interactions among animals is used to study animal behavior and how they create communities in the population, Rubenstein et al. [7] applied STN model to zebras presenting a detailed social study of the population. STNs fit perfectly the biologists problem but the common denominator of STN definition is the high execution time of algorithms. Furthermore, one of the new fields where hardware computation is becoming popular is neuroscience, science that focuses its research on the nervous system. In particular, this interest is manifested by computational neuroscience, an interdisciplinary research area that studies the brain functionalities in term of information processing. This type of research is done by looking at the interactions that occurs in the brain. One possible way to analyze the interactions that occur in the brain is to use the STNs computational model.

Our work focuses on the networks defined (implicitly) on neurons and their interactions or correlations with each other. The project that involves this work aims to find some patterns in these networks to understand, for example, the aging process in individuals, comparing young brain activities with older brain activities. Studies performed by neuroscientists involved in this project are done applying the concept of STNs to brain. The spatial embedding of the nodes is given by the neurons position in the brain, which does not change over the course of the time. The dynamics of these networks are derived by the rapid change of the interactions among entities during the time.

Brain networks belong to the class of biological networks, there exist different types of brain network based on the different type of data available. In particular, in this context, an edge represents a functional connection between two brain cells (neurons). This doesn't mean that a physical connection between the cells is present, these functional connections may or may not reflect the anatomical connectivities present in the brain.

The main problem of STN analysis applied on brain is the requirement of a massive data quantitative processing, which means long computation time. Brain activity is really dynamic and fast, which is very difficult to capture and analyze. Typically, this kind of analysis is done by aggregating raw data over a fixed (sliding) time window into a single network time step. However, the problem with such an approach is that the dimension of the window is not well-defined and the noise that is present in the data makes the analysis complex, as described by Llano at al. [8]. The brain data used in this work are collected using flavoprotein autofluorescence imaging technique stimulating a brain slice of the brain and recording the activation over the time [8]. However, different kinds of data can be used to define spatio-temporal networks with the program developed in the first part, such as fMRI data and brain calcium data. The proposed idea is to use network analysis to understand networks derived from pixel correlations over time in the resulting brain image time series. The application of the work in this thesis to neuroscience aims to help the domain experts to improve the level of their studies by decreasing the computation time of the STN definition, adding powerful tools that can be used to understand the results computed by exploiting the hardware acceleration.

CHAPTER 2

BACKGROUND

In this chapter we explain all the fundamental concepts required to understand the thesis work developed. The first section is focused on the spatio-temporal network model. Then we talk about the usage of GPUs for general purpose computation and about their programming model.

2.1 Spatio-Temporal Networks

Networks, or graphs, are the most common tool used to represent interactions among entities. In our world a great number of entities interacts, from animals to nervous system. All the interactions among them could be modeled as a graph where the interactions are represented by edges. However, in the real world the connections could change during the time. In static networks, this characteristic cannot be represented because when an edge is defined it cannot be removed. In fact, in static networks the time aspect is not considered. STN introduces the time variable in the problem, one precise interaction between two elements could be limited in the time. This type of network introduces a new idea of network model to use for interactions analysis, so the tools used for analysis have to change with them.

STN is a generic definition that can be applied to different sciences to study the interaction that the entities have. Berger-Wolf et al. [9] apply STNs to animals to understand the social behavior and extract dynamic communities created inside a population. Each individual represents a node that interact with the others moving also spatially. Instead, Przytycka et al. [10] apply the study of STNs to cellulars to study the reaction of cells to different stimulations and how they interact.

2.1.1 Spatio-Temporal Networks Definition

Beck et al. [11] define STN as a sequence of graphs, one per each time instant of the observation interval of the data, each graph is defined as:

$$G := (V, E) \tag{2.1}$$

Where, V represents the set of vertexes, or nodes, of graph G and $E \subseteq V \times V$ is the set of edges defined for the graph G. Each vertex is represented in turn, by a set of spatial coordinates that define the position of it in the spatial domain of the network. These coordinates may be static or dynamic, with static coordinates the vertexes does not change the position, networks with static coordinates are known as: Dynamic Network (DN), which are a sub-model of STN. On the other hand, with dynamic coordinates the vertexes change their position on the spatial domain of the problem, this end in an increasing level of difficulty to the STN definition.

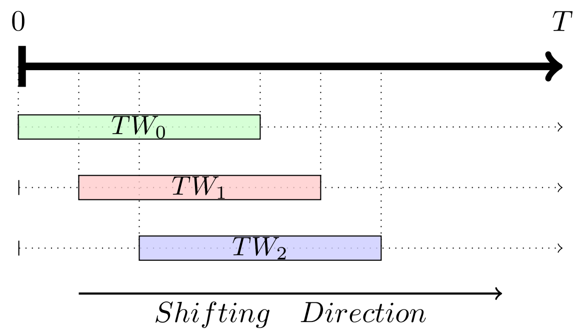
In STNs, as stated before, each time instant has its own graph, let's define T as the number of time instants of the observation period and $\mathcal{V}$, with cardinality P, as the set of all the nodes present in the input dataset. Now we can define a generic STN Γ as:

$$\Gamma := (G_1, G_2, ..., G_T) \tag{2.2}$$

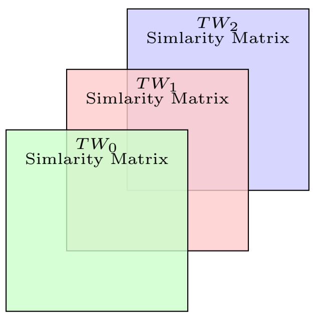
$G_i : i \in [1, T]$ is a graph as defined in Equation 2.1 where the sets of vertexes and edges change for every time instant i, and $V_i \subseteq \mathcal{V}$. To have a global view of the network we first assign a unique identifier to each node in the input dataset, then is possible to adopt a notation for the nodes like $n_{t,i}$, which means that the node $i - th$ exists at the time t of the timeline, the same thing could be done for the edges $e_{t,i,j}$, where it means that the edge from node i to node j exists at the time instant t (in both cases $t \in [1, T]$). This notation avoids confusion related to the time where each node or edge is present in the network. In STNs applied to real study cases the nodes represent abstraction of the subjects of the studies.

The kind of networks we are considering for this thesis work have in common the spatial domain space of the input data used to defined the nodes of the network. Data used for this purpose present the spatial aspect as determinant aspect of them, we can extract a spatial position for each element that is used also to define the network. Each element of the input data, which represent the nodes of the network, present a time sequence of values coming from the input dataset.

An important aspect of STN is the definition of the edges for each time instant graph. This process depends on a similarity measure computed among each node pair using the time sequences associated to the nodes. In particular, an edge is defined for a time instant t between node i and node j if the similarity value between the two nodes at time t is greater than a threshold THR. THR represents the first degree of freedom of the problem addressed, its values is used to define edges in each STN graph. The meaning of the similarity threshold change with the changing of the data domain used to compute the STN. In brain networks



Phase 1: Setting the TW size and shifting of the TW over the observation period.



Phase 2: Computation of the correlation matrix for each shift of the TW.

Figure 1: Shifting of the time window over the observation period. Each movement, see Phase 1, requires the computation of a new correlation matrix, Phase 2, since the values used to compute the similarity are different. Each similarity matrix assumes the shape showed in Figure 2.

the threshold represent how much the activity of two brain cells has to be similar to define an edge between them, in this case, moreover, brain cells does not move spatially, so we can consider this network type as static. Another example is about individuals networks, where the threshold value has to represent how much the individual remain closer to each other to consider interaction among them, in particular individuals networks present dynamic spatial position since the subjects have the ability to move. For this reason the value of the similarity threshold change based on the application domain and based on the kind of data collected.

2.1.2 Time Window: Temporal Resolution and Temporal Aggregation

The last important concept in STNs is the Time Window (TW). It represents the dimension of the look ahead from the current time instant where we want to look for interaction. The TW has a fixed size W ($W < T$) decided before the start of the computation, and it does not change

$$S_t = \begin{bmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,P} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,P} \\ \vdots & \vdots & \ddots & \vdots \\ s_{P,1} & s_{P,2} & \cdots & s_{P,P} \end{bmatrix}$$

Figure 2: Shape of the similarity matrix for a generic time instant $t \in [1, T]$, $s_{i,j}$ represents the similarity value between node i and node j at time t.

during the STNs definition process. The size of the TW represents the second degree of freedom of STN problem. Since the TW does not cover the whole observation period it is shifted by a fixed number of time instances, which is yet another parameter. However, if we come it for a shift of 1 then all the others are subsets (Figure 1 show an example of shifting). Each movement changes the values inside the TW. The values inside the TW are used to compute the similarity among the nodes of the network. The similarity must be recomputed for each shifting, assigning to every initial time of each shift a similarity matrix among each node involved in the network. An example of similarity matrix for one time instant is provided in Figure 2, where P is the number of nodes of the input data.

2.1.3 Spatio-Temporal Network Computation: a New Challenge

One of the goal of this work is the creation of a parametrized visualization for the results computed. To achieve this goal the results have to support the degree of freedom left in the visualization. For this reason the computation cannot filter or elaborate the data at run time, the similarity matrices have to be saved integrally. Furthermore the computational aspect has always been a critic point in STN definition because of the great mole of data to elaborate.

In the problem addressed we have seen that there are two degrees of freedom, the similarity threshold and the TW size, in the next part we explain the computational aspects related to the degrees of freedom.

- **Similarity Threshold**: It represents the spatial parameter of the problem since it is used to defines the edges in the network. Elaborate the spatial aspect of the data is the easier aspect of the problem. This is due by the *inclusion property* that the edges have: given similarity threshold X and the set of edges defined using X, E_X. We have that for every other set of edges E_Y defined with similarity threshold $Y < X$, the statement: $E_X \subset E_Y$ holds. However, we can see from the edge definition process that it is only a filtering process that takes only the node pairs that have threshold greater than the threshold given. So, this method could be performed at run time in the parametrized visualization thanks to availability of all the integral similarity matrices. The inclusion property simplify the edge definition when the threshold is changed.
- **Time Window Size**: It represents the temporal parameter of the problem. The temporal aspect of the similarity computation is the most challenging one. This parameter have to be decided before the computation of the similarity because the program has to know which values use to compute the similarity. We decided to keep this value fixed for the computation because the maths to apply at the results, if we want to change the time window size at run time, is too complex for an interactive front end framework like the visualization tool that we propose in this work. We got to this conclusion for different reasons, the most important is that the similarity could change with the changing of the

data domain, so all the maths operations applied to update the similarity at runtime has to be modified. A possible solution could be the creation of an ad hoc script to compute the similarity that is launched in background by the front end framework that takes as input the time window size, but the similarity computation would require too much time for a real time and interactive visualization.

2.2 Graphical Processor Units

2.2.1 GPU Evolution

GPU defines a computing architecture different from the usual well known processors. These devices were made to be used for graphics computation and elaboration. GPUs born in the late 1980s and they were thought to accelerate the graphics computation and the images rendering on the display, avoiding the CPU to make them. General users started to appreciate GPUs in the first 1990s, adding these new units to their personal computers. The purpose of the GPU until a decade ago was only the graphics elaboration for multimedia applications and gaming. Especially gaming helped the growth of the GPUs development making these devices affordable by the normal users. To improve the quality of their products, videogame companies needed powerful devices able to compute high quality images to render on the screens.

The history of the GPUs bring us to the recent past where, in 2006, NVidia released a new architecture, able to support CUDA[1] programming model, that changed the idea of GPU. "CUDA is a parallel computing platform and programming model invented by NVidia. It

[1]`https://developer.nvidia.com/cuda-zone`

enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)" [12]. In particular CUDA model added the possibility to perform general purpose computations on GPUs exploiting the massive number of parallel floating-point computational units. From that moment the computational world started to look at GPUs as a new instrument to use for general purpose computation. The growing of this market is also supported by the greater increase of the GPUs computational power than the CPU trend [5].

In the last years the computational power of GPU devices has been increased significantly, as reported by Owens et al. [13]. The key of this exponential growth for GPUs computational power is derived by the new paradigm of parallelism between computation units, [5]. The growth of GPU-based computation system is also related to the increase of the number of libraries and tools accessible that allows this new programming paradigm, like Nvidia CUDA. By no longer needing of external complex software to develop GPU-based programs, the development process becomes simpler and more feasible, even if the programming paradigms for these applications is completely new. Moreover, with the increase of the computational power of GPUs, the devices is improving research in many field of science. This thesis work uses the characteristics of these devices to increase the computational performance for spatio-temporal network.

GPUs applied to general purpose computation represent one of the new revolution in the computational field. Graphic processors and normal processors have completely different characteristics, the cooperation between the two achieve a considerable speed up of the computation. CPUs are optimized for low-latency access to cached data and for an out-of-order speculation execution of the program instructions. Meanwhile, GPUs are thought to optimize the parallelism computation among data at the expense of the latency in accessing data. However, this latency is covered by the presence of an high number of threads that have to be executed. In particular, in GPU Programming, one of the most difficult aspects is write code to keep the processors on the board always active. This fact hides the latency of the memory access, and it is achievable thanks to a lower cost of switching the thread execution context, compared with the higher cost of switching on CPU.

2.2.2 CUDA Programming Model

This section explains in details GPU architecture and logic behind this devices. The two main components that we can find in GPUs are the Global Memory (GM) and the Streaming Multiprocessor (SM). GM is the memory accessible by all the computational units where all the data are stored, this is the equivalent of the RAM for CPUs. This memory is characterized by a high bandwidth to achieve a fast exchange of data between the processors and the memory, it can be accessed by the graphic processors of course, and by the central processor of the machine. SM instead, represent the actual computational units of the device, each one has its own control units, register, execution pipeline and cache. These aspects make the SMs independent of each other.

The execution paradigm respected by GPU is Single Instruction Multiple Thread (SIMT). It means that the execution of the same function occurs on many different threads that work on different data. Each stream, in CUDA architecture, organizes the execution of the threads in group, called *warps*. The threads that compose a warps start the execution at the same time, but once started their are independent each other, each thread has its own instruction counter and state register. This allow each thread to execute also different branch of the program, avoiding waiting time between threads that would decrease the performance. SIMT is similar to Single Instruction Multiple Data (SIMD), the difference between these two is that SIMD executes the same instruction on all the data, without the possibility to go across different branches of the program. Meanwhile SIMT allow the programmers to write thread-level program, achieving the independence among different threads.

CUDA programming model is based on *kernel functions*, these are specific structured methods that are executed on GPU. In particular when the processor makes a kernel call, launch the execution of a kernel function on the graphic device, each thread created execute the whole function launched. Each call present a well defined structure, showed in Figure 3. The *grid* showed is composed by a set of *blocks* organized in matrix, these lasts contain the effective threads executed by the graphic processors, always organized in a matrix. The matrix that compose the grid of blocks could be have 1, 2 or 3 dimensions, and the size of each dimension is set by the programmer, the same happens for the matrix of thread in a block. The grid shape and the block shape change with the respect of many aspects of the program, however they should be chosen in a way to maximize the usage the graphic processors to hide the GM latency. The main constraints used to built the structure are the ones imposed by the resources available on the device (resisters, shared memory and global memory).

CUDA library defines some intrinsic variables for each thread. They are not controlled or created by the programmer and they could be accessed during the execution of a kernel function. They contain some useful thread information and the most relevant are:

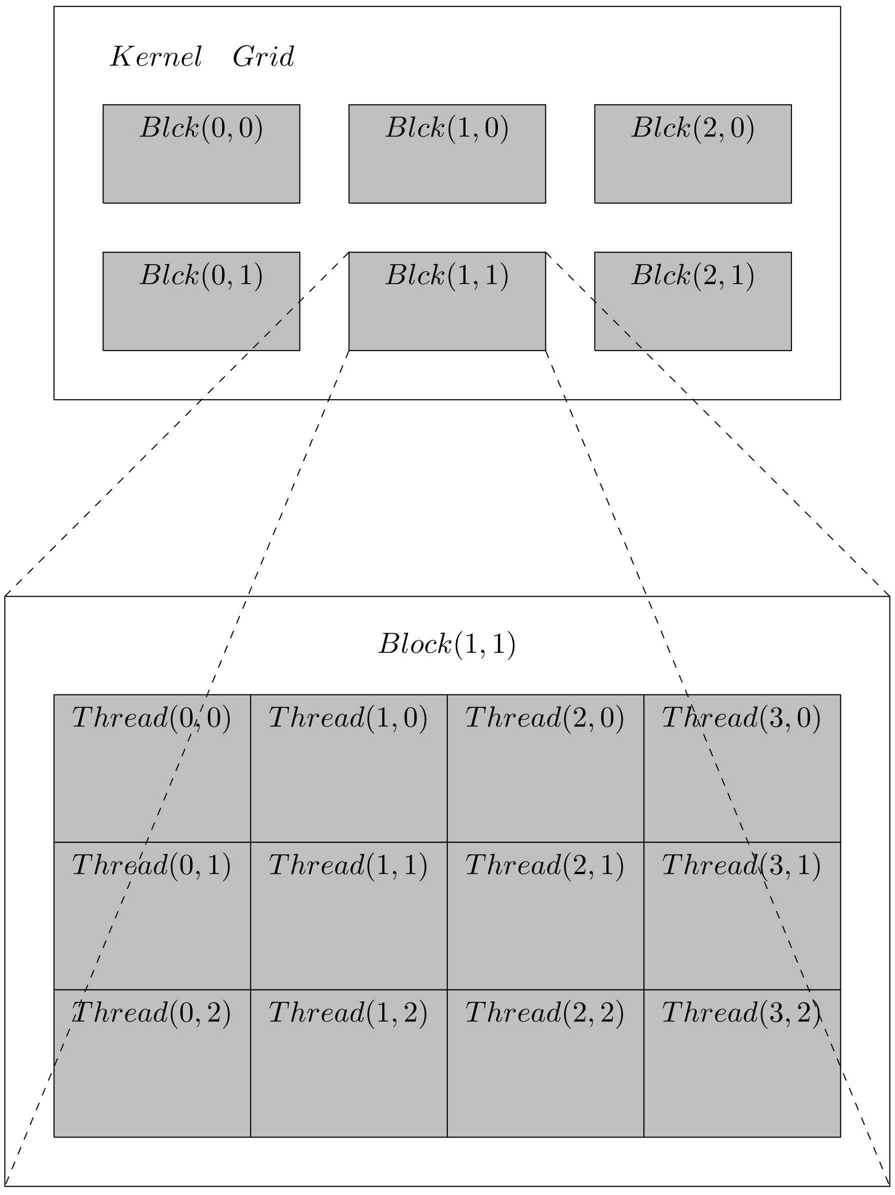


Figure 3: Kernel structure that must be defined for each GPU function.

- *threadIdx*, variable that contains the position (x, y and z) of the thread relative to the block of membership.
- *blockIdx*, variable that contains the position (x, y and z) of the block relative to the kernel grid.
- *blockDim*, variable that contains the size per each dimension of the blocks.
- *gridDim*, variable that contains the size per each dimension of the kernel grid.

Thanks to these intrinsic variables is possible to compute the absolute position and a unique number for each thread, computed as:

$$threadPosition.x = blockIdx.x * blockDim.x + threadIdx.x \tag{2.3}$$

$$threadPosition.y = blockIdx.y * blockDim.y + threadIdx.y \tag{2.4}$$

$$threadPosition.z = blockIdx.z * blockDim.z + threadIdx.z \tag{2.5}$$

$$\begin{aligned} uniqueThreadId =& threadPosition.x + (gridDim.y * threadPosition.y) + \\ & + (gridDim.x * gridDim.y * threadPosition.z) \end{aligned} \tag{2.6}$$

The previous formulas show the computation of the thread position and the thread unique id for a grid with three dimensions. A grid can use less than three dimension, in these cases the dimensions not used will be set automatically by the CUDA library to 1, zero is not admitted since it's not possible to have an empty block or grid.

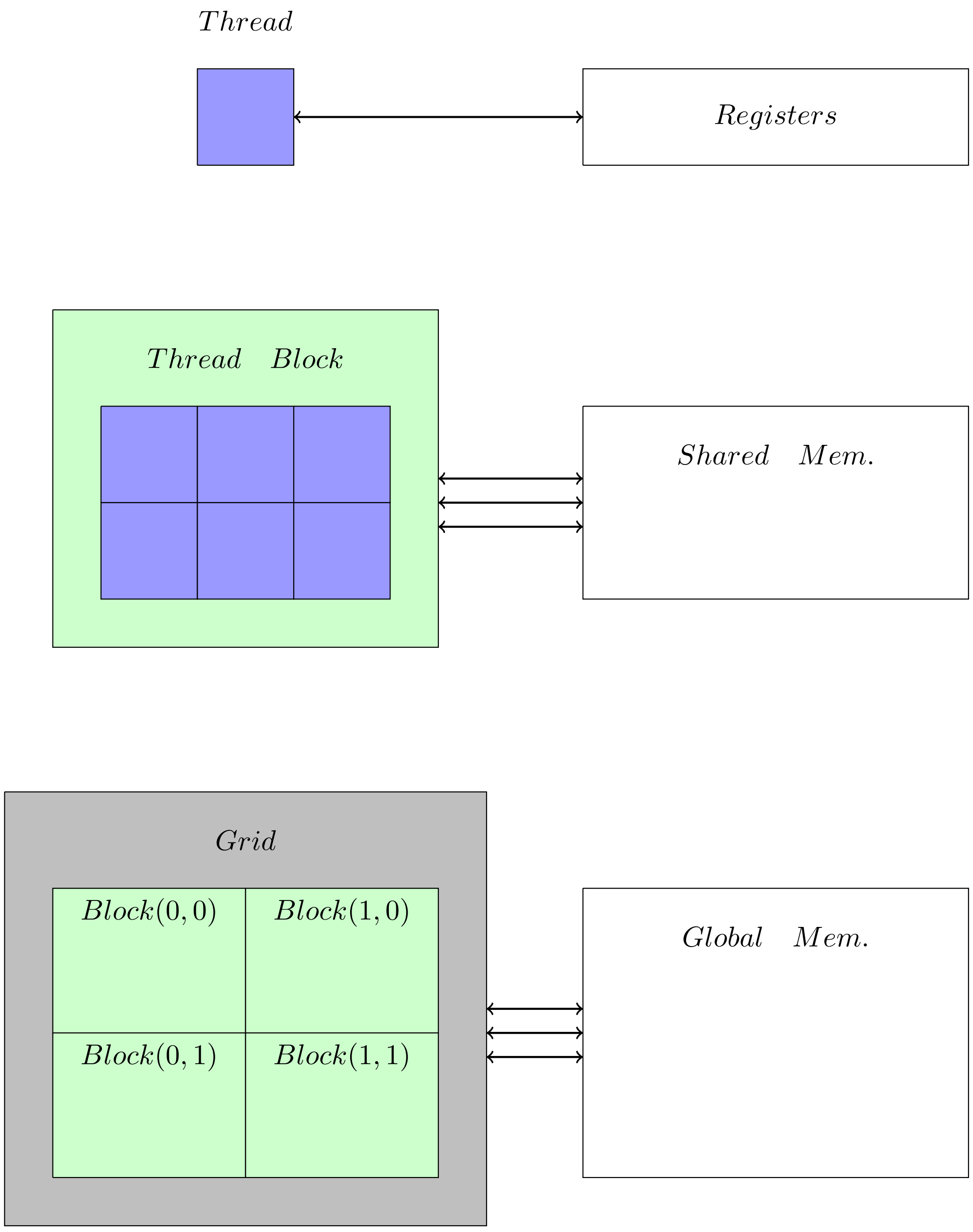


Figure 4: Kernel memory hierarchy defined for each kernel structure.

When the kernel function is called the GPU create a memory hierarchy based on the structure of Figure 4. Each thread has its own local memory, which is represented by the registers, they are used for the execution of the function. Each block, instead, can have some shared memory among all the threads in that specific blocks. At the highest level the grid access to the GM of the device. The shared memory of a block can be accessed only by the threads that are resident in the blocks owner of the shared bytes. There are no way to create a shared memory between threads of different blocks.

CHAPTER 3

RELATED WORK

In the world we live, many different objects and subjects interact among each other. Different or same kind, they want to communicate, from humans to animals until devices or biology cells. It is possible to identify many different types of interactions among these entities. The best way to represent interaction is with graphs, they are simple to understand and to study, they are well known by the scientific community, and they are supported by a lot of math tools that make the studies easier. Many of them allow to extract a dynamic aspect from static graphs to represent changing in the interactions, as discussed by Holme et al. [14].

However, there is an important aspect that the common used graphs are not able to show, the time factor. During recent years the applications of STNs is growing, and many different fields are looking at this new model. Holme et al. [14; 15] discuss in details the potential of this model and present a series of example contexts where STN model is used.

STNs are based on the fact that the interactions among the entities involved in the nets change during the time, so they present a dynamic characteristic that could be represented by the passing of the time. The extraction of temporal networks occurs in different ways according to the application field. Proximity networks for humans could be one of these, technologies like Wi-Fi or Bluetooth sensors guarantees a high resolution data that could be used for the network extraction. Toth et al. [16] present studies where the input data are extracted using these technologies, creating proximity networks used for the analysis. Data collected with this technique give a relative position of each single individual involved in the experiments with the respect to the other. Chen and Valdano [17; 18] applied proximity sensors to animals to built dynamic networks. Different technologies, like GPS tracers, could be used to get absolute position of individuals in the domain space, Berger-Wolf et al. [9] present a specific application of STNs to zebras where they aim to study the social interaction aspect of the animals involved.

In brain networks the most common technology to extract data for the network computation is the functional Magnetic Resonance Imaging (fMRI). It measures the level of the oxygen in the different part of the brain, this level of blood is strictly correlated by the activity level of the specific brain sections. The interactions among brain sections is defined by the similarity measure adopted computed between the blood level in different brain regions. Smith et al. [19] show an interesting work that has the aim to extract meaningful networks from fMRI brain data. One alternative technique used to extract data from the brain is the exploitation of the fluorescence property of Flavoproteins, as preformed by Llano et al. [8]. In this case thanks to this particular property, the stimulation of brain slices increase the luminosity of the active brain cells.

The novel concept of STN is introducing new kinds of possible analysis to perform on them, one of the most interesting is the community analysis. It wants to understand how the nodes create groups among each other, called communities, and how or when the subjects change community. Berger-Wolf et al. [9] present a framework that using STN identifies the communities created by the subjects and the interactions among them.

The fundamental aspect in STN definition is how to compute the interaction, this aspect is strictly related to the context of application of the model. The level of interaction is defined by the computation of a similarity measure able to identify when two subjects are related to each other. A common solution adopted in brain networks is the usage of a correlation coefficient between the signals that represent the nodes during the observation time. Llano et al. [8] present a work that uses the Pearson Correlation Coefficient. Another example, applied on proximity networks, is the euclidean distance among individuals and the time spent close each other. In both cases the interaction occurs when a fixed threshold value is exceeded in a precise time instant. This defines an edge between the subjects that exceeded the threshold.

As already stated, the problem addressed in this work presents two degree of freedom, the dimension of the time window and the threshold value over which consider interaction. This aspect makes the analysis of temporal networks complex since every change in one of the network parameters means a full recomputation of the network. In fact, the computational aspect is nowadays the bottleneck of STNs analysis, since they take a long time to be computed. This aspect is due to the high number of node to analyze and correlate each other, which is the common denominator in every application of temporal network.

Previous work addressed this problem, Cattaneo et al. [20] perform the same computation developed in this work, they compare the execution time of the computation between different devices, precisely FPGA, GPU and CPU. The results presented show a performance increasing of 14$\times$ for GPU and 30$\times$ for the FPGA with the respect to the serial CPU computation. However, the authors focus on the hardware acceleration provided by the devices tested. Our work also shows the challenges addressed by the massive computation performed. We look at the STNs from a different perspective, we generalize the computation of STN independently from the context of application. Wang et al. [21] propose an hybrid framework that use the CPU-GPU computational power to define brain networks, the framework proposed can be used as first step of an execution pipeline that analyze the brain networks computed. Another similar work that show how graphic processor can be used to compute pairwise similarity on matrix is proposed by Kim et al. [22], where different similarity measures are computed. In addition, Chang et al. [23] show the decreasing of execution time for the Pearson Correlation Coefficient computation and Manhattan distance computation using GPU devices. Our work aims to confirm the importance of the acceleration provided by hardware devices exploiting the high parallel computational power of the GPU to compute STNs. However, massive computation on GPU introduces the problem of the resources usage, Lee et al. [24] present a formulation that is used to balance the computation on graphical devices to prevent the running out of resources.

STNs introduces different challenges in the scientific world, one of the most addressed in the last few years is the creation of a meaningful visualization of the networks. Beck et al. [11] group all the temporal networks visualization. They present a great overview of the main works released in the last years categorizing them based on the kind of visualization performed. Based on the hierarchy proposed, our work belongs to the ”Time-to-Time” problem where the networks are precomputed off-line. The main reason why the visualization of STN is challenging is that is that they are a young tool for social iteration studies and only in the last few years something interesting started to come out, confirmed by the fact that during lasts years the number of publications regarding this problem grew significantly. Ma et al. [25] proposes ”SwordPlots” a new visualization technique used to explore brain networks with the focus on social communities visualization. Instead, Van et al. [26] show a new way of temporal network visualization, they considered snapshots of the networks as points of different domain space with two juxtaposed views.

Previous works addressed the similarity computation on GPU and the STN visualization, as seen before those are standalone works that don’t combine the different steps. With our work we want to propose a full framework that doesn’t exist in the scientific community, it aims to concatenate all the functions that before were done separately, creating an ad hoc solution for the STN elaboration and analysis. Starting from the data preparation arriving to the results visualization or results analysis, passing through the STN definitions. In fact our work was born as an interdisciplinary among different branches of computer science, since it aims to exploit the computational power of high-speed devices to compute STNs for big data to create an interactive visualization. The context of application presented is neuroscience, which is increasingly looking at GPU devices to improve the level of the studies. The high speed computation of temporal networks is supported by a visualization tool that allows to play with the parameters of the problem addressed, making the results computed understandable. This application domain want to show the usefulness of this framework in the analysis performed by domain experts.

CHAPTER 4

COMPUTATION ACCELERATION

The first part of this work is centered on the exploitation of the GPU computational power to achieve a reasonable computation time of the similarity matrices used to define DNs. The goal is to compute the similarity measure adopted among each node pairs present in the input dataset.

The framework we present is composed by two different components: a *Wrapper* and a *Core Program*, organized as in Figure 5. The idea behind the structure of Figure 5 is to make the input data independent from the similarity computation, in this way the framework achieve an high flexibility. As described in Chapter 1, DNs born to be applied in different contexts, where the data collected are total different. So the wrapper prepare the input data turning them into a specific format that is taken as input by the core program to compute the similarity.

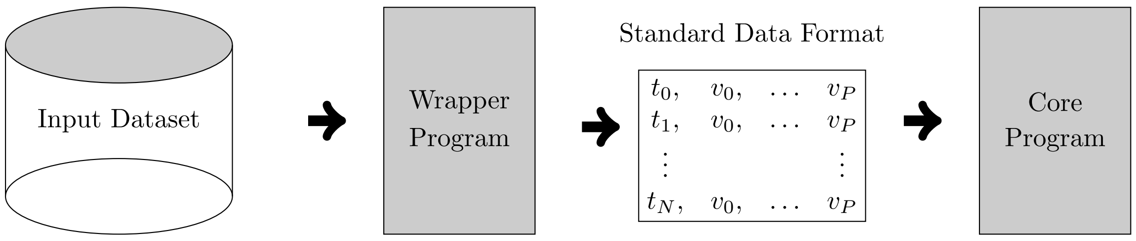

Input Dataset
Wrapper Program
Standard Data Format
$t_0, \; v_0, \; \dots \; v_P$
$t_1, \; v_0, \; \dots \; v_P$
$\vdots \quad \vdots$
$t_N, \; v_0, \; \dots \; v_P$
Core Program

Figure 5: Framework structure, subdivided in two parts: a wrapper program that prepare the data and a core program which compute the similarity measure.

4.1 Edge Definition: Node Similarity Measures

DNs are defined according to a similarity measure that defines how much two nodes are related to each other. In this section we present different meaningful similarity measures that could be used to defines DNs.

Pearson Correlation Coefficient

Pearson Correlation Coefficient (PCC), known also as ρ, measures the linear correlation between two variables X and Y. Its values are limited to the interval $[-1, +1]$, +1 means the highest positive correlation between the two variables, -1 means th lowest negative correlation between the variables and 0 means no correlation between X and Y. PCC is computed as:

$$\rho_{X,Y} = \frac{Cov(X,Y)}{\sigma_X \sigma_Y} \tag{4.1}$$

Where $Cov(X,Y)$ represents the covariance between X and Y, and is computed as:

$$\begin{aligned} Cov(X,Y) &= E[X - \bar{X}]E[Y - \bar{Y}] \\ &= E[XY - XE[Y] - YE[X] + E[X]E[Y]] \\ &= E[XY] - E[X]E[Y] - E[Y]E[X] + E[X]E[Y] \\ &= E[XY] - E[X]E[Y] \end{aligned} \tag{4.2}$$

Instead σ_X is the variance of X, compute as:

$$\begin{aligned} \sigma_X = Cov(X, X) &= E[X - \bar{X}]^2 \\ &= E[X^2] - E[X]^2 \end{aligned} \tag{4.3}$$

Spearman's rank Correlation Coefficient

Spearman rank Correlation Coefficient (SCC), known also as r_s, measures the level of correlation using a monotonic function. A correlation value of $+1$ or -1 means that one of the variable is an exact monotonic function of the other, and no values are repeated in the variables. SCC is defined only between Rank Variable (RV). It is computed as:

$$r_s = \rho_{rg_X, rg_Y} = \frac{Cov(rg_X, rg_Y)}{\sigma_{rg_X} \sigma_{rg_Y}} \tag{4.4}$$

Where $Cov(rg_X, rg_Y)$ is the covariance between the two RV rg_X, rg_Y and σ is the variance of those.

Kendall rank Correlation Coefficient

Kendall rank Correlation Coefficient (KCC), known also as τ, measures the ordinal association between two different quantities. It is based on the tau test, which is used to see the statistical independence using the tau coefficient. It is defined as:

$$\tau_{X,Y} = \frac{(\# \quad of \quad concordtant \quad pairs) - (\# \quad of \quad discordant \quad pairs)}{\frac{n(n-1)}{2}} \tag{4.5}$$

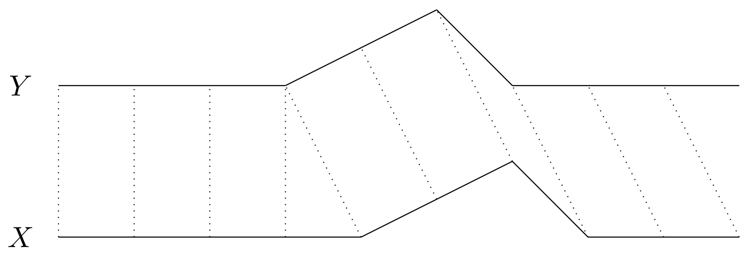


Figure 6: Dynamic time warping example, the dashed lines represent the match applied by the algorithm between the signals.

A pair (x_i, y_i), where x_i and y_i are the i-th values of the variables X and Y respectively, is concordant if given any other pairs (x_j, y_j) with $i \neq j$ one of the two conditions, $x_i < x_j \wedge y_i < y_j$ or $x_i > x_j \wedge y_i > y_j$, hold. When this condition is not true the pair of values is discordant.

Dynamic Time Warping

Dynamic Time Warping (DTW) is an algorithm that want to compute the similarity between two temporal sequences of values, what DTW do is find a match between the two signals over the time following some constraints imposed before the algorithm starts. The similarity derives from how much the two signals match in the time component, Figure 6 shows an example of dynamic time warping applied to two different signals.

Euclidean Distance

Euclidean Distance (ED) measures the distance between two points in a spatial domain, it is computed as:

$$d_{p,q} = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{4.6}$$

The ED is useful to see how far two entities are in the domain space, the network is defined by the permanence of two entities close (under a threshold) for a certain period of time fixed a priori.

4.2 Data Preparation

The transformation of the input dataset in the data format taken as input by the core program is performed by an intermediate passage, executed by a wrapper program.

A wrapper program is a software component that aims to hide the call to an internal routine. In this work the wrapper component has the purpose of data adapter. It reads the input data, elaborates and converts them into a specific format that will be used by the internal routine that will be called by the wrapper. Figure 7 shows the format of the data needed by the core program to compute the correlation, t is the observation time and v is the values that each node assumes over the time, in particular $v_i : i \in [0, P]$ is the value of the node i in the time instant associated to the row. In this work the role of wrapper is played by a PYTHON script that takes the collected data and converts them in the format accepted by the core program.

4.3 Similarity Computation

In this section we explain how we compute the similarity measure that will be used to build the network. In this work we decided do adopt the PCC as similarity measure. The computation of the similarity is made by the core program. It represents the main part of the work and more precisely it computes the correlation among each nodes pair. This part of the work aims to optimize the computation on GPU devices minimizing the execution time. The

$$\begin{array}{lllll}
t_0, & v_0, & v_1, & ...... & v_P \\
t_1, & v_0, & v_1, & ...... & v_P \\
... & & & & \\
... & & & & \\
t_N, & v_0, & v_1, & ...... & v_P
\end{array}$$

Figure 7: Standard data format accepted by the core program as input. P represents the nodes number of the input data and N is the number of observation instants of the input data. Each row represent a time instant (identified by t_i) with all the values for every node in that time instant (identified by v_i).

core program consists of a C++ based software extended with the CUDA library. The decision to use C++ is due to the fact that this language allows the managing of program aspects at a lower level compared with other programming languages that support GPU programming.

4.3.1 Computational Models

Similarity Computational Models

As already stated previously the computation of the similarity occurs for every pair of nodes involved in the problem. Two different ways could be used to compute the similarity:

- **Compute by Pair**: In this case the atomic component of the data is the pair of node. We compute the similarity of each pair independently without mixing the computation of different similarity matrices.
- **Compute by Node**: In this case, instead, the atomic component of the data is a single node. given a node the similarity is computed for all the pairs associated to that node at once. Composing only at the end the similarity matrices.

In the implementation presented in this thesis we decided to adopt the **Compute by Pair** model because it fits better the GPU structure that we have to define for each GPU call we have to perform. We found more easier to transform a similarity matrix into a GPU structure where each cell represents a single pair instead of creating an ad hoc structure with the nodes as atomic component.

Dynamic Programming Model

Another important consideration to make is about the possibility to use the Dynamic Programming (DP) to compute the similarity. The idea is to avoid the complete recomputation of the similarity using the value of the previous time instant. The usage of DP is strictly related to the similarity measure adopted in the problem since the formula used to compute the similarity change with the respect of the problem.

$$Sim_t(X,Y) = Sim_{t-1}(X,Y) + \Delta_{t,t-1}(X,Y) \tag{4.7}$$

In Equation 4.7 we see an example of how the similarity has to be computed adopting the DP paradigm. With this computational model we need only to compute the similarity for the first time instant ($t = 0$) with the similarity formula. Then every other similarity value ($t > 0$) is computed adding the $\Delta_{t,t-1}$ to the previous time instant similarity value. The most difficult aspect of DP is to find the $\Delta_{t,t-1}$ component to be added (Equation 4.8). $\Delta_{t,t-1}$ ideally should be in function of the values that are removed from the time window and the values that are added to the time window.

$$\Delta_{t,t-1}(X,Y) = Sim_t(X,Y) - Sim_{t-1}(X,Y) \tag{4.8}$$

However, the complexity of the $\Delta_{t,t-1}$ computation affect the time and space complexity of the program and sometimes is not convenient as implementation choice.

Regarding the PCC computation that we have to perform we tried to compute the $\Delta_{t,t-1}$ needed to see if DP is convenient to this purpose. Since the PCC is computed using the covariance we decided to compute the $\Delta_{t,t-1}$ factor for the covariance computation and then compute the PCC using Equation 4.1. We use as example the $\Delta_{t,t-1}$ factor computed between time 0 and 1. Applying Equation 4.8 we have that $\Delta_{1,0}$ is equal to:

$$\Delta_{1,0} = Cov(X,Y)_1 - Cov(X,Y)_0 \tag{4.9}$$

Substituting the Covariance definition (Equation 4.2) in the $\Delta_{1,0}$ formula (Equation 4.9) we obtain:

$$\Delta_{1,0} = E[X_1Y_1] - E[X_1]E[Y_1] - E[X_0Y_0] - E[X_0]E[Y_0] \tag{4.10}$$

Where X_i and Y_i are the two vectors of values with the time window that start at the $i - th$ time instant. After substituting the average definition in Equation 4.10 and making some maths simplifications we obtain:

$$\Delta_{1,0} = \frac{1}{(N+1)^2} \cdot \left[N \cdot x_{N+1} \cdot y_{N+1} - N \cdot x_0 \cdot y_0 - \sum_{i=1}^{N} x_i \cdot \Big(y_{N+1} - y_0 \Big) - \sum_{i=1}^{N} y_i \cdot \Big(x_{N+1} - x_0 \Big) \right] \quad (4.11)$$

Where x_i and y_i are the values of the vectors at the time instant i, and N is the size of the time window. We can see from Equation 4.11 that the $\Delta_{1,0}$ factor needed for the PCC computation is pretty complex. Moreover there are some component that are needed for the computation like the summations values. These represent ulterior data to keep during the computation. This increase the spatial complexity of the program significantly, so, we decided to avoid the DP model for the PCC computation that we have to perform.

4.3.2 Data Considerations

A DN is described first by nodes. A node is an entity that can interact with other nodes. For this problem the input dataset is represented by a nodes collection that are extracted from the data collected by the researchers, each one represents a physical entity, which is the subject of the study. Each node must be represented by vectors of values that stand for the signal of the measure taken in each observation instant.

One of the challenging aspects of the computation on GPU is represented by the amount of data to process, and to store. The aim of the program is the computation of the similarity measure adopted in the specific context. It calculates a similarity matrix for each observation instant between each pair of nodes of the input dataset. The computation of PCC (measure adopted to our problem) requires the covariance between each node pairs for each time instant of the observation period. Correlation matrices and covariance matrices have the same dimension.

With some quick calculations, shown here, is computed the amount of memory hypothetically needed to compute the PCC at once. To exploit the properties of a dynamic network, the number of nodes should be significantly high, but this value is strictly related to the field of application. Sciences like neuroscience need to compute a huge amount of data, since in the brain the number of neurons is very high. We take as example for the memory estimation 10000 different nodes observed over 1000 time instants, which is a possible input dataset dimension. The computation of the correlation must be done for each pair of nodes. First we need to load the data in the local memory, this takes $nNodes \times nTime \times sizeof(float)$, where $nNodes$ is the number of potential nodes and $nTimes$ is the number of time instants observed, so $10000 \times 1000 \times 4B \sim 10$ MB is the amount of memory needed for the data. Then the covariance matrices (one per time instant) must be computed and then stored on the memory, they have a total dimension of $nNodes \times nNodes \times nTime \times sizeof(float)$, the third dimension ($nTime$) represents the time component of the problem, which means $10000 \times 10000 \times 1000 \times 4B \sim 372$ GB. Since they have the same shape and dimensions, the same amount of memory is also needed for the correlation matrices. These quick calculations show that the processing of the whole data for the sample problem described requires a self adaptive program that splits the computation based on the resources available on the machine.

4.3.3 Implementation Details

The algorithm implemented is divided into six different phases: device reading, data loading, computation balancing, covariance computation, correlation computation and result saving. Phases covariance computation and correlation computation are executed on GPU. The decision to divide the correlation computation in two phases (covariance and correlation) come from the fact in this way the two functions remain simple and requires less resources when they are running on the devices. The drawback of this decision is that we have to perform two GPU calls instead of one but for each call in this case we are able to compute more similarity matrices. Now we explain all the details of each algorithm phase.

Phase 1. Device reading

The algorithm queries the machine to get the characteristics of the devices available for the computation. The characteristics needed by the program are:

- *name*: name of the device.
- *totalGlobMem*: number of bytes available in the global memory of the device.
- *sharedMemPerBlock*: maximum number of bytes that a block can share among threads.
- *regsPerBlock*: maximum number of registers that a block can use.
- *warpSize*: maximum number of threads that can reside at the same time in a SM.
- *maxThreadsPerBlock*: maximum number of threads that a block can have.
- *maxThreadsDim[3]*: maximum size that the block can have on the three dimensions.

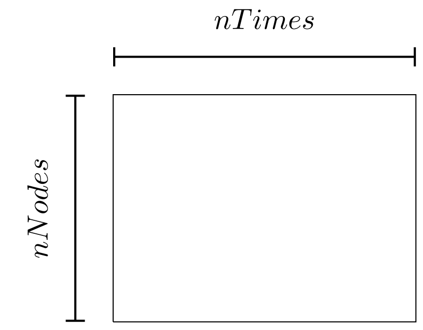


Figure 8: Format of the data loaded on each GPU containing the program input data.

- *maxGridSize[3]*: maximum size that the grid can have on the three dimensions.

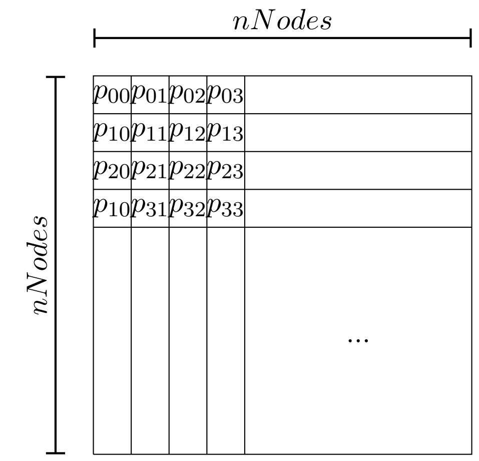
These variables define the architecture where the program is going to run.

Phase 2. Data Loading

The input data are loaded on the main memory of every GPU device present in the local machine. One pointer for every device to the data loaded is kept for the whole program life. The choice to maintain the data on the GPUs for the whole execution is made to improve the performance avoiding frequent memory transfers from machine global memory to GPU memory, even if the data are not used in all the computation phases. The structure of the data loaded is showed in Figure 8, where $nTimes$ represents the number of observation instants and $nNodes$ is the nodes number given as input to the program.

Phase 3. Computation Balancing

In this phase the algorithm defines two main things, the structure used for the kernel functions defined in phases **4** and **5** and the splitting of the computation to avoid the running out of



$nNodes$
$nNodes$
p_{00} p_{01} p_{02} p_{03}
p_{10} p_{11} p_{12} p_{13}
p_{20} p_{21} p_{22} p_{23}
p_{10} p_{31} p_{32} p_{33}
...

Figure 9: Matrix that contains each nodes pairs of the input data. $nNodes$ is the number of nodes in the input data and $p_{i,j}$ represent the node pair composed by nodes i and node j.

resources. This phase of the algorithm is really important because it make the program flexible and independent from the machine.

The model behind the GPU computation is that every thread in the grid structure defined has to compute the results for the atomic component of the data. For the computational model adopted (compute by pair) the atomic component of the result is a pair of nodes. So ideally each thread has to be associated to one pair of nodes and it has to compute the results for that pair. So, we have to built a grid structure that contains all the node pairs of the problem. Figure 9 shows an example of matrix containing all the node pairs. Recalling Figure 3 of Chapter 2, we see that for each GPU structure we have to define the dimensions of the grid and the dimensions of the blocks. So we have to turn the matrix of Figure 9 into the structure of Figure 10 (example of 2D GPU structure).

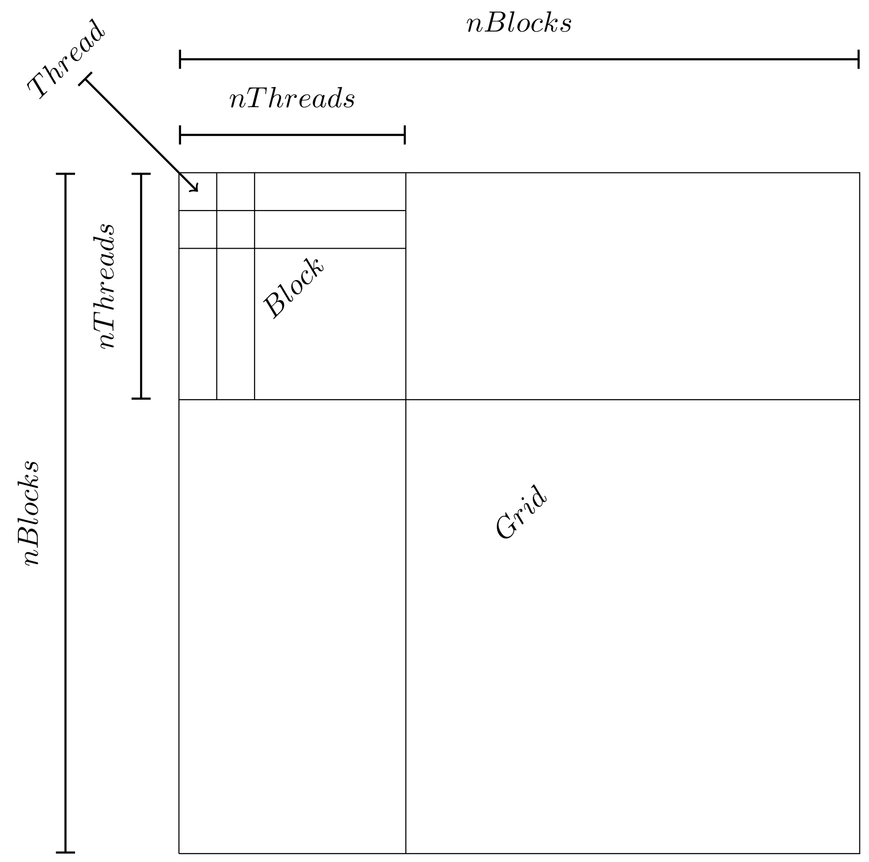


Figure 10: Kernel structure used for the kernel functions in the problem for one observation instant. *nBlocks* represent the number of blocks in each dimension in the grid and *nThreads* represent the number of thread in a single block per each block dimension. The structure used present a quadratic shape, the number of thread per each grid dimension is equal to the nodes number of the input data, in this way we can represent each node pairs.

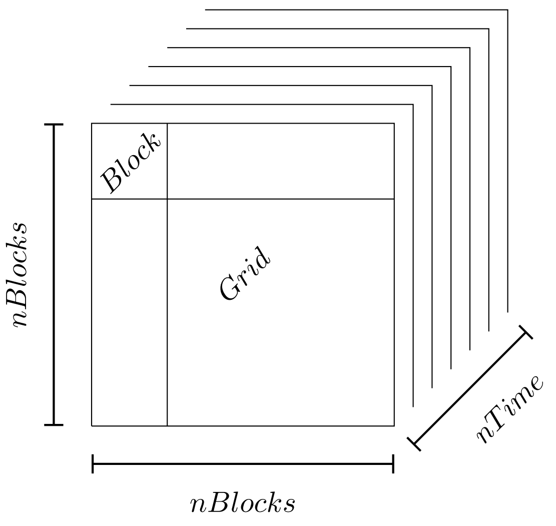


Figure 11: Kernel structure used for the kernel functions in the problem extend with the time component. The time is represented by the $z - axis$ and $nTime$ is the number of observation instants that compose the observation period in the input data.

The number of threads present in each dimension in the structure of Figure 10 is given by the multiplication $nBlocks * nThreads$ and it must be equal to the number of nodes of the input dataset ($nNodes$). An important assumption made to build the structure described is that the number of nodes must be less or equal than the maximum number of thread in a block dimension multiplied by the maximum number of blocks in a grid dimension. The maximum number of thread that we can define for each dimension is strictly dependent to the GPU architecture. However, generally thin number is really high compared with the number of nodes in usual DNs, so the assumption that the number of nodes will never be bigger is legit.

At this point we only exposed how GPU structure looks like. However we didn't consider the time component of the problem, since we have to compute a result matrix for each sliding of the time window. The solution we adopted is the introduction of the the third dimension in the structure showed in Figure 10 to obtain a final design like the one in Figure 11, where the $z - axis$ value represents the observation instant.

The hardware of the local machine impose some limitations. In particular, we focus on those imposed by GPU devices, which are listed here.

- **Global memory**, it is not possible allocate more than the available memory on the GPUs. The GM contains all the data structures needed for the computation. When the number of input nodes is considerably high, the memory is not enough to contains all the data at the same time.
- **Registers number**, a thread to be executed needs a certain amount of registers that is defined by the kernel function. The number of registers usable by a single block of the grid structure is limited by the hardware.
- **Shared memory**, the amount of bytes sharable within a block of threads is limited.

From the constraints highlighted above we focused on the first two since the kernel functions defined for this problem do not require the usage of shared memory.

The **Global Memory** limitation make impossible the computation of the results matrices for all the observation instants in one GPU call, since the space to be instantiated to contains the results is bigger that the memory space available on graphical devices. So we decided to split the computation of the results over the time component in more than one *step* (referring to Figure 11, we split over the $z-axis$). The number of steps is computed by the function defined

Algorithm 1 It returns the number of steps that fit the resources available on the devices on the local machine. Variable *globalMemory* contains the minimum amount of memory available on a single GPU.

```
Function GetSplit():int
    nSplit ← 0
    memoryUsed ← EstimateMemoryUsage(split)
    while globalMemory < memoryUsed do
        nSplit ← nSplit + 1
        memoryUsed ← EstimateMemoryUsage(split)
    end
    return nSplit
end
```

Algorithm 2 It returns an estimation of the memory bytes needed by the kernel functions on the GPU memeory. *nTime* is the number of observations. In the instant of highest memory usage there must be allocated: bytes for the data (function line 2), the covariance matrix (function line 3) and the correlation matrix (function line 4).

```
Function EstimateMemoryUsage(nSplit):int
    obsToProcess = ⌈ nTime / nSplit ⌉
    memory ← nNodes * obsToProcess * sizeof(int) / 1204
    memory ← memory + nNodes * nNodes * obsToProcess * sizeof(float) / 1204
    memory ← memory + nNodes * nNodes * obsToProcess * sizeof(float) / (1204 * nGPUs)
    return memory
end
```

in Algorithm 1, it returns the number of steps needed to perform the computation without running out of memory. The computation of the steps number is supported by Algorithm 2 that returns the amount of memory needed on a single GPU to compute the results given a number of steps. However, if the local machine is a multi-GPU machine we decided to spread the computation of the results matrices over the different devices, dividing the matrices shown in Figure 11 on the vertical axes. Each part of the matrix given by the division is computed on a different GPU. The decision to split the matrix vertically allows to have more time instants in each step since the memory required for each step is lower. In case of multi-GPU machine another possible implementation choice could be the execution of different steps on different GPUs. We decided to avoid this solution because the differences of the time instants computed by the different GPUs could causate problem in the results writing in RAM since multiple thread could write in the same data space causing conflicts. In the solution adopted (vertical split) this problem cannot appear because the time instant computed are the same for every GPU and when the computation is finished the results are merged before the copy in the machine global memory. Figure 12 shows an example of computation balancing according to the solution adopted.

Further the limitation discussed before there is the **Registers Number** constraint. This does not affect the balancing of the computation described before, but it limits the number of threads in a block of the grid structure. This constraints is used to compute the blocks size and the gird size of the GPU structures. We decided to adopt the formulation proposed by Lee et

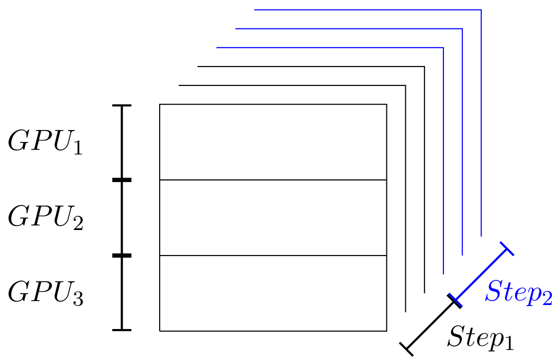


Figure 12: Example of computation split defined in phase 3 on a sample matrix, the machine in this case has three GPU devices.

al. [24], they show how to compute the kernel structure dimension, starting from the registers requirements of the kernel functions.

The following equation represents the main constraint that must be respected:

$$R_{block} * Blocks_{SM} \leq MaxReg_{SM} \tag{4.12}$$

R_{block} is the number of registers required by a single block in the grid. $Blocks_{SM}$ is the number of blocks that can be executed by one SM at the same time. $MaxReg_{SM}$ is the maximum number of registers allocable in one SM.

The number of registers needed by a block (R_{block}) depends on the number of threads executed at the same time and is computed as follow:

$$R_{block} = ceil(ceil(W_{block}, W_{allocation}) * T_{warp} * R_{kernel}, R_{allocation}) \tag{4.13}$$

We have that:

- W_{block}: number of warps in a single block.
- $W_{allocation}$: value that represents the allocation factor of the warps. It is possible allocate a number of warps multiple of $W_{allocation}$.
- T_{warp}: number of thread per warp.
- R_{kernel}: number of registers used by a kernel function.
- $R_{allocation}$: value that represent the allocation factor of the registers. It is possible allocate a number of registers multiple of $R_{allocation}$.

Among all the parameters of Equation 4.12 the only two that depend on the implementation are W_{block} and R_{kernel}. The first one is given by the number of threads in a block (decided by the programmer) divided by the dimension of the warps, while the second is strictly related to the kernel function implementation. All the other parameters are fixed and defined by the device architecture. In our case for NVidia devices they depends on the capability of the NVidia GPU.

The number of blocks executable in one SM at the same time ($Blocks_{SM}$, from Equation 4.12) is computed as follow:

$$Block_{SM} = min(Blocks_w, Blocks_r) \tag{4.14}$$

$Block_{SM}$ is the minimum between two values:

- $Block_w = \frac{MaxWraps_{SM}}{W_{block}}$, number of wraps needed by a single block.

- $Blocks_r = \frac{MaxRegister_{SM}}{R_{block}}$, number of register needed by a single block.

$MaxWraps_{SM}$, $MaxRegister_{SM}$ contain respectively the maximum amount of warps that could be resident in a SM and the maximum amount of register allocable by a block. All of these are defined by the capabilities of the NVidia GPU.

From the previous formulas we see that the only free parameter we have is W_{block}. In fact, it is used to compute R_{block} (Equation 4.13).

$$W_{block} = \max_{n}\{n|R_{block} * Blocks_{SM} \leq MaxReg_{SM}\} \tag{4.15}$$

After the evaluation of the steps number and the warps number of a single block we compute the size of the grid and the block of the structure needed by the kernel. The dimensions of the kernel components are computed following the next operations:

1. Compute the number of observations instant to process in one step and its offset from the first observation instant.
   $obsToProcess = \left\lceil \dfrac{nTime}{nSplit} \right\rceil$
2. Compute the division over the GPUs available of the nodes on the y axis, and the offset for each division from the first node. The number of nodes per each device is given by the division of the number of nodes by the number of devices. However if the machine has only one device this computation is irrelevant since it returns the total number of nodes.
   $nodePerDevice = \left\lceil \dfrac{nNodes}{nGPUs} \right\rceil$

3. Compute the dimensions of the block in functions of the number of warps per block computed before. From Equation 4.15 we get the number of warps in a single block (W_{block}), so the number of thread in a block will be: $W_{block} * T_{warp}$. From this number we compute the dimension of the block.

   - $blockDim.x = \left\lceil \sqrt{W_{block} * T_{warp}} \right\rceil$
   - $blockDim.y = \left\lceil \sqrt{W_{block} * T_{warp}} \right\rceil$
   - $blockDim.z = 1$

4. Compute the dimensions of the grid in function of the number of thread per block, it must have three dimensions:

   - $gridDim.x = \left\lceil \frac{nNodes}{blockDim.x} \right\rceil$
   - $gridDim.y = \left\lceil \frac{nodePerDevice}{blockDim.y} \right\rceil$
   - $gridDim.z = obsToProcess$

Phase 4. Covariance Computation

The covariance, needed by the program to compute the correlation (Equation 4.1), it is computed on GPU using a function specifically written to parallelize the computation. Algorithm 3 shows the kernel function defined to compute the covariance of one node pair, the two nodes identifiers are contained in variables p and q. The function is executed on each thread defined in the grid structure associated to the function, the sizes of the structure are computed in the previous phase of the algorithm.

Algorithm 3 It is executed on GPU and it computes the covariance among two nodes, *p* and *q*. The input variable *timeWindow* represent the dimension of the time window, *nodeOffset* is the offset on the $y - axis$ due to the division of the computation among different devices while *timeOffset* is the offset on the $z - axis$ due to the split of the computation in more steps.

```
Function ComputeCovariance(timeWindow, nodeOffset, timeOffset):matrix
    p ← blockIdx.x * blockDim.x + threadIdx.x
    q ← blockIdx.y * blockDim.y + threadIdx.y
    t ← blockIdx.z
    for $i \in [0, timeWindow)$ do
        product ← product + data[q + nodeOffset][timeOffset + t + i] * data[p][timeOffset + t + i]
        pSum ← pSum + data[q + nodeOffset][timeOffset + t + i]
        qSum ← qSum + data[p][timeOffset + t + i]
    end
    covariance[q + nodeOffset][p] = (product / timeWindow) - (pSum / timeWindow) * (qSum / timeWindow)
    return covariance
end
```

The main difference between an equivalent function that compute the covariance on CPU is the computation of p and q. They are the two nodes unique identifiers that compose the pair associated to the thread. As shown in Section 2.2, for each thread we can compute the absolute position of the thread in every dimension of the structure. We use the absolute position of the thread to get the node identifiers. In particular the x position represents the absolute number of node p. While the y position represents the relative number of node q, this is relative because the computation has been divided over the $y - axes$ (recall to Figure 12). So to get the absolute number of q is necessary sum to the relative position the node offset (computed in Phase **3**). The z position represents the relative time instant of a the observation processed, to get the absolute observation time instant is necessary to add the observation offset, which is the difference between the staring time instant of the current step and the initial time of the observation period. The access to the data loaded structure on the GM of the GPU must be done with the absolute number of the nodes and the absolute number of the time instant.

In some rare cases it could happen that the input signal for a node is constant within the TW selected, this particular case results in the variance equal to zero, known as zero-problem. The variance cannot be equal to zero since it will be at the denominator of the correlation (Equation 4.1). To avoid the zero-problem we introduced a little noise when the covariance results zero.

Phase 5. Correlation Computation

PCC is computed using Equation 4.1, as happens for the covariance, it must be computed for each pair of nodes. Algorithm 4 shows the code that defines the kernel function used to compute the PCC of one node pair. The function is executed on each thread of the kernel structure defined in the Phase **3** of the algorithm.

Even in this case the nodes identifiers are represented by the absolute position of the thread in the GPU structure, p and q as in the previous phase are the variables with the nodes identifiers. The tricky part of the correlation kernel function is the indexes computation, they are needed to access the covariance matrix of the nodes, computed in Phase **4**. The covariance matrices are already stored in the global memory of each device and is accessed using the absolute indexes calculated in the function. PCC, instead, is computed in the last line of code before the return statement.

Algorithm 4 It is executed on GPU and it computes the correlation among two nodes, p and q. The input variable *timeWindow* represent the dimension of the time window, *nodeOffset* is the offset on the $y - axis$ due to the split of the computation among different devices while *nodeNumber* is the nodes number of the input dataset.

```
Function ComputeCorrelation(timeWindow, nodeOffset, nodeNumber):matrix
    p ← blockIdx.x * blockDim.x + threadIdx.x
    q ← blockIdx.y * blockDim.y + threadIdx.y
    t ← blockIdx.z

    timeInstant ← t * nodeNumber * nodeNumber
    pCovIndex ← (p * nodeNumber + p) + timeInstant
    qCovIndex ← ((q + nodeOffset) * nodeNumber + (q + nodeOffset)) + timeInstant
    covIndex ← ((q + nodeOffset) * nodeNumber + p) + timeInstant

    corrIndex ← (q * nodeNumber + p) + timeInstant
    correlation[corrIndex] = covariance[qCovIndex] / sqrt(covariance[pCovIndex]) *
    sqrt(covariance[qCovIndex])
    return correlation
end
```

Phase 6. Results Saving

The algorithm stores the correlation matrices computed in Phase **5**. The saving process creates one text file for each observation instant. It is performed by an independent thread so that the program can continue its execution. To reduce the size of the output file and the saving time it is stored only half of the correlation matrix (it is symmetric over the diagonal). As anticipated, the saving process is executed by a fixed number of independent threads that run in parallel to the GPU program, when all the threads available are busy and the main program has to write new results, it stops the execution waiting the end of one of them.

4.4 Experimental Evaluation

In this section we present the performance evaluation of the GPU program developed. We show the advantages of using GPU computation for general purpose problems and we highlight the low computation time achieved even when the number of the nodes increase significantly.

4.4.1 Evaluation Dataset

The input data for the tests performed came from one of the datasets made available by neuroscientists to compute DNs on brain. In particular, the one used is composed by a series of images where each one represents a different observation instant of the brain. Each image is a frame of a video that records the stimulation process occurred during the experiments made on a brain slice (Figure 13 shows an example of input image). Each image represent a brain slice from a lab rat, the activity of the cells is extracted stimulating the slice and looking at the luminescence of those. In fact, thanks to a protein contained in the brain, the stimulation make the active cells fluorescent, increasing the level of luminosity. The video that records the experiments is able to catch this luminosity change.

We assume that each pixel of the images could potentially be a brain cell. The signal representing the activity of each cell is taken using the level of the gray color of each pixel. In this specific context a node of the network is represented by a brain cell. In Figure 14 is showed an example of sequence of images from the input dataset, for each pixel of the images we extract the hue values and we store all those in a text file in the format showed in Figure 7. Therefore, each node has a vector of values that represent the luminosity of it for each frame of the observation period that is used to compute the correlation between each pair of brain cells.

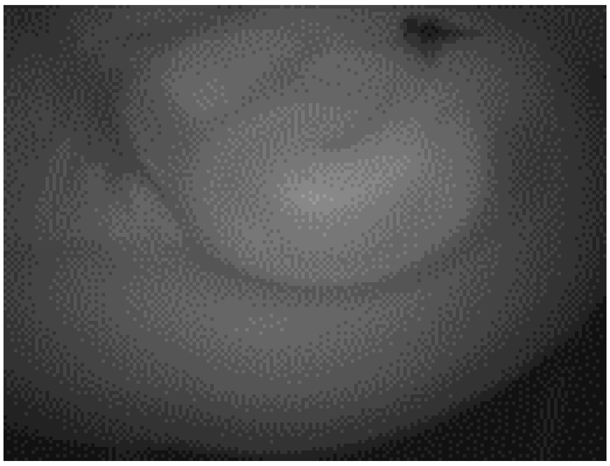
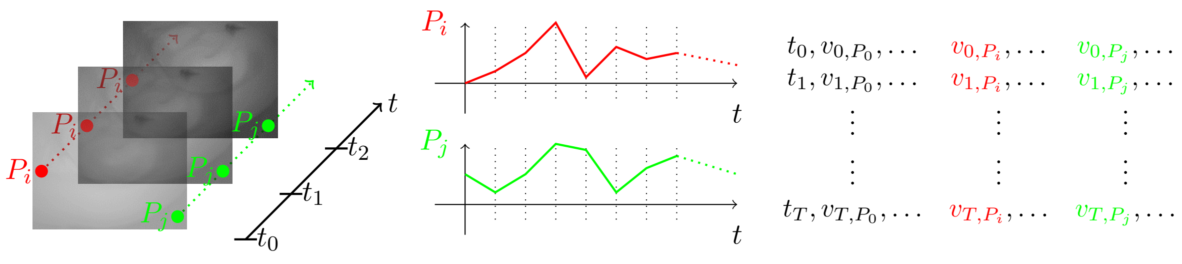


Figure 13: Sample input image, it is a frame extracted from the video that record the experiments performed.


P_i
P_j
t
t_2
t_1
t_0
P_i
P_j
t
t
$t_0, v_{0,P_0}, \dots$ $v_{0,P_i}, \dots$ $v_{0,P_j}, \dots$
$t_1, v_{1,P_0}, \dots$ $v_{1,P_i}, \dots$ $v_{1,P_j}, \dots$
$t_T, v_{T,P_0}, \dots$ $v_{T,P_i}, \dots$ $v_{T,P_j}, \dots$

Input images series.

Pixel discrete function that represent the hue value over the time.

Discrete function converted in the input data for the similarity computation.

Figure 14: Example of input dataset represented by a series of images extracted from a video containing the experiment preformed on the brain. Each image is has a dimension of 130×172 for a total of 22360 pixels. Each pixel is characterized by a discrete function that represent the hue value of the pixel in each image. All the pixels values are stored in a text file in the format showed, where on each row are stored the vales of each pixel (v_{k,P_i}) in the observation instant of the row.

TABLE I: TEST PERFORMED ON THE PROGRAM DEVELOPED.

# nodes	**window size**	**# coefficients**
100	25 50 75 100 125 150 175 200	8870000
200	25 50 75 100 125 150 175 200	35480000
400	25 50 75 100 125 150 175 200	141920000
800	25 50 75 100 125 150 175 200	567680000
1600	25 50 75 100 125 150 175 200	2270720000
3200	25 50 75 100 125 150 175 200	9082880000
6400	25 50 75 100 125 150 175 200	36331520000
12800	100	18350080000
22360	200	49996960000

We decided to perform different tests with different number of nodes from the same input dataset, using subsets of pixels on the images. In tests where the number of input nodes is lower than the total number of pixels we extracted random pixels in the images. Table I shows the number of nodes used for the tests executed and the time windows sizes used for the different runs. The last column of the table shows the number of correlation coefficients to compute for **one** observation instant, this value makes a clear idea about the dimension of the computation performed. The growth of the number of nodes is exponential except for the last which is the maximum number of nodes extractable from the input images have, exact number of pixel. Each image has a dimension of 130 $\times$ 172 pixels, for a total of 22360 pixels per image. The number of images in the dataset used is 1000, one per each frame of the experiment video.

4.4.2 Evaluation Environment

All the evaluation tests are performed on the same machine, which have the following characteristics:

- **CPU**: Intel i7 960 @ 3.20 GHz
- **RAM**: 3 × 4 GB @ 667 MHz
- **GPU**: 3 × 4GB NVidia GeForce GTX 770 (EVGA), Memory Bandwidth of 224 GB/s
- **Storgae**: 2 × 2TB HDD Segate ST3200054AS
- **OS**: Windows 10

The development environment used is Visual Studio 2013[1] with integrated NVidia Nsight[2], which is an extension released by the NVidia company to support CUDA programming in Visual Studio environment. This technology allows, besides the compilation of CUDA code, the debugging of CUDA programs and the monitoring of the program execution on GPU. The version of NVidia Nsight used contains the last CUDA compiler released by NVidia, which is version 7.5, that support C++ version 11.

[1]`https://www.visualstudio.com/en-us/visual-studio-homepage-vs.aspx`

[2]`http://www.nvidia.com/object/nsight.html`

TABLE II: PROGRAM EXECUTION TIME.

# Nodes	**Exe. Time** [s]	**GPU Time** [s]	**Wait Time** [s]	**GPU %**	**Wait %**
100	28.125	0.084	27.125	0.299	96.444
200	92.0	0.309	90.5	0.336	98.37
400	295.625	1.177	290.875	0.398	98.393
800	1149.125	4.709	1133.5	0.41	98.64
1600	4674.5	19.16	4618.625	0.41	98.805
3200	18493.0	76.296	18279.0	0.413	98.843
6400	72485.75	304.284	71616.75	0.42	98.801
12800	307466.0	1106.469	303913.0	0.36	98.844
22360	833987.0	5928.43	821084.0	0.711	98.453

4.4.3 Results Analysis

In this section we analyze the outcome of the tests performed. We show the power of GPU devices used for general purpose computations and the bottlenecks we encountered during the execution.

Program Execution Time

The first results we present is the program execution time, Table II shows the data collected during the tests, in particular:

- **# Nodes**: number of input nodes used for the tests.
- **Exe. Time**: average program execution time of the different runs with different TW. This data include the time needed to load the data and the time to compute and save the results on the local storage. This last term is the most expensive because for each time instant the program has to save a great amount of data.

- **GPU time**: average execution time spent by the GPU to accomplish all the calls. It includes the time needed to compute the covariance matrices and the correlation matrices. As described in Section 4.3, the two matrices are computed in two different GPU calls and the total execution time is the sum of the time needed by each call to be accomplished.
- **Wait Time**: average waiting time spent by the processor in idle mode waiting for a thread to save the results.
- **GPU %**: percentage of the time spent by the GPU computation with the respect of the total execution time.
- **Wait %**: percentage of the time spent waiting the saving of the results with the respect of the total execution time.

The data in Table II show clearly that the computation of the results on GPU requires a very low time compared with the execution time of the algorithm. This great results was possible thanks to the high GPUs computational power. However, the saving of the results needs a very long time, this long time is influenced by the fact that we didn't filter the results and we save the whole correlation matrices computed. Obviously, the time needed to save the results is strictly dependent on the storage devices used. The last two columns of the table highlight better what just stated showing how the execution time of the algorithm is spent by the different operations performed.

Figure 15 remarks how the main bottleneck of the problem addressed is the saving time, in the chart plots is clearly showed that most of the time of the execution is spent waiting the writing of the results on local storage.

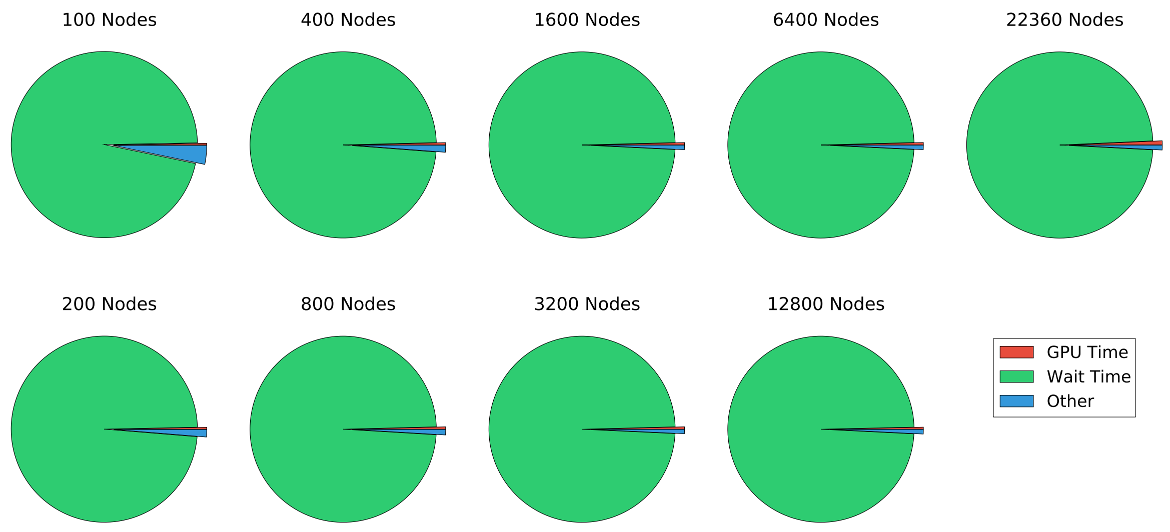


Figure 15: Execution time repartition of the GPU program is spent.

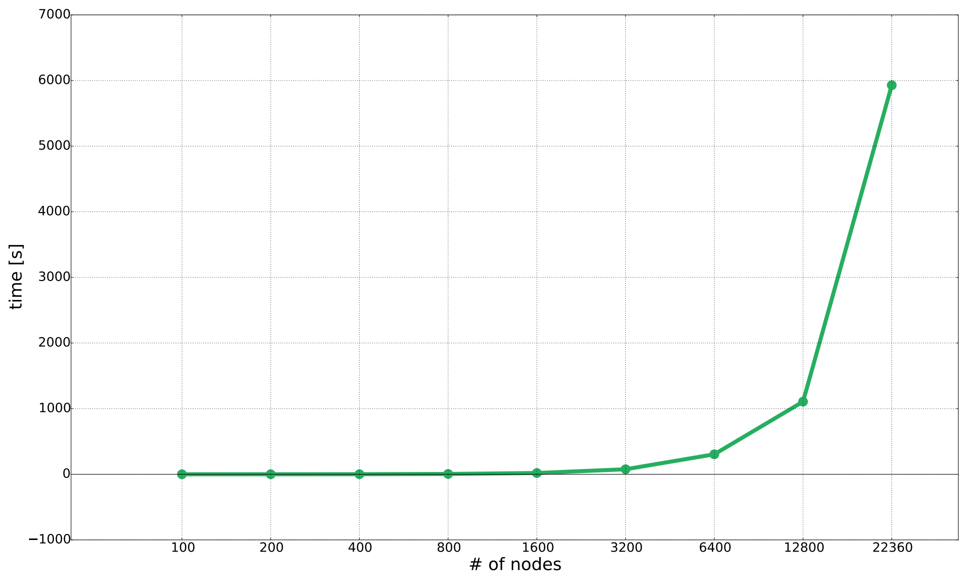


Figure 16: GPU time variation with the increasing of the input nodes number.

Figure 16 shows a linear growing of the GPU time with the increasing of the input nodes number (linear because on the $x - axis$ the input node number grow exponentially).

GPU Execution Performance

The second performance measure that we evaluate is the computation time needed by each GPU call (covariance computation and PCC computation). Table III shows the average execution time needed by each call to compute the results on graphic devices. We can clearly see that the computation of the covariance matrices requires $\sim 80\times$ the time required to compute the correlation. This is due by the complexity of the two GPU functions. The function that computes the covariance presents a loop that goes over the TW to get the data with a complexity of $O(W)$, where W is the time window size, instead the correlation function has a constant execution cost.

The execution time stated in Table III, in both the cases, shows the average time of one GPU call. One GPU call compute all the data for one step. Each call as showed in Figure 12 could have more than one observation instant to process. This justify the strange behavior in the computation time, that grows and the decrease. In fact, this behavior is due by the number of observation instants to compute for each call. Table III shows also the number of GPU calls that are performed by the program (# **Step** column). Figure 12 highlights that when the number of input nodes is big the computation is divided in multiple steps, each step compute a fixed amount of observation instants (# **Obs** column).

To have a better understanding of the load balancing provided by the program, Figure 17 shows how the number of steps and the number of observation instants computed in one call

TABLE III: EXECUTION TIME FOR GPU FUNCTIONS.

# nodes	Cov. GPU Time [ms]	Cor. GPU Time [ms]	# Steps	# Obs.
100	83.353	0.81	1	889.0
200	306.773	2.095	1	889.0
400	1170.549	6.719	1	889.0
800	669.41	3.36	7	127.0
1600	488.944	2.343	39	23.0
3200	377.731	1.852	201	4.0
6400	378.029	1.852	801	1.0
12800	1221.42	6.625	901	1.0
22360	7380.215	21.07	801	1.0

changes in function of the input nodes number. As we expected the two functions have inverse behavior since more step I have less observation instants I have to compute in one call and vice versa. An important consideration is that the exponential behavior highlighted by the plot is due by the exponential growing on the number of nodes, so we can say the the number of steps is linearly dependent with the respect to the number of nodes.

Another important measure used to evaluate the performance is the bandwidth. This value represent the ratio of the data movement between the kernel function execution and the GPU global memory. The bandwidth achieved could be affected by different factors: how the memory is used, how many memory bytes are accessed by the kernel function, how the data are manipulated, etc. The company that produce the graphic devices always provide the theoretical bandwidth that the device can support. This value represent an ideal case that rarely could be

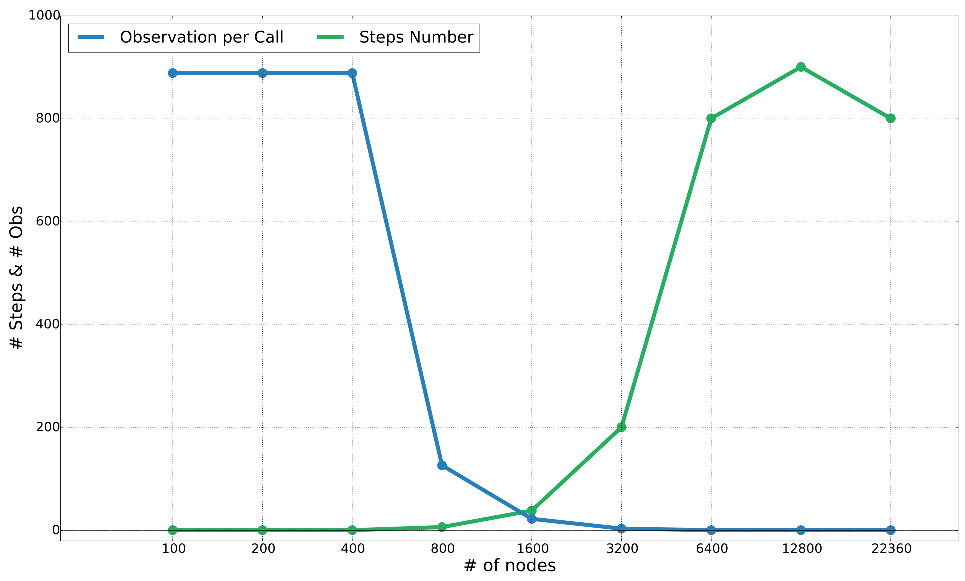


Figure 17: Changing of the number of steps and the number of observation per call in function of the number of nodes.

achieved in real problem. Effective bandwidth[1], which represents the bandwidth achieved by a real problem, is computed as:

$$BW_{Effective} = \frac{B_r + B_w}{10^6 * time} \tag{4.16}$$

Where B_r is the number of bytes read from the GPU global memory by the kernel function and B_w is the number of bytes write on the GPU global memory by the kernel function. Instead, $time$ represents the execution time of the kernel function, expressed in milliseconds.

[1]https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/

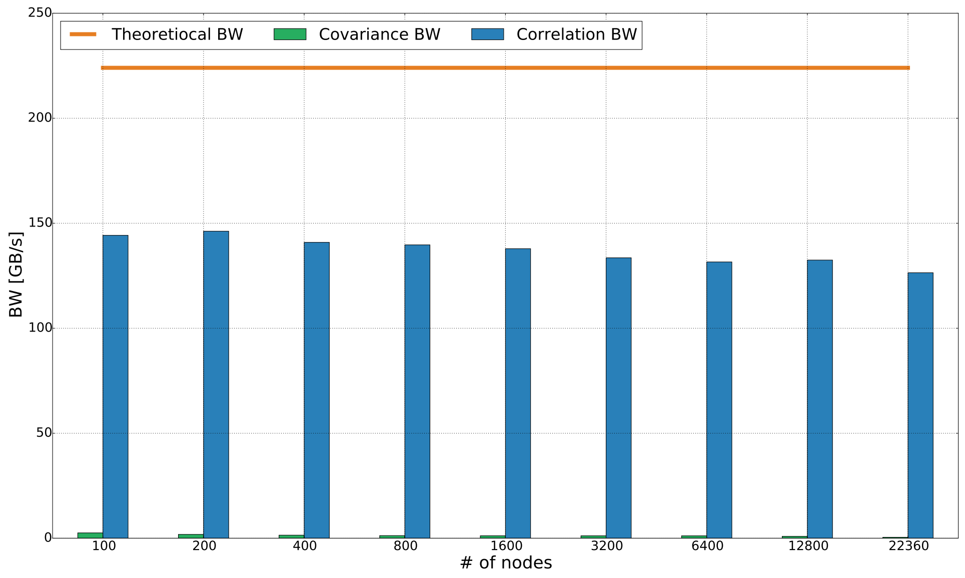


Figure 18: Effective bandwidth achieved by the two kernel functions defined in the algorithm.

Figure 18 shows the effective bandwidth of each kernel function defined, we can see that the bandwidth achieved by the covariance kernel function is very limited compared with the one achieved by the correlation kernel function. This is due by the fact that the covariance kernel function has to execute a higher number of instruction that don't involve the global memory. Te high bandwidth of correlation kernel function is justified by the fact that the kernel function that compute the correlation is more memory addicted than the kernel function that compute the covariance.

CHAPTER 5

DYNAMIC NETWORKS VISUALIZATION

In this chapter we present the second part of this thesis work, composed by a visualization tool developed to show the dynamic networks computed with the program presented in Chapter 4. The main idea behind this tool is to improve the understanding of DNs, including how they evolve during the observation period and how the nodes interact each others. This tool focuses on brain network visualization.

In the context of brain networks, scientists were previously unable to interact with the data because the problem parameters were hard-coded in the similarity computation. This means that the similarity matrices were filtered before the visualization processes. We propose a different approach to this problem: we leave one of the two problem parameter free. In our case, the similarity threshold can be fixed at run time by the user. We are able to achieve this aspect thanks to a considerable speed-up of the similarity computation and thanks to the availability of the whole similarity matrix for each observed instant. By having all of the results computed, the process that generates the edges among nodes could be performed at run time just before the visualization. The threshold value chosen can change the shape of the network, which consequently affects the results derived by the analysis applied to the networks computed.

Behind this tool there is a big database with all the data precomputed, all these data are made available to the user to be selected for the visualization process. The interactivity of the tool we present allows the user to add and remove different versions of the data in the database and keep track of different versions of the networks.

5.1 Visualization Considerations

The visualization of DN presents different challenges:

- The amount of data to process makes the visualization challenging. This great amount of data to precess requires a high performance system to have a real time rendering of the information. For this reason, we decided to make the computation of the similarity measure adopted to define the network off-line and separately from the visualization system. The reason why we divided computation and visualization is because computation time for the similarity, when the number of input nodes is considerably high, doesn't allow the creation af a real time rendering of the results.
- The visualization tool that we are presenting want to be interactive, so the user can play with the parameters of the problem and see how the results change. This aspect of the tool increase significantly the amount of data to compute. This is a direct consequence of the similarity threshold as free parameter. In fact, we have to save the whole similarity matrices without filtering the results.

The two points just described show which are the main challenges that we had to go through during the developing phase of the visualization tool.

5.2 Visualization Tasks

This tool aims to simplify DNs analysis. During the development process we tried to understand the most important tasks that neuroscientists want to perform on the data computed.

- **T1 - Dynamic Network Exploration**: One of the main tasks that neuroscientists want to perform is the ability to navigate through the network. With the possibility to zoom and move through the network they can receive more details about nodes and edges. An important aspect is the possibility to select a node and read all the information about it, including outgoing edges with relative correlation value and node position in the brain. The absolute node/edge position in the brain is important to neuroscientists to see which brain regions are active during the experiments.

- **T2 - Dynamic Threshold Update**: Neuroscientists also want to interact with the parameters of the problem. By updating the similarity threshold at run time, the shape of the network changes and domain experts can immediately see the differences. This dynamic update means that the process of edge definition occurs at run time after the threshold is fixed.

- **T3 - Dynamic Networks Comparison**: Due to the two degrees of freedom of the problem addressed, the number of networks that can be generated is considerably high. For this reason, domain experts want to compare the different network versions.

- **T4 - Local Analysis**: In addition to **T1**, neuroscientists want to be able to perform a local analysis of brain sections without considering interconnections with other parts of the brains. This because the brain is divided into regions that have specific functions and would be interesting analyze them separately.

- **T5 - Future Complex Analysis**: Theses kinds of brain studies are relatively new, neuroscientists are still trying to understand what the best models to apply are. For this reason, we developed a simple and flexible tool that can be easily expanded. One example of these kinds of studies is the extraction of social communities using the algorithm presented by Berger-Wolf et al. [9; 27].

The visualization tool presented is a base work that could be expanded with any new analysis performed by neuroscientists in the next years. The application of DNs to brain is a new trend and for this reason much of this field as gone unexplored. Therefore, we want to keep the tool flexible to this kind of future additions.

5.3 Visualization Tool

The visualization tool we developed is a web-based application, so it can run on every browser without particular dependences. The main technologies used are the classical web languages: HTML, CSS and JAVASCRIPT, including the visualization library D3[1], which is JS based.

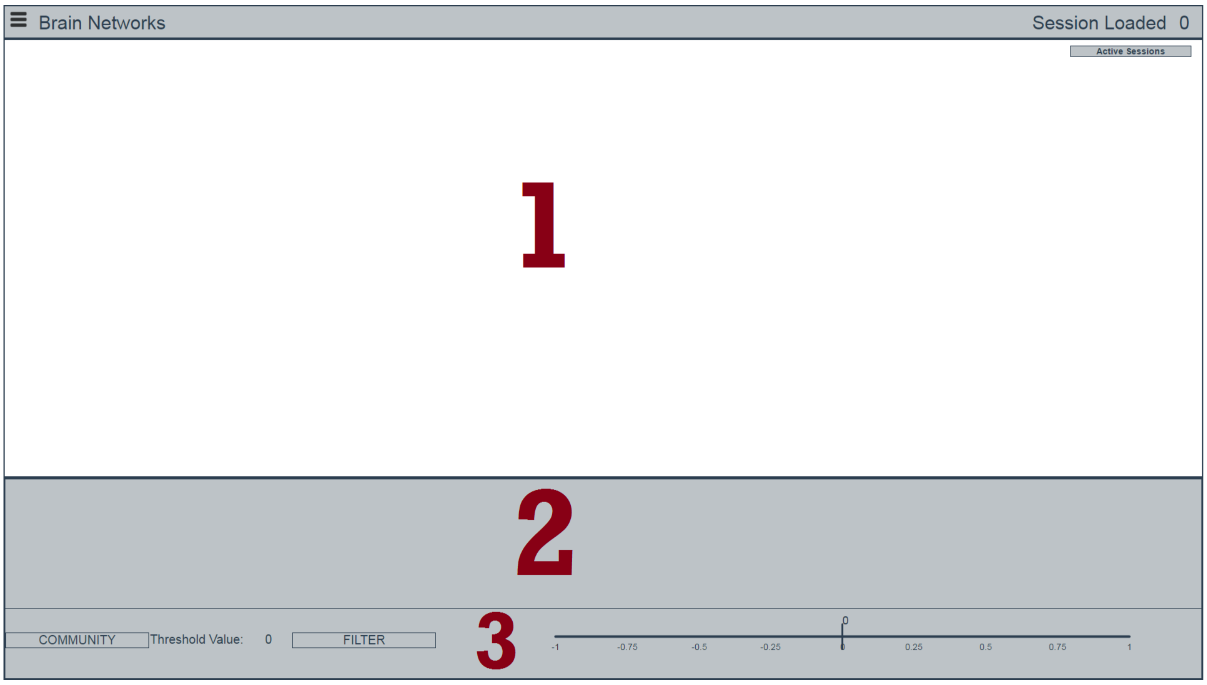


Figure 19: Tool interface, main page of the interactive visualization tool developed to display dynamic brain networks.

5.3.1 Visualization Prototype

The tool is composed by a web page that is divided into three main sections. One is used for network visualization, one for the timeline and the third for data filtering. Figure 19 shows the main page of the web application. There is a header with all the general information: on the top left corner we have a menu button and on the top right there is an info box that contains

[1]D3 examples and documentation can be found here: `https://d3js.org/`

all the information related to the data loaded in the system. Each main section of Figure 19 is used for:

- **Part 1**: space adopted for the network visualization. Inside this section the nodes and edges will be drawn. The user can navigate through and see how the DN is structured for the time selected.
- **Part 2**: space used to visualize a timeline representing the observation period. The input data, as described previously, is characterized by an observation period where each instance is associated with a network. In this section a time axis is shows to the user is order to select the time instance to visualize.
- **Part 3**: feature used to select the correlation threshold to apply at the data loaded. The threshold can be changed at any time, and the filtering is applied to any data loaded in the system.

5.3.2 Session Manager

This tool keeps track of different versions of the data that are computed using the algorithm presented in Chapter 4. For this reason, the concept of session is introduced, which represents a version of the data that can be loaded and visualized in the system. The different sessions can be managed using the menu that appears on the left side of the page.

Figure 20 shows the session menu, which is divided into two different parts. The top one has three buttons that are used to manage the sessions, and they perform creation, deletion, and reset of the data. The last button, *Reset*, is used to remove the loaded session from the system

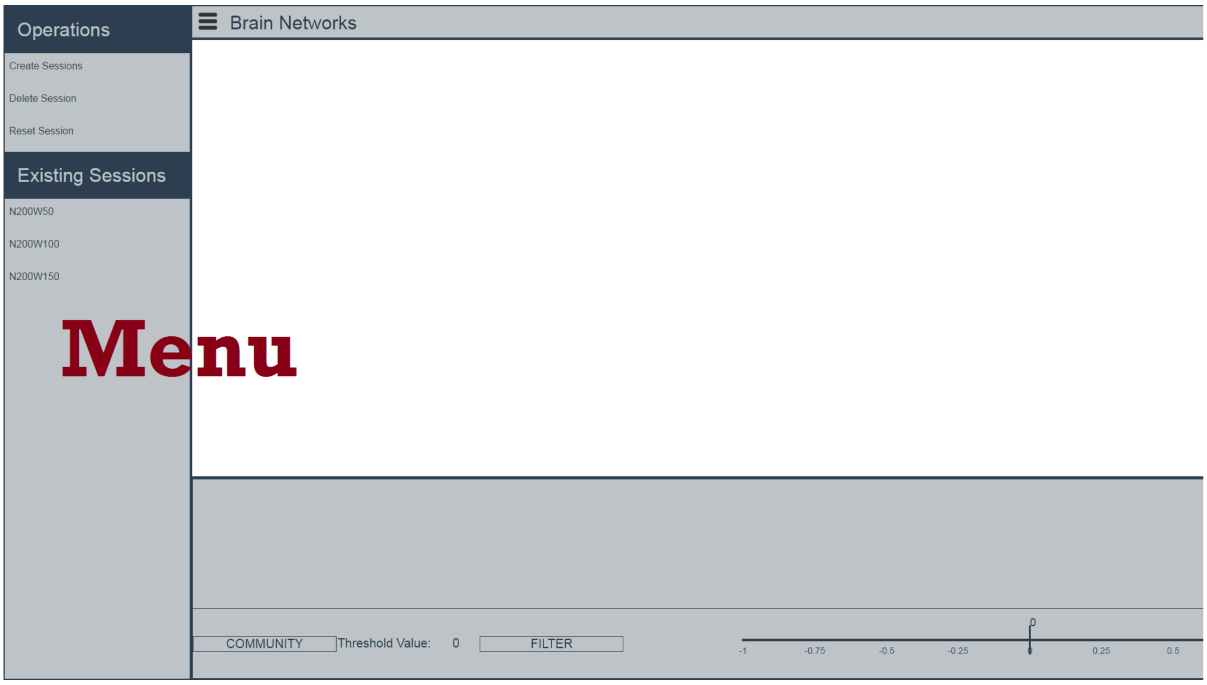


Figure 20: Tool menu, used to manage the data sessions. It allows the creation, deletion and load/unload of the data in the system.

and start the visualization from zero. The creation/deletion of a session consists in add/remove an entry in an on-line database where all the sessions are recorded. During the creation phase, the system ask the user for a folder name where the session data will be uploaded manually in the web-server. This process may seem tedious, but since the mole of data could be very high it is better to leave the uploading to an external program which is optimized for this kind of work.

The bottom part of the menu is composed of the sessions list created by the user. By clicking on the name of the session the system will load the data of the one selected. As stated before,

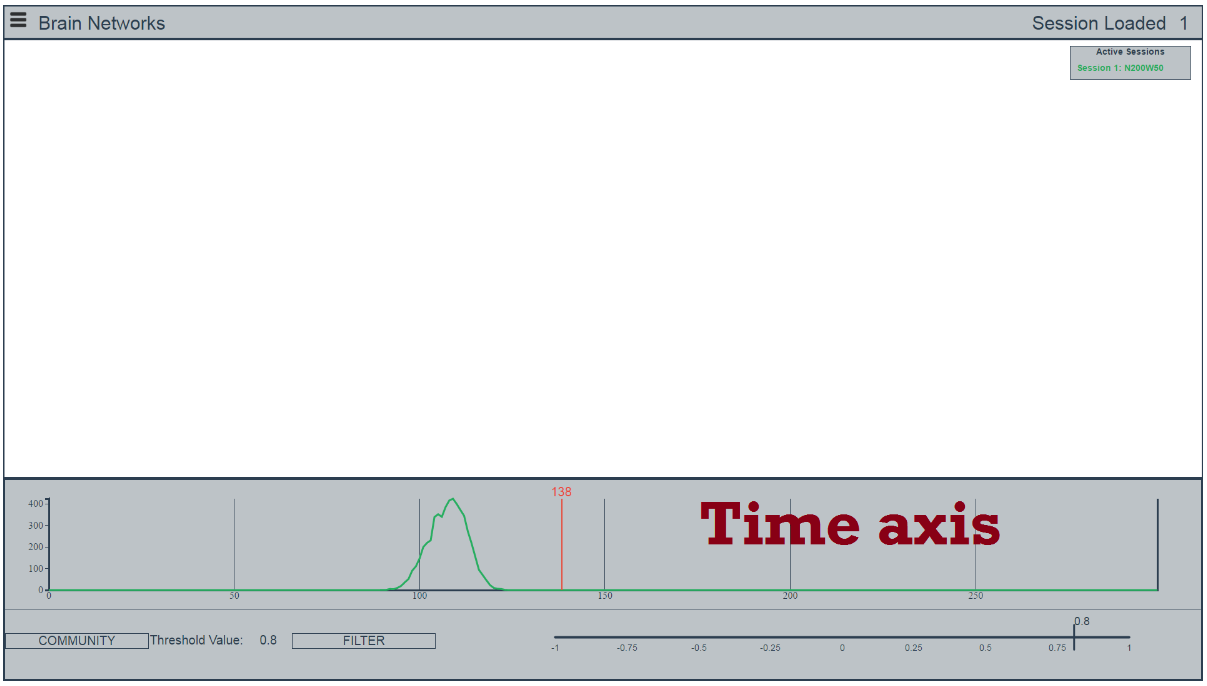


Figure 21: Data filtering, process that take the data loaded in the system and filter them using the threshold set by the user. It shows the time axis with the number of sedges for each time instant.

the system is able to manage more than one session at a time. In fact the user can compare two different data versions. The loaded sessions are shown in the main page (Figure 19) on the top right corner. Each one is represented by a color that will be used by the framework to visualize the data associated to that session. When a session is loaded, all the correlation matrices for each instance, saved in text files (Phase **6** of algorithm in Chapter 4), are loaded in the memory. In this way the filtering process will be faster when the user changes the threshold.

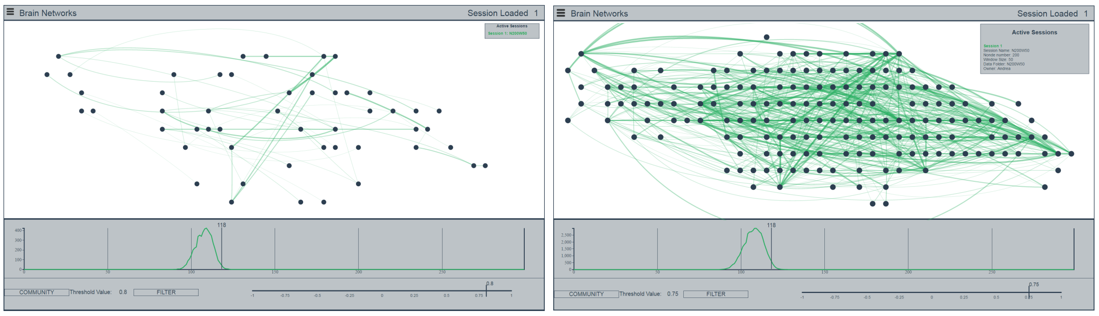


Figure 22: Different threshold networks, examples of network displayed with different correlation threshold,the left image has a greater threshold so less edges are present in the network, while the right image has a lower threshold and more edges are created for the network.

5.3.3 Data Filtering

Data filtering is performed setting the threshold using the slider on the right and clicking in the *Filter* button [**T2**]. The filtering process defines the network as follows: each node pair that have correlation greater than the threshold becomes an edge of the network at the time instant where the correlation is higher. Figure 21 shows how the tool appears after the filtering of the data loaded in the system. On the timeline is plotted a graph that represents the number of edges during each time instant. In fact on the $x - axis$ there is the time component while on the $y - axes$ there is the number of edges. The red vertical line represents the mouse pointer that moves over the timeline, the red number on top of the line is the time instant that the mouse is pointing. Clicking with the mouse on the timeline the system will draw the network for the time instant selected. Figure 22 shows an example of two networks visualized with two different correlation threshold.

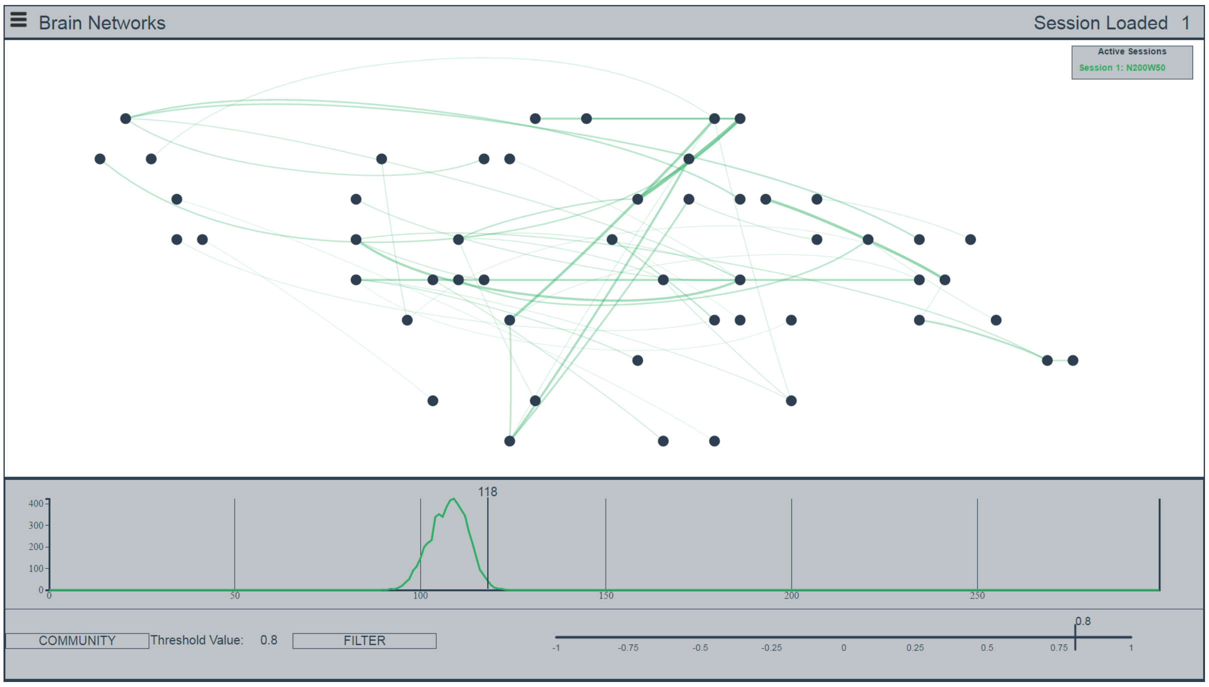
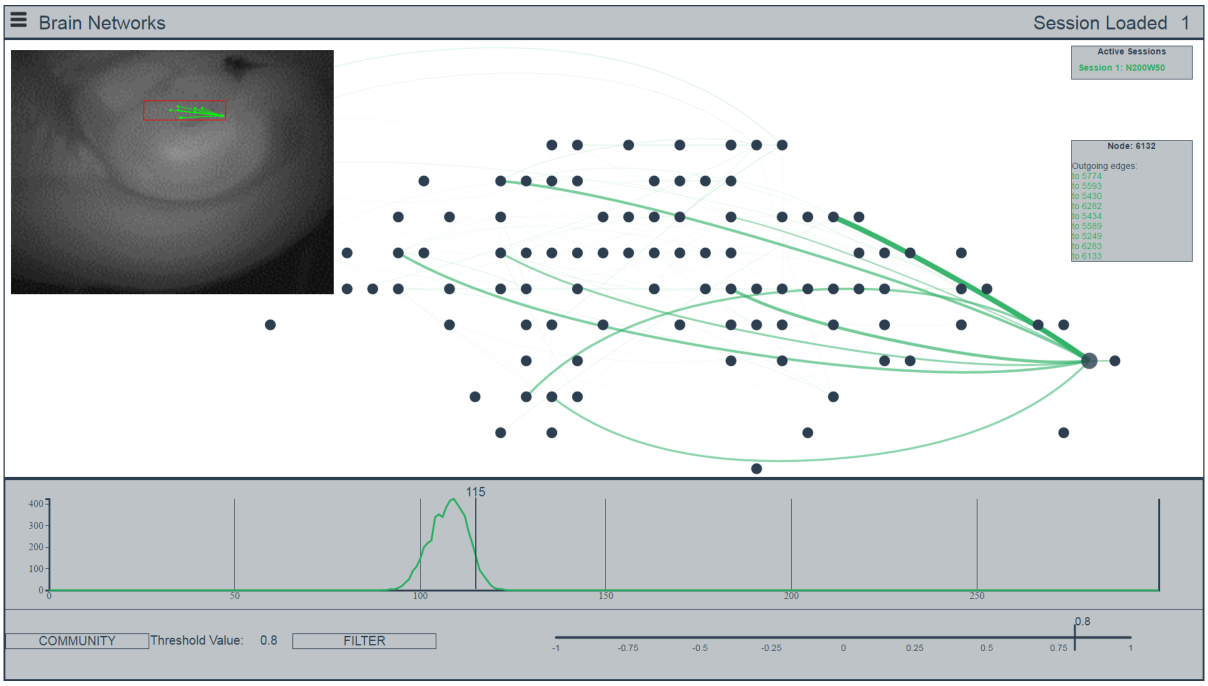


Figure 23: Network visualization, rendering of the network created by the filtering process, it is showed the network associated to the time instant selected in the time axis.

5.3.4 Network Exploration

Figure 23 shows the visualization of a network for one session data, the circles represent the brain cells, and their position in the space represents the position they have in the brain with the respect to the other nodes. The visualization shows only the part of the brain that contains edges, in fact the red rectangle in the brain image on the left part represents the portion of the brain visualized, this means that out of that rectangle there are no edges. The scale of the network is performed autonomously by the system, moreover the network could be moved and zoomed by the user with the mouse pointer [**T1**]]. The stroke and the opacity of the edges are



Session 1: N200W50

Node: 6132

Outgoing edges:
to 5774
to 5593
to 5430
to 6282
to 5434
to 5589
to 5249
to 6283
to 6133

Threshold Value: 0.8

Figure 24: Network mapping, it shows the solution we adopted to map the network drawn on the brain. This mapping allows the identification of the absolute position of nodes in the brain.

directly proportional to the correlation that two nodes linked by the edge have, so the bigger and the more opaque the edge is, the greater is the correlation among the two nodes.

When the network is visualized one of the main problems that could happen is the presence of a high number of edges, so the visualization could result in chaos. To give the edges a meaning we want to propose a visualization system that maps the edge of the network on the brain image. This is very helpful to understand the position in the brain and what we looking at. We decided to adopt a mapping to avoid the overlapping between the brain image and the network computed since it can results is confusional and be difficult to understand.

Figure 24 shows how we implemented the map between a part of the network and the brain. If the user goes over a node with the mouse, the system draw the node and the edges related to the node selected in the brain. This shows the exact position of the edges on the nodes in the brain. Moreover some important information are showed when the user goes over the node, like connections and correlation relative to the links, besides the nodes position in the brain.

5.3.5 Network Comparison

One of the main characteristics that our system has is the comparison between different version of networks [**T3**]. When more than one session is loaded in the system the behavior of the system is the same described in the previous sections, but everything is performed for all the sessions loaded. So on the timeline it is possible to see a number of graphs equal to the number of sessions loaded.

Figure 25 shows how the visualization tool appears when more than one session is loaded, the network is represented by different colors and also the map on the brain is represented by the color associated to the session. In this way the user can see the differences for the time instant selected on the timeline.

5.3.6 Additional Future Analysis

The tool presented in this chapter has the aim to simplify the comprehension of the results derived by the DN applied to brain. Since this kind of application is pretty new the scientists are still working on models to apply on the networks computed to extract relevant information. For this reason we would need to keep the implementation of the visualization minimalistic and

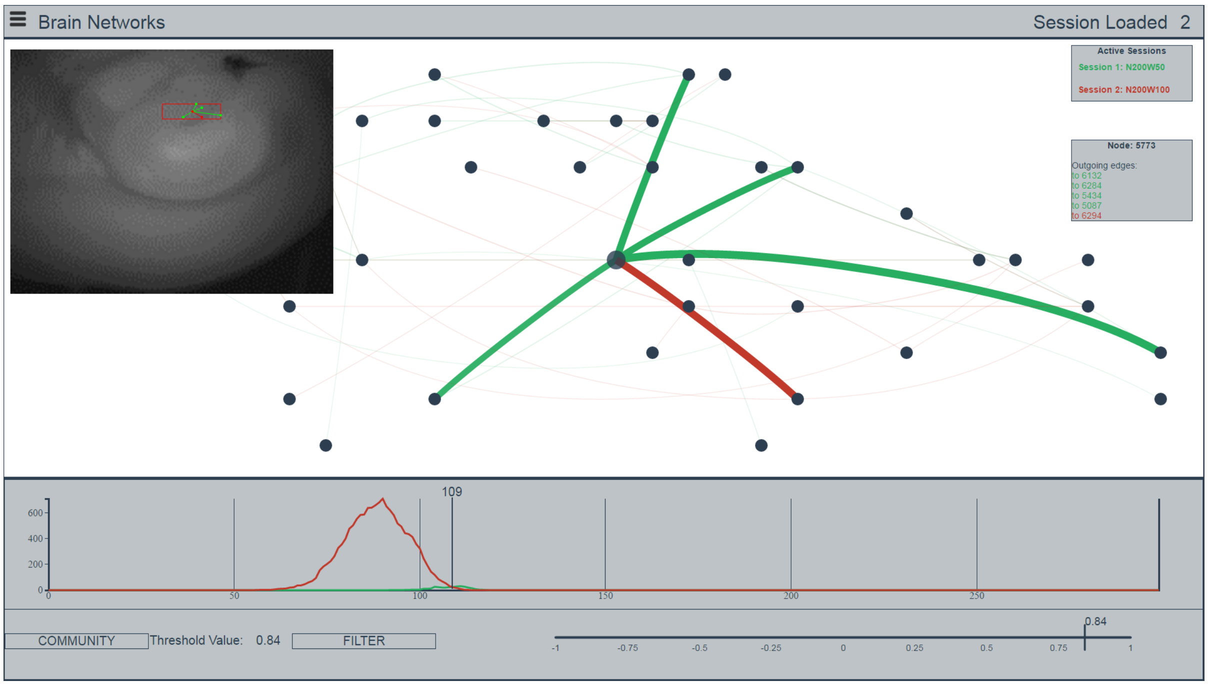
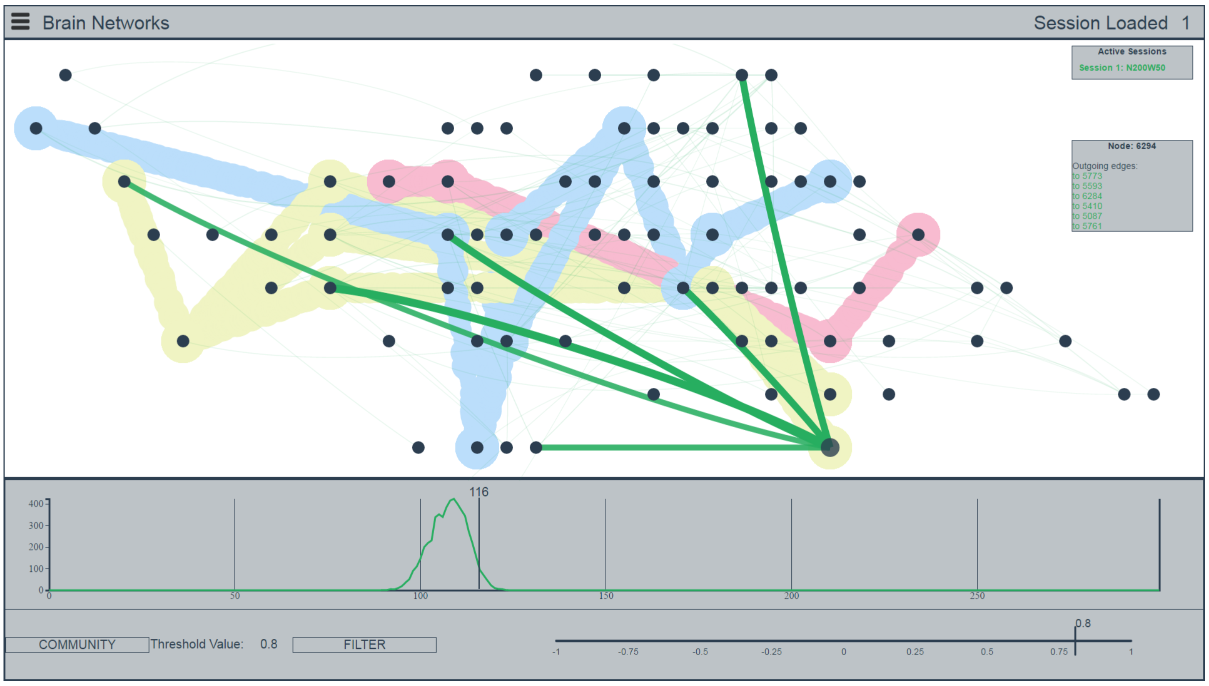


Figure 25: Network comparison, it shows the comparison between two different data session loaded in the system, each one is represented by a color in the network.

simple to increase the flexibility of the system, so in the future the addition of new visualization features will be easy.

One possible example of analysis performed is the communities extraction using [9] algorithm. The communities are showed overlaid on the network visualized in the system, this example shows how our visualization tool could be enrich with new kind of data derived from the network. For this reason we keep the flexibility as first requirement for the system [**T5**].



Active Sessions

Session 1: N200W50

Node: 6294

Outgoing edges:
to 5773
to 5593
to 6284
to 5410
to 5087
to 5761

Figure 26: Communities example, a possible analysis to display is the the communities extraction from the raw data. The communities are showed overlaid on the network visualized, each color in the image show a different community.

CHAPTER 6

CONCLUSIONS

We presented an interdisciplinary work able to compute DNs and visualize them in a clear and easy way to make the results understandable. During the development process we encountered some bottleneck that slowed what we expected at the beginning, in fact the high computational power of the GPU is limited by the slow writing speed of the HDD used in the test environment. This doesn't really affect the visualization process since we presented an off-line tool that shows network, and all the data are precomputed. The real bottlenecks occurs when external analysis have to be performed on the data computed. This saving process can be very long because all the data must be saved and only then analyzed. In fact, what is highlighted by the results is that the similarity computation time per se (without saving time) remains very low with the increasing of the input nodes number. That was the object of the work presented.

6.1 Future Work

Taking advantage of the low computation time for the similarity matrices, the main future work that could be developed is the creation of a more complex pipeline that make the results analysis on-line. This means that the GPU program wold be included in the pipeline as first step of it. Every step of the pipeline is executed on the same context, so the results computed by the GPU are resident in the memory. This aspect avoid the saving of the results on the local storage, avoiding to spend most of the time saving the results. However, the mole of data computed is very high and sometimes the machine global memory is not sufficient to contain all the data computed. In fact in these cases is necessary to filter the results, keeping only the most relevant for the analysis. Now we avoid the filter of the data for one reason, give a clear idea of all the results computed, but if the GPU computation becomes on the first step of a pipeline, the results are analyzed in the execution context of the GPU and the analysis will take only the interesting data needed.

Another possible further work is the implementation of different similarities measures in the program so that the user can chose which one compute on GPU. We have seen that the computation on GPU is very powerful and the expansion in this direction can increase the level on analysis of the results. Different similarities means also different version of results, in some field the application of DN is pretty new, and scientists don't still know which one is better for the context of application, so in this way would be possible to compare and find the best measure to adopt.

Regarding the visualization tool used to show the networks during the time, the most interesting future work is to include more analysis results. Allow the domain experts to select which results visualize and create an overlaid visualization that shows it on the network. In this case the work depends on the kind of future analysis will be performed on the resuls computed.

APPENDIX

D3 LIBRARY

Data Visualization is one of the new sciences that aims to communicate information using graphs and draws. This new science needs powerful tools that must be able to process data quickly and make them easily usable by the programmer, achieving a good final visualization result. One of the new tool used in the data visualization world is D3, D3[1] is a JAVASCRIPT based library that allows to create vectorial components in a web page. JAVASCRIPT is the most installed programming language in the world, this help the growing of libraries like D3, moreover all the modern browser nowadays are able to render Scalable Vectorial Graphics (SVG), even mobile devices. The vectorial components allows the creation of complex charts accessible by most of the Internet user. D3 born from the library PROTOVIS[2], which is considered the predecessor of it, both are invented by Mike Bostock as visualization tools to manipulate web pages component, like HTML or SVG components, and web pages styles, like CSS. D3 is a powerful tool that stay in the middle between the data manipulation and the data visualization, it simplifies both the processes thanks to simple functions that could be used by the programmers to load and render the data desired.

[1]D3 examples and documentation can be found here: `https://d3js.org/`

[2]PROTOVIS website: `http://mbostock.github.io/protovis/`

CITED LITERATURE

1. Ananthanarayanan, R. and Modha, D. S.: Anatomy of a cortical simulator. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing, page 3. ACM, 2007.

2. Plesser, H. E., Eppler, J. M., Morrison, A., Diesmann, M., and Gewaltig, M.-O.: Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In Euro-Par 2007 parallel processing, pages 672–681. Springer, 2007.

3. Buck, I.: Gpu computing: Programming a massively parallel processor. In Code Generation and Optimization, 2007. CGO'07. International Symposium on, pages 17–17. IEEE, 2007.

4. Wen-Mei, W. H.: GPU Computing Gems Emerald Edition. Elsevier, 2011.

5. Sanders, J. and Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010.

6. Shi, L., Wang, C., and Wen, Z.: Dynamic network visualization in 1.5 d. In Visualization Symposium (PacificVis), 2011 IEEE Pacific, pages 179–186. IEEE, 2011.

7. Rubenstein, D. I., Sundaresan, S. R., Fischhoff, I. R., Tantipathananandh, C., and Berger-Wolf, T. Y.: Similar but different: Dynamic social network analysis highlights fundamental differences between the fission-fusion societies of two equid species, the onager and grevys zebra. PloS one, 10(10):e0138645, 2015.

8. Llano, D., Ma, C., Stebbings, K., Di Fabrizio, U., Kenyon, R. V., and Berger-Wolf, T.: A novel dynamic network analysis reveals aging-related fragmentation of cortical networks in mouse. Unpublished manuscript, 2016.

9. Berger-Wolf, T. and Kempe, D.: A framework for community identification in dynamic social networks. In Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM Press, 2007.

10. Przytycka, T. M., Singh, M., and Slonim, D. K.: Toward the dynamic interactome: it's about time. Briefings in bioinformatics, page bbp057, 2010.

CITED LITERATURE (continued)

11. Beck, F., Burch, M., Diehl, S., and Weiskopf, D.: A taxonomy and survey of dynamic graph visualization. In Computer Graphics Forum, 2016.

12. Parallel programming and computing platform — cuda — nvidia. `http://www.nvidia.com/object/cuda_home_new.html`. Accessed: 03-02-2016.

13. Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C.: Gpu computing. Proceedings of the IEEE, 96(5):879–899, 2008.

14. Holme, P. and Saramäki, J.: Temporal networks. Physics reports, 519(3):97–125, 2012.

15. Holme, P.: Modern temporal network theory: a colloquium. The European Physical Journal B, 88(9):1–30, 2015.

16. Toth, D. J., Leecaster, M., Pettey, W. B., Gundlapalli, A. V., Gao, H., Rainey, J. J., Uzicanin, A., and Samore, M. H.: The role of heterogeneity in contact timing and duration in network models of influenza spread in schools. Journal of The Royal Society Interface, 12(108):20150279, 2015.

17. Chen, S., Ilany, A., White, B. J., Sanderson, M. W., and Lanzas, C.: Spatial-temporal dynamics of high-resolution animal networks: What can we learn from domestic animals? PloS one, 10(6):e0129253, 2015.

18. Valdano, E., Poletto, C., Giovannini, A., Palma, D., Savini, L., and Colizza, V.: Predicting epidemic risk from past temporal contact data. PLoS Comput Biol, 11(3):e1004152, 2015.

19. Smith, S. M., Miller, K. L., Salimi-Khorshidi, G., Webster, M., Beckmann, C. F., Nichols, T. E., Ramsey, J. D., and Woolrich, M. W.: Network modelling methods for fmri. Neuroimage, 54(2):875–891, 2011.

20. Cattaneo, R., Natale, G., Sicignano, C., Sciuto, D., and Santambrogio, M. D.: On how to accelerate iterative stencil loops: A scalable streaming-based approach. ACM Transactions on Architecture and Code Optimization (TACO), 12(4):53, 2015.

21. Wang, Y., Du, H., Xia, M., Ren, L., Xu, M., Xie, T., Gong, G., Xu, N., Yang, H., and He, Y.: A hybrid cpu-gpu accelerated framework for fast mapping of high-resolution human brain connectome. PloS one, 8(5):e62789, 2013.

CITED LITERATURE (continued)

22. Kim, S., Ouyang, M., and Zhang, X.: Compute spearman correlation coefficient with matlab/cuda. In Signal Processing and Information Technology (ISSPIT), 2012 IEEE International Symposium on, pages 000055–000060. IEEE, 2012.

23. Chang, D.-J., Desoky, A. H., Ouyang, M., and Rouchka, E. C.: Compute pairwise manhattan distance and pearson correlation coefficient of data points with gpu. In Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. 10th ACIS International Conference on, pages 501–506. IEEE, 2009.

24. Lee, D., Dinov, I., Dong, B., Gutman, B., Yanovsky, I., and Toga, A. W.: Cuda optimization strategies for compute-and memory-bound neuroimaging algorithms. Computer methods and programs in biomedicine, 106(3):175–187, 2012.

25. Ma, C., Forbes, A. G., Llano, D. A., Berger-Wolf, T., and Kenyon, R. V.: Swordplots: Exploring neuron behavior within dynamic communities of brain networks. Journal of Imaging Science and Technology, 2015.

26. van den Elzen, S., Holten, D., Blaas, J., and van Wijk, J. J.: Reducing snapshots to points: A visual analytics approach to dynamic network exploration. Visualization and Computer Graphics, IEEE Transactions on, 22(1):1–10, 2016.

27. Berger-Wolf, T., Tantipathananandh, C., and Kempe, D.: Community identification in dynamic social networks. Link Mining: Models, Algorithms and Applications, 2010.

28. Sekara, V., Stopczynski, A., and Lehmann, S.: The fundamental structures of dynamic social networks. arXiv preprint arXiv:1506.04704, 2015.

VITA

NAME	Andrea Purgato
EDUCATION	
	Master of Science in Computer Science, University of Illinois at Chicago, May 2016, USA
	Master Degree in Computer Science and Engineering, Sep 2016, Politecnico di Milano, Italy
	Bachelor's Degree in Engineering Of Computing Systems, Sep 2014, Politecnico di Milano, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	High level knowledge
	2014 - TOEFL examination (78)
	A.Y. 2015/16 One Year of study abroad in Chicago, Illinois
	A.Y. 2014/16. Lessons and exams attended exclusively in English
SCHOLARSHIPS	
Spring 2016	Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend in EVL Lab.
TECHNICAL SKILLS	
Basic level	C#, Unity, Access
Average level	Data science & Machine learning techniques, database & DBMS knowledge, Excel
Advanced level	Object-Orined Programming (Java, C++, Python, Smalltalk), code versioning (Git, SVN), Web-Programming(HTML, CSS, JS, D3), GPU-Programing (CUDA)
WORK EXPERIENCE AND PROJECTS	

VITA (continued)

Jan 2016 - May 2016	Research Assistant in EVL Lab.
Jan 2016 - May 2016	Part of the brain project at UIC
Sep 2015 - Dec 2015	Dynamic Network visualization project
	Design a web-based platform to visualize dynamic network of baboons.
Mar 2015 - Jun 2015	Task Scheduler for FPGA
	Design a scheduler of tasks for task-graph for hybrid system with FPGA and processors.

PUBBLICATIONS	
May 2016	Resource-Efficient Scheduling for Partially-Reconfigurable FPGA-based Systems