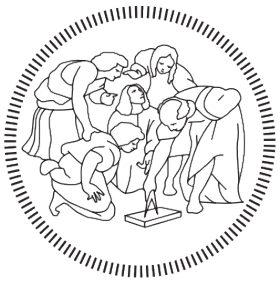
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

Master Degree in:
Computer Science and Engineering



POLITECNICO
MILANO 1863

Hardware Accelerated Framework for Spatio-Temporal Networks Analysis

Supervisor:
Prof. Marco D. Santambrogio

Master Thesis by:
Andrea Purgato

Student id:
836656

Academic Year 2015-2016

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

Edsger W. Dijkstra, 1970

Acknowledgments

This work has been realized in collaboration with the University of Illinois at Chicago, the first acknowledgments go to *CompBio* lab and *EVL* lab and all their members, in which most of the work has been developed. Then I have to thank my advisor Marco D. Santambriogio for the help given during these last months during the development process. Thanks also to *NECST* lab at Politecnico di Milano, to have provided a great work place where conclude this thesis work.

A special thanksgiving goes to the Italian crew in Chicago with which I shared many good memories during my permanence in the Windycity, thanks to Andrea, Roberto, Vittorio, Davide, Benedetto, Federico, Arianna, Chiara, Marc and Lorenzo for the amazing time in Chicago. I would also like to thank two special friends that supported me day by day during this period, 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 all of this possible. Thanks to have supported me every single day.

AP

Sommario

Questo lavoro focalizza il suo interesse sulla costruzione di un framework che ha lo scopo di facilitare l'analisi delle *Spatio-Temoral Networks*. Il framework sviluppato include una pipeline di operazioni in grado di elaborare i dati raccolti dai ricercatori, calcolare le networks e visualizzare i risultati interattivamente grazie ad una piattaforma web. Uno dei più grossi problemi della definizione delle *Spatio-Temoral Networks* è il tempo impiegato per la loro computazione, problema dovuto alla grande quantità di dati da elaborare. Per gli scienziati che vogliono effettuare studi su questo modello diventa quindi complicato svolgere complesse analisi in tempi brevi sui dati, rallentando così l'avanzamento degli studi. Grazie a dispositivi hardware come GPU siamo in grado di fornire una soluzione al problema della computazione. La prima parte di questo lavoro mostra come le GPU rappresentino una valida soluzione low-cost a questo tipo di problema. Mostriamo nei dettagli come approcciare il problema della computazione su GPU, sfruttando la tecnologia CUDA, analizzando le performance ottenute facendo diversi tests con dati reali raccolti dai ricercatori. Come anticipato, il framework presentato in questa tesi crea un flow diretto dal dato raccolto sul campo ad una rappresentazione visuale delle *Spatio-Temoral Networks* applicate al campo di provenienza del dato. Per questo motivo la seconda parte del lavoro si concentra sullo sviluppo di una piattaforma interattiva dove i ricercatori possono vedere ed analizzare le reti calcolate. In particolare, i questo lavoro di tesi, ci concentriamo sull'applicazione del modello *Spatio-Temoral Networks* al campo della neuroscienza, quindi al cervello.

Summary

The thesis work 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 this work is the creation of a framework that going from the raw data collected by domain experts arriving to a results visualization allows an interactive analysis of the spatio-temporal networks. The first part of the work focuses on the realization of an hardware computation system able to speed up the networks definition. The second part of the work includes an interactive tool to visualize the networks computed using the system 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 [2, 17]. 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 units. This kind of devices represent a low-cost solution to this problem since the level of parallelization they have is really high and the hardware architecture they have match exactly the requirements for the problem addressed. With the first part of this thesis, besides the hardware solution we present a detailed evaluation of the performance obtained during spatio-temporal networks definition.

The second part of the thesis 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 be visualized. 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 is composed by two different parts. One of them is used to compute the similarity measure needed to define the spatio-temporal networks (explained in Chapter 5). While, the other aims to visualize interactively the networks computed, leaving one degree of freedom to the final user (explained in Chapter 7). As already stated, 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 (details in Chapter 4). This framework wants to avoid the dependability problem given by that the different software are not able to communicate among each others.

Contents

List of Figures

List of Tables

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 [6, 28].

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), which presents a different architecture concept for the computation parallelization [20].

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 an hardware accelerated approach to speed up the STNs inference pro-

cess. In particular, we focus our interests on GPU devices, presenting a program that exploit the computational power of these. 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. [22] 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. [19] 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. [14]. The data used in this work come from fMRI exams taken on human beings and from experiments made on brain extracted from lab rat, which thanks to the exploitation of the flavoprotein autofluorescence they were able to catch the brain activity. 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. [4] 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. [18] 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. [3] define STNs 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:

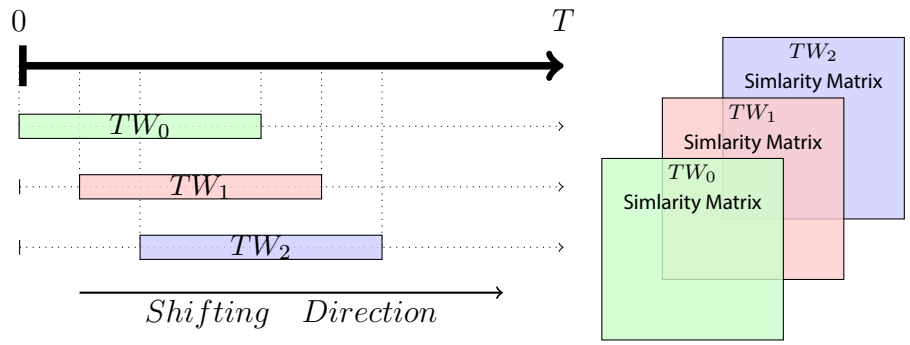
$$\Gamma := (G_1, G_2, ..., G_T) \tag{2.2}$$

$G_i : i \in [1, T]$ is a graph as defined in 2.1 where the sets of vertexes and edges change for every time instant i, and $V_i \subseteq \mathcal{V}$ since a vertex can be without edges.

To have a global view of the network we first assign a unique identifier to each node in the input dataset $\mathcal{V}$, 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 have in common the spatial domain space of the input data used to define the nodes of the network sine all them are extracted from brain. Moreover, each element of the input data, which represent the nodes of the network, present a time sequence of values coming from the input dataset that represent the activity of the element.

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 values 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 from now on



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 2.1: 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 2.2.

shorted as 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 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 long 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 during the STNs definition process. The size of the TW represents the second degree of freedom of STN problem.

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

Figure 2.2: 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.

Since the TW does not cover the whole observation period it is shifted by a fixed number of time instances, which is another parameter. However, if we set it for a shift of 1 then all the others are subsets (Figure 2.1 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 2.2, 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 visualization 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 enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)" [1]. 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 [20].

In the last years the computational power of GPU devices has been increased significantly, as reported by Owens et al. [16]. The key of this exponential growth for GPUs computational power is derived by the new paradigm of parallelism between computation units, [20]. 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.

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

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.

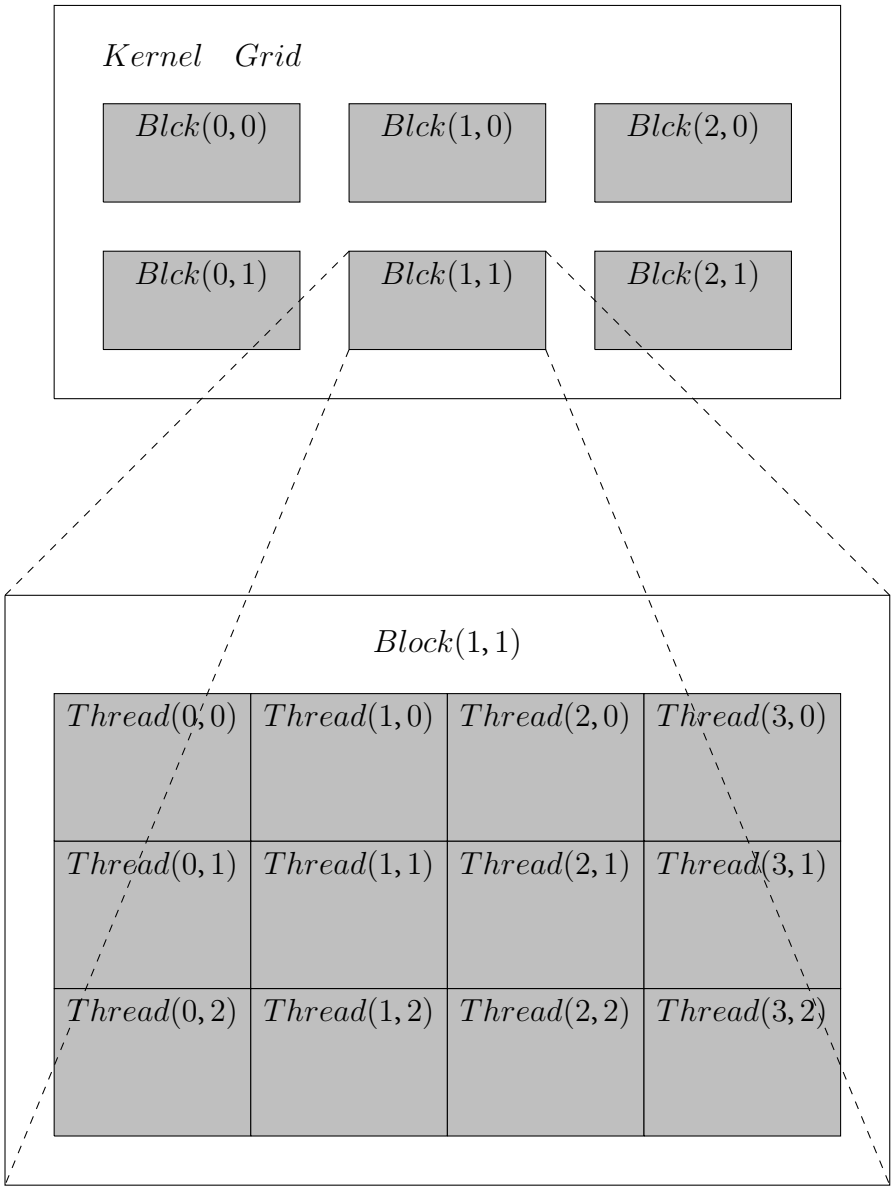


Figure 2.3: Kernel structure that must be defined for each GPU function.

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 2.3. 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:

- *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

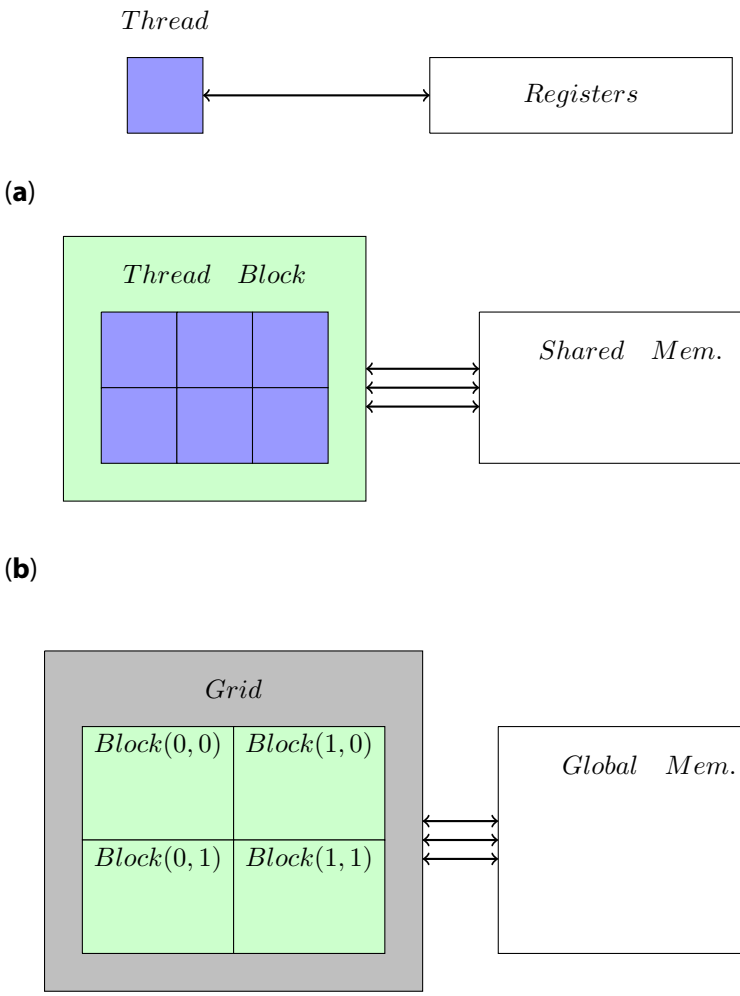


Figure 2.4: Kernel memory hierarchy defined for each kernel structure. (**a**): each thread has some registers used to make the computation. (**b**): each block may have a fixed amount of shared memory among the threads within the block. (**c**): at the grid level the access is only on the global memory of the GPU.

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.

When the kernel function is called the GPU create a memory hierarchy based on the structure of Figure 2.4. 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 Works

Spatio-Temporal Network model born to represent better the dynamic aspect of the interaction in a population of entities like animals, human or biological cells. 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. [11]. However, there is an important aspect that the common static graphs are not able to show, the time factor.

3.1 Spatio-Temporal Network Applications and Extraction

During recent years the applications of STNs is growing, and many different fields are looking at this new model. Holme et al. [11, 10] discuss in details the potential of this model and present a series of example contexts where STN model is used. The authors focus their interests in showing the powerfulness of STN applied to many different fields, they provide a very large overview of the kind of analysis that may be performed on that model.

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. [24] 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. So the computation of the network results different since we do not have a full perspective of the entities positions. Chen and Valdano [9, 25] applied proximity sensors to built dynamic networks. In particular, the first want to extract some information from the networks build about the behavior of domestic animal when in group. While the authors of the second article try to predict the spreading od epidemics based on the interactions occurred among the individuals subject of the study.

Different technologies, like GPS tracers, could be used to get absolute position of individuals in the domain space, Berger-Wolf et al. [4] 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. [23] 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. [14]. In this case thanks to this particular property, the stimulation of brain slices increase the luminosity of the active brain cells. In this case, however, the experiments require the cut in slices of the brain, so more difficult to perform on living beings.

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. [4] present a framework that using STN identifies the communities created by the subjects and the interactions among them. A community is defined by a set of entities that have a higher level of interaction among each other than with the extra-community entities. All the entities in a community are considered closer that with the other.

3.2 Spatio-Temporal Network Computation and Visualization

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. [14] present a work that uses the Pearson Correlation Coefficient. In other cases like proximity networks, the euclidean distance among individuals and the time spent close each other is a better measure for the similarity. 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. [7] 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. [27] 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. The authors of this article compute the Pearson Correlation Coefficient on fMRI data to define STN with a really high number of nodes. Another similar work that show how graphic processor can be used to compute pairwise similarity on matrix is proposed by Kim et al. [12], where different similarity measures are computed. The authors show a GPU program developed with CUDA called by a Matlab environment that speed up the similarity computation.

In addition, Chang et al. [8] show the decreasing of execution time for the Pearson Correlation Coefficient computation and Manhattan distance computation using GPU devices. The results presented show a performance increment up to $90\times$ for the Manhattan distance computation and up to $38\times$ for the correlation computation with the respect of classical CPU execution. Our work aims to confirm the importance of the acceleration provided by hardware devices ex-

ploiting 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. [13] present a formulation that is used to balance the computation on graphical devices to prevent the running out of resources. We decided to adopt this formulation in our work, all the details are presented in Chapter 5.

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. [3] 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. [15] proposes "SwordPlots" a new visualization technique used to explore brain networks with the focus on social communities visualization., "SwordPlots" shows different characteristics of the network visualized simultaneously so that the understandability of the results is increased. 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

Hardware Accelerated Framework

With this thesis we propose a full framework that want to create a complete pipeline going from the raw data collected to a final visual solution that allows the analysis of the STNs. The whole process is composed by different steps that are independent each others, in this way it is simple keep the program flexible to any future changes. Figure 4.3 shows all the steps that compose our pipeline. Below here we anticipate some information about each step.

1. *Data Preparation*: step one of the pipeline composed by a script program that aims to turn the raw data collected in a specific format that can be read by the hardware accelerated program developed at step 2. The *Raw Data* given as input of the pipeline are strictly related to the experiment performed, so they can present different format. For this reason this step is necessary to keep the hardware acceleration section independent from the initial data. Moreover, since each input dataset can be different from the others, this step requires a custom program to elaborate and prepare the data collected. The independence of the various steps from each others allows the substitution of the script based on the input dataset.

2. *HW Acceleration*: in this step of the pipeline are computed the similarity matrices used for the STN definition using GPU devices that can guarantee a high parallel computational power. This step is the most complex since the mole of data to process makes the managing of the resources not easy. The program computes all the similarity matrices saving them on storage so they can be used by the visualization tool or by other program dedicated to the results analysis.

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

Figure 4.1: 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).

3. *STN Visualization*: the last step of the pipeline is composed by an interactive web-based application used to visualize the STN computed with the GPUs. In particular this part of the work aims to help domain experts in the understanding of the evolution of the STN over the time. Thanks to the results format computed in the previous step the user has one degree of freedom in the STNs definition and visualization.

Between each step of the pipeline proposed in Figure 4.3 we can see the presence of a dataset. Each dataset, except the input one, has a fixed format so the different part of the framework can communicate easily and can be independent each other, so the substitution of one of them does not result in a complex operation if the new version respect these standards. Figure 4.1 shows the data format adopted between step one and two of the pipeline, in particular t is the observation time and v is the values that each node assumes over the time. The single value $v_i : i \in [0, P]$ is the value of the node i in the time instant associated to the row. Figure 4.2, instead, shows the data format adopted between step 2 and step 3, where each line is composed by: p_i, p_j, s_{ij}, which are respectively: the identifiers of node i, the identifiers of node j and the similarity between node i and node j.

4.1 Data Preparation

In this section we talk about the datasets used as test cases for this work and how we elaborated them to make them usable by the hardware program. All the data used are real and collected thanks to domain experts, which have provided them kindly to help the development of our work. This part of the work represents **Step 1** of the pipeline described previously (see Figure 4.3). The datasets adopted are two, both come from neuroscience world but from complete different branches.

$$
\begin{array}{lll}
p_0, & p_0, & s_{00} \\
p_0, & p_1, & s_{01} \\
\ldots & & \\
p_i, & p_j, & s_{ij} \\
\ldots & & \\
p_P, & p_P, & s_{PP}
\end{array}
$$

Figure 4.2: Standard data format accepted by the visualization tool program as input. p_i is the identifier for the first node of the pair, p_j is the identifier for the second node of the pair and s_{ij} is the similarity between the two nodes.

In both cases the purpose of apply STNs to the brain is to catch and study the interaction among neurons to extract more useful information about the brain. Following here we describe the two datasets, for convenience we gave them a name so it will be easy to distinguish between them.

1. *BioData*: dataset 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 coming from a rat lab. The activity of the brain 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 makes the active cells fluorescent, increasing the level of luminosity. The video that records the experiments is able to catch this luminosity change. For this specific dataset the images present a dimension of 172×130 pixels, for a total of 22360 pixels in each figure. The number of images, so the number of time instants, is 1000.

2. *HumData*: the other dataset is extracted from fMRI exams taken on human beings. The data coming from fMRI are composed by a series of files (*gif* format files), each one representing one time instant of the resonance duration. Each file, then, contains compressed within itself different images, these shows the activity of one specific slices of the brain, each slice subdivide the human brain on the z-axis as shown in Figure 4.4. The images of one slice in this dataset present a dimension of 79×79 pixels, for a total of 6241 pixels in each figure. The number of images, so the number of time instants, is 250.

Figure 4.5 shows an example of images extracted from both the dataset used in this work. We assume that each pixel of the images could potentially be a brain cell. So the similarity matrix will contain the similarity computed among each

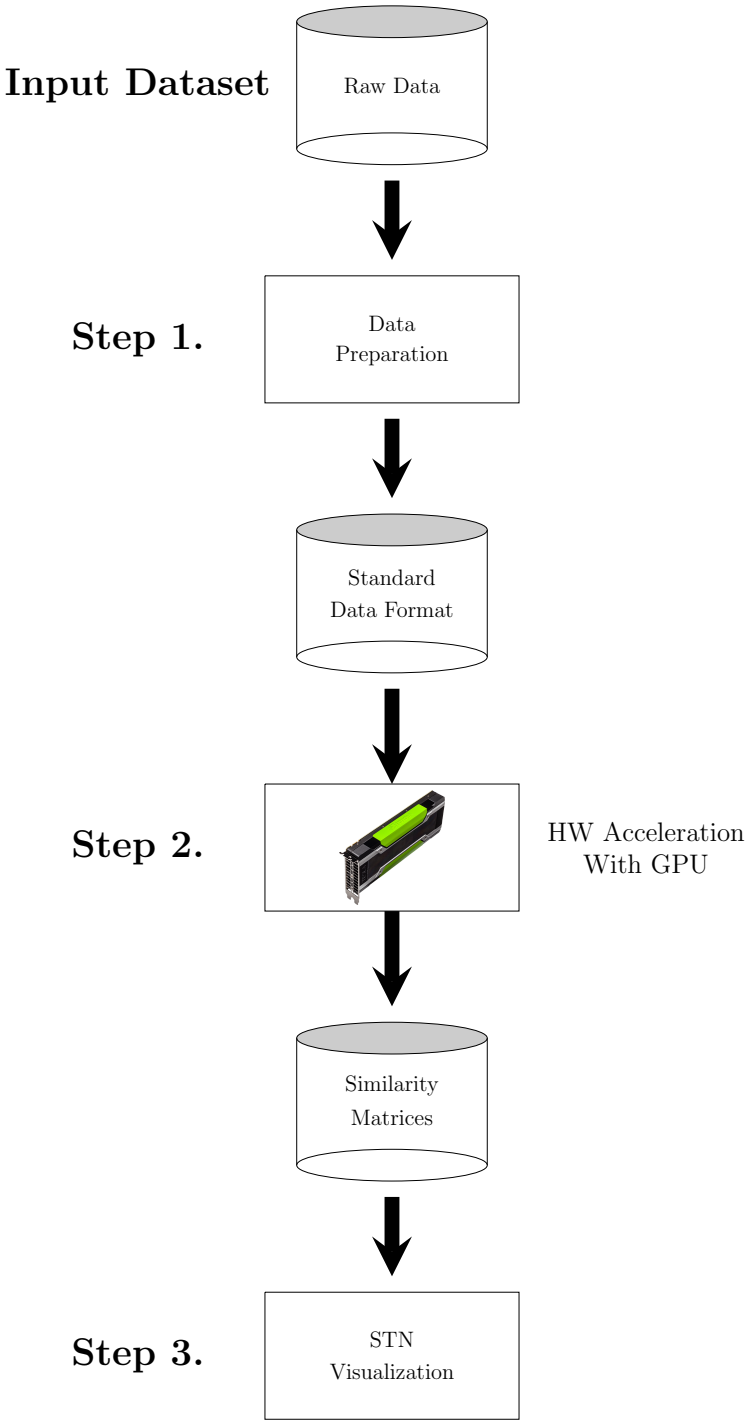


Figure 4.3: Diagram of the pipeline implemented in this thesis work.

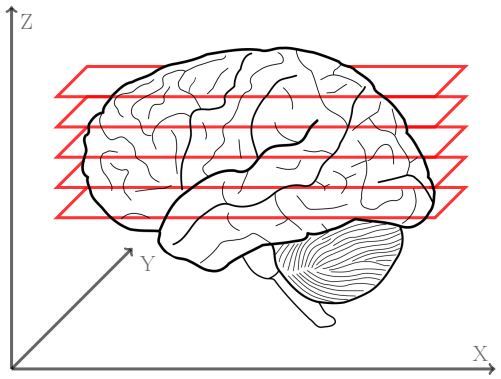


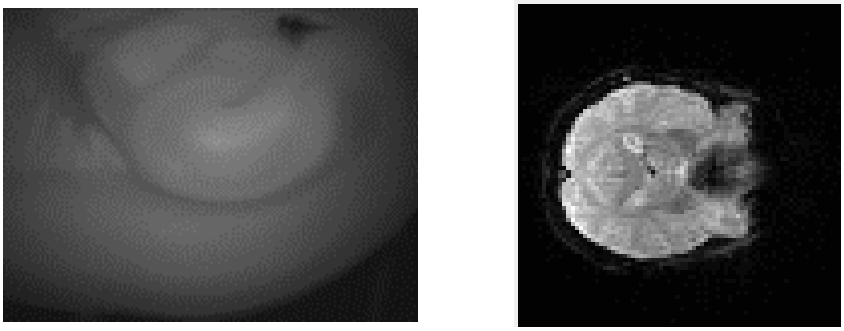
Figure 4.4: In this image is shown how the fMRI exam taken on the human partition the brain in different slices. Each red line represent one slice present in the fMRI results.

pair of pixels in the images. An important remark to make is that the brain cells, which are the nodes of the network, have a fixed position in space, so from now on we can talk about DN instead of STN which is more general.

The signal representing the activity of each cell is taken using the level of the gray color of each pixel. In Figure 4.6 is showed an example of sequence of images from the input dataset, for each pixel of the images we extract the hue values of the gray component of the pixel and we stored all of those in a text file. For the *BioData* the extraction of these values resulted easier because the images were already clean and not compressed, so we converted the images in ASCII saving the hue value in numeric format (value from 0 to 255, where 0 is black and 255 is withe). On the other hand, for the *HumData* we first had to extract the images from the file compressed containing all the brain slices, once the images had been extracted we performed the same operation done on the other dataset to extract the hue value of each pixel in each time instant. At this point, each node has a vector of values that represent the hue value of it for each time instant of the observation period that is used to compute the correlation between each pair of brain cells.

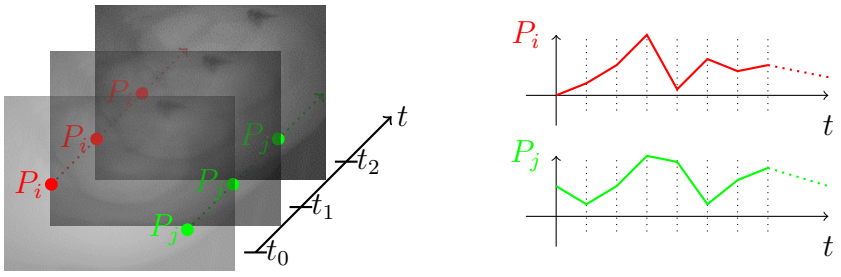
4.2 Hardware Acceleration

In this section we explain our considerations about the implementation of the program used to computes the similarity measure adopted for this problem, it represents **Step 2** of the pipeline of Figure 4.3.



(**a**) (**b**)

Figure 4.5: These two images are extracted from the two datasets used to test the framework. (**a**): image coming from the *BioData* dataset. (**b**): image coming from the *HumData* dataset and it represent only one slice of the human brain among all the slice that fMRI exam gives back in the results.



(**a**) (**b**)

$$
\begin{array}{lll}
t_0, v_{0,P_0}, \ldots & v_{0,P_i}, \ldots & v_{0,P_j}, \ldots \\
t_1, v_{1,P_0}, \ldots & v_{1,P_i}, \ldots & v_{1,P_j}, \ldots \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
t_T, v_{T,P_0}, \ldots & v_{T,P_i}, \ldots & v_{T,P_j}, \ldots
\end{array}
$$

(**c**)

Figure 4.6: Example of input dataset transformation in input data taken from the hardware accelerated program. In each image of the series, (**a**), each pixel is characterized by a discrete function that represent the hue value of the pixel itself, (**b**). All the pixels values are stored in a text file, (**c**), where on each row are stored the vales of each pixel (v_{k,P_i}) in the observation instant of the row.

4.2.1 Edge Definition: Node Similarity Measures

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

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 $[-1, +1]$, where +1 means the highest positive correlation between the two variables, -1 means the lowest negative correlation between the variables and 0 means no correlation between *X* and *Y*. PCC is computed as:

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

$Cov(X,Y)$ 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 Variables (RV). It is computed as:

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

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

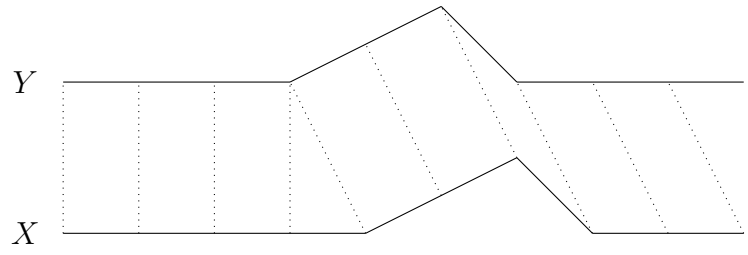


Figure 4.7: Dynamic time warping example, the dashed lines represent the match applied by the algorithm between the signals.

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 thanks to the *tau coefficient*. It is defined as:

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

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 4.7 shows an example of how dynamic time warping matching works 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.

4.2.2 Similarity Computational Model

We present now different computational models that can be used to compute the similarity in the DNs definition problem, we also show the advantages and disadvantages of each one explaining also which one we adopted and why. In this thesis work we decided to adopt the Pearson Correlation Coefficient (PCC) index as similarity measure. However many others version of this work can be easily developed implementing other similarity measures, comparing the results obtained with the one obtained here.

Similarity Computational Considerations

As already stated previously the computation of the similarity must occur for every pair of nodes present in the data collected. 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 time instant by time instant, without mixing the computation of different temporal entity.
- **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:

$$\begin{aligned}\Delta_{1,0} =& \frac{1}{(N+1)^2} \cdot \Bigg[N \cdot x_{N+1} \cdot y_{N+1} - N \cdot x_0 \cdot y_0 - \\ & - \sum_{i=1}^{N} x_i \cdot \Big(y_{N+1} - y_0 \Big) - \sum_{i=1}^{N} y_i \cdot \Big(x_{N+1} - x_0 \Big) \Bigg]\end{aligned} \tag{4.11}$$

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.

4.2.3 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.

4.2.4 Implementation Details

We present in this section the implementations we decided to develop for the hardware architecture adopted. As already stated we adopted as similarity measure the PCC index (Equation 4.1). With this work we propose two different versions: one hybrid where the computation is partially executed on the hardware device and partially on the CPU, on the other hand, the second version is fully implemented and executed on the hardware device.

Hybrid Version

The hybrid implementation splits the computation between the GPU and the CPU. We decided to implement this version to reduce the amount of data that must be transferred between the central memory and the device and to have a smaller device function so that a lower amount of resources is required. Starting from the formula of the PCC (Equation 4.1) we can see that to compute it we need the covariances between each pair of nodes. The covariance formula can be written as:

$$\begin{aligned} Cov(X,Y) &= E[X - \bar{X}]E[Y - \bar{Y}] \\ &= \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \mu_x) * \frac{1}{N}\sum_{i=0}^{N-1}(y_i - \mu_y) \end{aligned} \tag{4.12}$$

After valuating what is convenient to compute on CPU we decided to keep on CPU: the differences between the values and the average, and the standard deviation. In this way we are avoiding the computation of the average on the device so the function will be smaller and it will require less resources.

Algorithm 1 Function thats computes: the differences between the value and the average and the standard deviation for each node. This function is executed on CPU before the computation of the PCC on the hardware device.

```
Function HybridCpuFunction(data):
    for n ∈ [0, nNodes) do
        sum ← 0
        for f ∈ [0, timeWindow) do
            sum ← data[n * nNodes + f]
        end
        avg ← sum / timeWindow
        sum ← 0
        for f ∈ [0, timeWindow) do
            diff[n * nNodes + f] ← (data[n * nNodes + f] - avg)
            sum ← pow(diff[n * nNodes + f],2)
        end
        devstd ← sum / timeWindow
    end
end
```

Algorithm 1 shows the function used to compute the differences and the standard deviation on CPU. All the results computed are then moved on the device so that the PCC can be calculated as shown in Algorithm 2. What is important about Algorithm 2 is that it shows the computation for only one pair of node, identified by *x* and *y*, the function is executed then for every pair of nodes of the problem in parallel on the GPU processors.

Algorithm 2 Function that computes the correlation value for a pair of nodes, *x* and *y*. The computation occurs using the values already computed on the CPU previously (*diff* and *devstd*).

```
Function HybridDeviceFunction(diff, devstd, x, y):
    sum_x ← 0
    sum_y ← 0
    for f ∈ [0, timeWindow) do
        sum_x ← diff[x * nNodes + f]
        sum_y ← diff[y * nNodes + f]
    end
    avg_x ← sum_x / timeWindow
    avg_y ← sum_y / timeWindow
    correlation ← (avg_x * avg_y) / (sqrt(devstd[x]) * sqrt(devstd[y])
end
```

Full Hardware Version

The other version we present in this thesis work computes the PCC completely on the hardware devices. This version of the program needs the whole input dataset without any modifications loaded on the GPU memory. The first operation to perform is the computation of the covariance, we decided to simplify the computation of it as shown in Equation 4.13 so that we do not need to keep track of the averages to compute the differences, but we can directly goes on the covariance computation.

$$\begin{aligned} Cov(X,Y) &= E[X-\bar{X}]E[Y-\bar{Y}] \\ &= E[XY]-E[X]E[Y] \end{aligned} \tag{4.13}$$

After the covariance computation the function implemented calculates the correlation using the formula shown below:

$$\rho_{X,Y} = \frac{Cov(X,Y)}{Cov(X,X) * Cov(Y,Y)} \tag{4.14}$$

Algorithm 3 shows the computation of the correlation between two random nodes, identified by *x* and *y*. The function is executed fully on the target hardware device for every pair of nodes in the problem in parallel.

4.3 Spatio-Temporal Network Visualization

In this section are described all the considerations we made while developing the tool used to visualize the networks computed thanks to the GPU program. This tool focuses its attention to the brain networks visualization problem. 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 parameters 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.

Algorithm 3 Function that computes the PCC for a single pair of nodes, identified by *x* and *y*. It is executed on the target HW device for every pair of node in parallel.

Function *FullDeviceFunction(data, x, y):*

```
sum_x ← 0
sum_y ← 0
sum_xy ← 0
sum_xx ← 0
sum_yy ← 0
for f ∈ [0, timeWindow) do
    sum_x ← diff[x * nNodes + f]
    sum_y ← diff[y * nNodes + f]
    sum_xy ← diff[x * nNodes + f] * diff[y * nNodes + f]
    sum_x ← diff[x * nNodes + f] * diff[x * nNodes + f]
    sum_y ← diff[y * nNodes + f] * diff[y * nNodes + f]
end
avg_x ← sum_x / timeWindow
avg_y ← sum_y / timeWindow
avg_xy ← sum_xy / timeWindow
avg_xx ← sum_xx / timeWindow
avg_yy ← sum_yy / timeWindow
correlation ← (avg_xy − avg_x ∗ avg_y) / ((avg_xx − avg_x ∗ avg_x) ∗ (avg_yy − avg_y ∗ avg_y))
```

end

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.

4.3.1 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 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.

4.3.2 Network Visualization Key Points

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

- **Dynamic Network Exploration**: One of the main feature that neuroscientists want to have is the ability to navigate through the network. With the possibility to zoom and move through the network where 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.
- **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.
- **Dynamic Networks Comparison**: Due to the two degrees of freedom of the problem addressed, the number of different networks that can be generated is considerably high. For this reason, domain experts want to compare the different network versions.
- **Local Analysis**: Another important key-point is the ability to perform 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.

The visualization tool we present 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.

CHAPTER 5

GPU Implementation Details

In this chapter we explain the details and the important aspects of the GPU program implementation. One of the key point of GPU programming is the managing of the resources available on the device. The program has to avoid the running out of resources, and to do this it has to be able to balance the computation. We propose now the solution adopted to balance the computation in both the version described previously (hybrid and full hardware).

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 5.1 shows an example of matrix containing all the node pairs. Recalling Figure 2.3 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 5.1 into the structure of Figure 5.2 (example of 2D GPU structure). All the tests performed are run considering the full datasets, so using all the pixels in the images as potential nodes and all the time instants. In all the tests performed we fixed the time window at 50 frames.

The number of threads present in each dimension in the structure of Figure 5.2 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

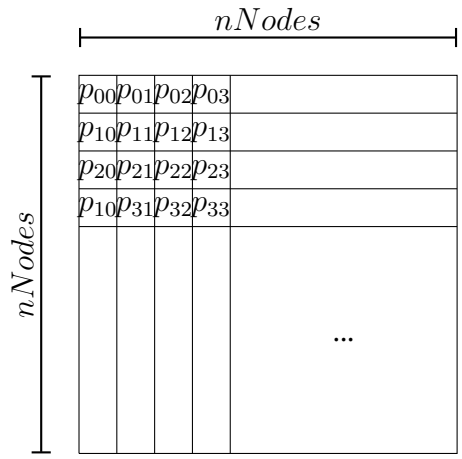


Figure 5.1: Matrix that contains each nodes pairs of the input data. $nNodes$ is the number of nodes in the input data and $p_{i,j}$ represent the node pair composed by nodes i and node j.

architecture. However, generally this 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 5.2 to obtain a final design like the one in Figure 5.3, where the $z-axis$ value represents the observation instant.

5.1 Computation Balancing

The hardware of the local machine impose some limitations. In particular, we focus on those imposed by GPU devices. Among those listed below we keep into account only the first two since the kernel functions defined for this problem do not require the usage of shared memory.

- **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.

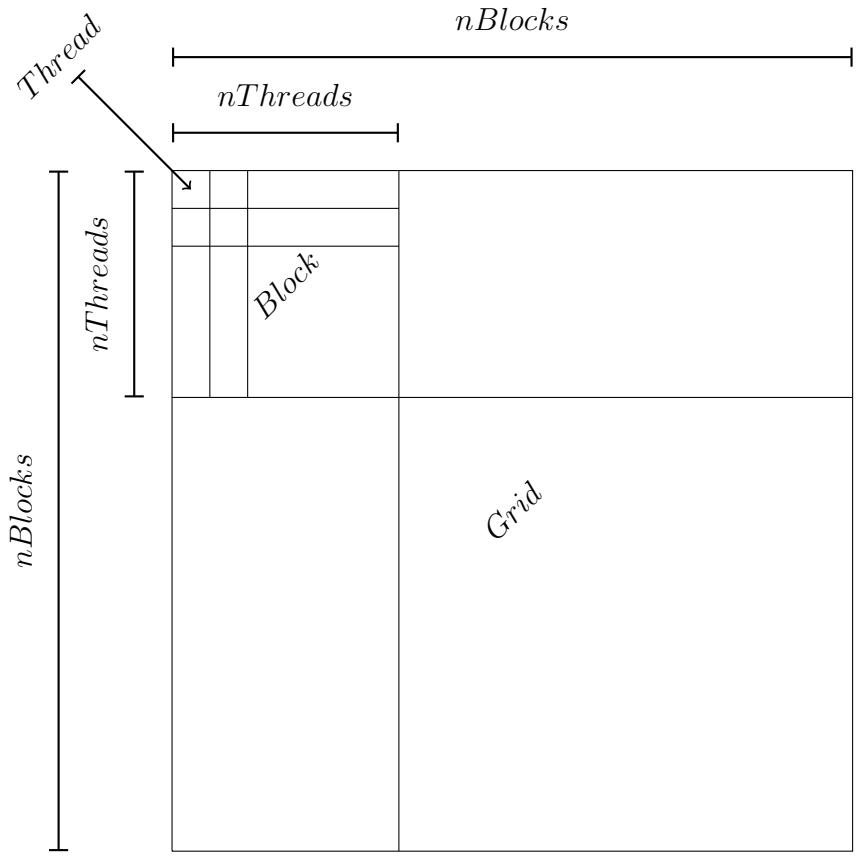


Figure 5.2: Kernel structure used for the kernel functions in the problem for one observation instant. $nBlocks$ represent the number of blocks in each dimension in the grid and $nThreads$ represent the number of thread in a single block per each block dimension. The structure used present a quadratic 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.

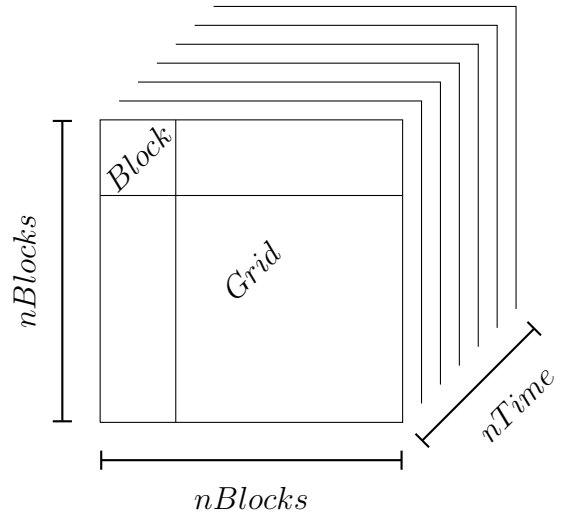


Figure 5.3: Kernel structure used for the kernel functions in the problem extend with the time component. The time is represented by the $z - axis$ and $nTime$ is the number of observation instants that compose the observation period in the input data.

- **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.

To deal with the limitation imposed by the architecture of the GPU we first have to query the devices available on the machine and read all the characteristics of them. These will be used later to scale the computation avoiding the running out of resources. The most interesting characteristics read on the GPU devices 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.

Algorithm 4 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 5 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
```

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

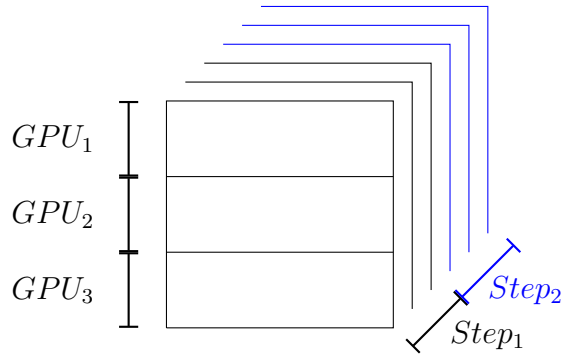


Figure 5.4: Example of computation split defined in phase 3 on a sample matrix, the machine in this case has three GPU devices.

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 5.3, we split over the $z - axis$). The number of steps is computed by the function defined in Algorithm 4, 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 5 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 5.3 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 5.4 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 al. [13], 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:

$$R_{block} * Blocks_{SM} \leq MaxReg_{SM} \tag{5.1}$$

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:

$$R_{block} = ceil(ceil(W_{block}, W_{allocation}) * T_{warp} * R_{kernel}, R_{allocation}) \tag{5.2}$$

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 5.1 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 5.1) is computed as follow:

$$Block_{SM} = min(Blocks_w, Blocks_r) \tag{5.3}$$

$Block_{SM}$ 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 5.2).

$$W_{block} = \max_n \{n | R_{block} * Blocks_{SM} \leq MaxReg_{SM}\} \tag{5.4}$$

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.
   $obsToProcess = \left\lceil \frac{nTime}{nSplit} \right\rceil$
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.
   $nodePerDevice = \left\lceil \frac{nNodes}{nGPUs} \right\rceil$
3. Compute the dimensions of the block in functions of the number of warps per block computed before. From Equation 5.4 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:
   - $gridDim.x = \left\lceil \frac{nNodes}{blockDim.x} \right\rceil$
   - $gridDim.y = \left\lceil \frac{nodePerDevice}{blockDim.y} \right\rceil$
   - $gridDim.z = obsToProcess$

All the formulation presented above results in the dimension of the GPU structure created to support a generic GPU function. An important remark to make is that each thread in a grid executes the same function, the computation showed above are applied to any GPU function defined. In particular we defined one GPU function for the hybrid version and one GPU function for the full hardware version. Obviously, the two functions present different resource requirements, in fact the shape of the GPU structure results different. The common denominator among the two implementations is that each thread compute the similarity for one give node pair. The two nodes composing the pair are given by the position of the thread in the structure, the *x* and *y* position set the node number of the pair associated to the thread, computed using the formulas shown in Equation 2.3.

5.2 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. The structure of the data loaded is shown in Figure 5.5, where $nTimes$ represents the number of observation instants and $nNodes$ is the nodes number of the input dataset.

5.3 Results Saving

The program stores the correlation matrices at the end of the computation of each one. The saving process creates one text file for each matrix. 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

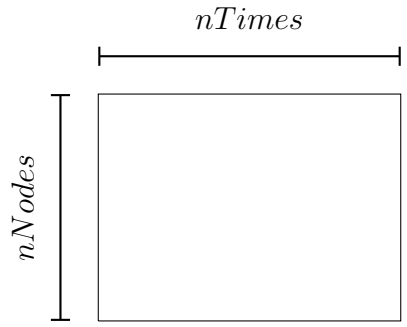


Figure 5.5: Data format loaded on each GPU containing the program input data.

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 functions, 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, this avoid the overload of the BUS.

CHAPTER 6

Performance Evaluation

This chapter is dedicated to the tests performed to evaluate the performance of the GPU program. The development of the GPU program has been performed with 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.

We now show the results obtained running the different versions presented before (hybrid and full) on the devices compared with the same computation execution on the CPU. We want to show how the hardware acceleration is a valid solution for massive parallel computation. In details the comparison in term of execution time occurs among different run, specifically we decided to run different tests: one using only one GPU, one using three GPUs in parallel (all the same model) and one using a general CPU. We are going o compare the execution time among these looking for pros and cons of every execution. The system where all the tests were run has installed Window 10 Enterprise Edition 64 bit as OS and presents 12 GB of RAM Dual-Channel DDR3 @ 668 MHz (3×4 GB). The local storage used to save the results is a Samsung SSD 840 EVO 120 GB. Furthermore, the devices used in the tests performed are:

- **CPU**: Intel i7 960 @ 3.20 GHz
- **GPU**: NVida GTX 770 4 GB @ 667 MHz

[1] https://www.visualstudio.com/en-us/visual-studio-homepage-vs.aspx
[2] http://www.nvidia.com/object/nsight.html

Table 6.1: Execution time of the runs performed on the *BioData* dataset. **Full** column contains the execution time of the full HW version of the program implemented. **Hybrid HW** column, instead, contains the execution time of the GPU function developed for the hybrid version of the program. **Hybrid HW + CPU** column contains the time needed by the hybrid version (CPU + HW) to get the similarity computed.

Device	**Full** [s]	**Hybrid HW** [s]	**Hybrid HW + CPU** [s]
Multi GPU	2.645	1.237	1.349
Single GPU	6.108	2.841	2.95
CPU	140.0	140.0	140.0

All the tests performed are run on the whole dataset, so all the similarity matrices are computed and saved. In particular, *BioData* contains 250 different images and *HumData* contains 1000 images, so will be the number of matrices saved by the program fro the two datasets. However, to make a direct comparison we focus on the computation time of one image of the dataset. In fact all the time shown in the results tables refer to the computation of only one similarity matrix.

Table 6.1 shows the execution time for the tests run on the *BioData* dataset, extracted from the rat lab data. This present a great number of nodes to take into account, so the execution results heavy to perform, due to the high amount of data to process. The data, as we expected, highlight the advantage of the usage of HW devices for this kind of computation, in particular the speed up obtained is: $\sim 53\times$ on multi GPU and $\sim 23\times$ on single GPU for the full HW version, while $\sim 113\times$ on multi GPU and $\sim 49\times$ on single GPU for the hybrid version. The execution time of the hybrid hardware function is lower since the function itself is smaller than the full hardware function implemented, to draw a conclusion about which is better we have to consider with the hybrid execution time the time needed by the CPU to compute the part executed on it. Comparing the **Full** column and the **Hybrid HW + CPU** column we see that the hybrid version is still convenient in term of execution time.

Table 6.2 shows the tests execution time performed on the other dataset available, *HumData*, in this case the amount of data to process is lower with the respect of the other dataset used, in fact, in this case the managing of the resources had been easier. Even in this case the execution time highlight the convenience of GPU acceleration with the respect of the CPU execution. In particular we obtained a speed up of $\sim 41\times$ on multi GPU and $\sim 23\times$ on single GPU for the full HW version, and a speed up of $\sim 106\times$ on multi GPU and $\sim 48\times$ on single GPU for the hybrid version. The same of *BioData* goes for *HumData*, the hybrid version considered with the CPU time is still convinient in term of execution time.

As already anticipated, both the results presented in Table 6.1 and in Table 6.2

Table 6.2: Execution time of the runs performed on the *HumData* dataset.**Full** column contains the execution time of the full HW version of the program implemented. **Hybrid HW** column, instead, contains the execution time of the GPU function developed for the hybrid version of the program. **Hybrid HW + CPU** column contains the time needed by the hybrid version (CPU + HW) to get the similarity computed.

Device	**Full** [s]	**Hybrid HW** [s]	**Hybrid HW + CPU** [s]
Multi GPU	0.267	0.103	0.123
Single GPU	0.463	0.23	0.249
CPU	11.0	11.0	11.0

show the computation time for only **one** frame or time instant of the dataset. However, the computation of the full dataset is composed by multiple calls of the same function, so the execution time of the program for the whole dataset consist in the multiplication of the time needed to compute one time instant by the number of the time instants in the data collected. But in this work we are interested in showing which one among the solution implemented is the most efficient in term of execution time, so we look exclusively at the velocity of the algorithm based on a single element.

One last important consideration to make is that in the previous execution time data there is no included the time needed to save the similarity matrices on the hard disk. This phase is really important since the data computed are needed for the visualization presented in Chapter 7 or they can be used in further analysis to study the neurons interactions. However, this phase is the most expensive, due to the high number of nodes that compose the similarity matrix and to the high number of time stamp to save. The time needed to save one single similarity matrix are: $1031s$ ($17m$ and $11s$) for the *BioData* dataset and $74s$ ($1m$ and $14s$) for the *HumData* dataset. The time needed to save the similarity on file, which is much greater than the time needed to compute the similarity, represent the bottleneck of the pipeline presented in this thesis work, moreover if we think that to perform analysis on the results we need all the similarity matrices so we have to save them for each time instant of the dataset the final execution time of the whole program grow significantly.

CHAPTER 7

Networks Visualization Tool

In this chapter we present the outcome of the visualization tool development, this tool aims to visualize the dynamic networks computed with the program presented in Chapter 5, this tool represents **Step 3** of the pipeline shown in Figure 4.3 of Chapter 4.

7.1 Visualization Tool Prototype

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. The tool we present is a prototype where much work has to be done yet enriching it with new more complex features. However, we want to show the usefulness of this kind of visualization in providing help to domain experts. You can find the visualization tool here[2].

7.1.1 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

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

[2] `http://purgato.evl.uic.edu:3000/`

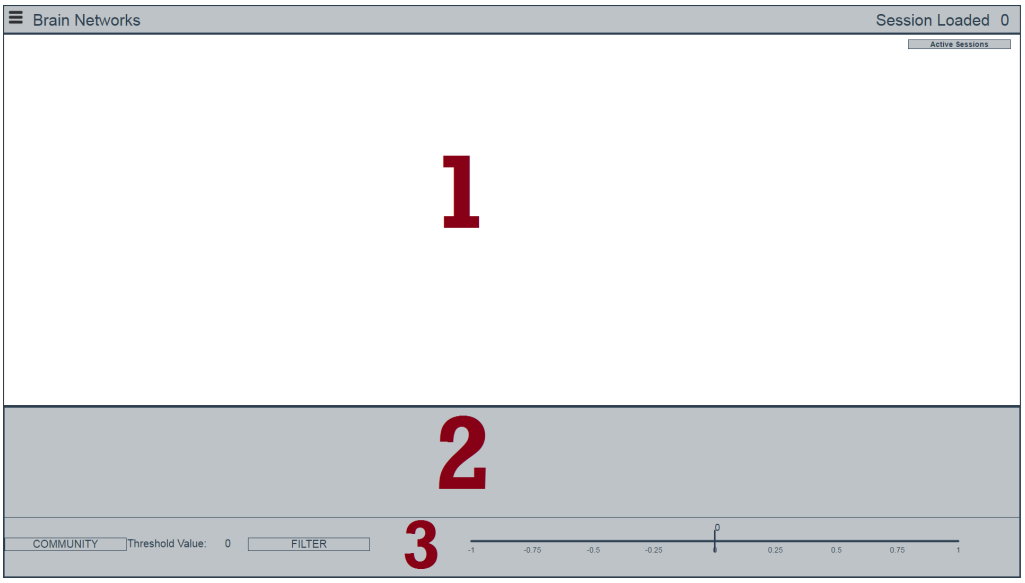


Figure 7.1: Tool interface, main page of the interactive visualization tool developed to display dynamic brain networks.

visualization world is D3, it 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[1], 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.

7.1.2 Visualization Organization

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 7.1 shows the main page of the web application. There is a header with all the general information: a menu button and an info box containing all

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

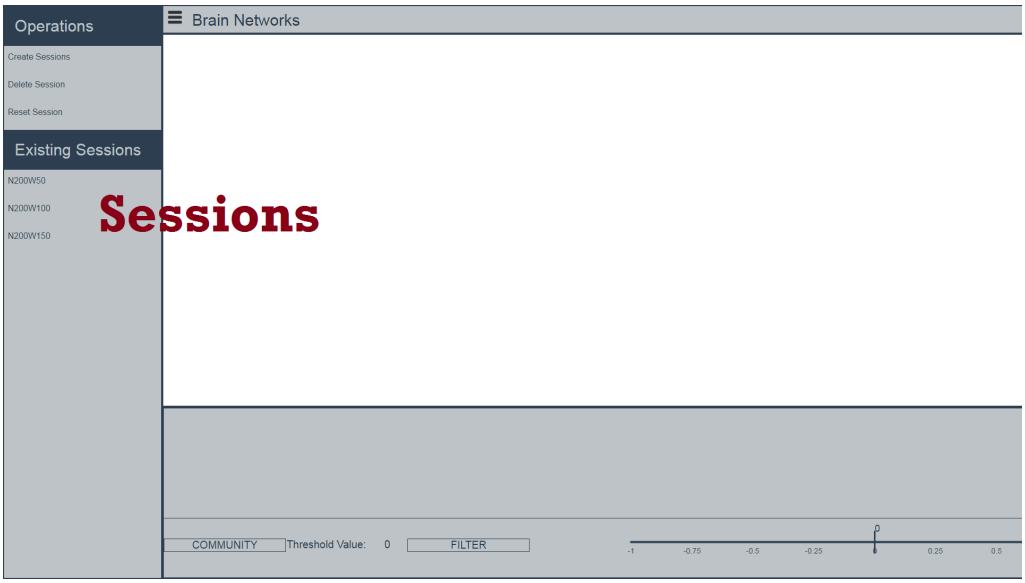


Figure 7.2: Tool menu, used to manage the data sessions. It allows the creation, deletion and load/unload of the data in the system.

the information related to the data loaded in the system. Each main section of Figure 7.1 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 instant 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 used to define the DN. The threshold can be changed at any time, and the filtering is applied to any data version loaded in the system.

7.1.3 Session Manager

This tool keeps track of different versions of the data that are computed using the algorithms presented in Chapter 5. 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 7.2 shows how he system manage the sessions. It is possible create, delete and reset the sessions in the system. 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 reset function, instead, consists in the unload of all the data previously loaded in the system. The sessions loaded are the ones selected by the user to be visualized. The loading operation, in fact, is different from the creation, while the first actually load and prepare the data for the visualization the creation is used to add a new version of the DNs in the system database. To load a new session is necessary to click on the name of the session, previously created, shown in the menu.

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, 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 7.1) 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 are loaded in the memory. In this way the filtering process will be faster when the user changes the threshold.

7.1.4 Edge Definition and Network Visualization

The edge definition is performed only after the setting of the similarity threshold. They are defined thanks to a filtering process that keeps the nodes pairs with similarity greater than the threshold. Figure 7.3 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 of the observation period. 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 7.4 shows an example of two networks visualized with two different correlation thresholds.

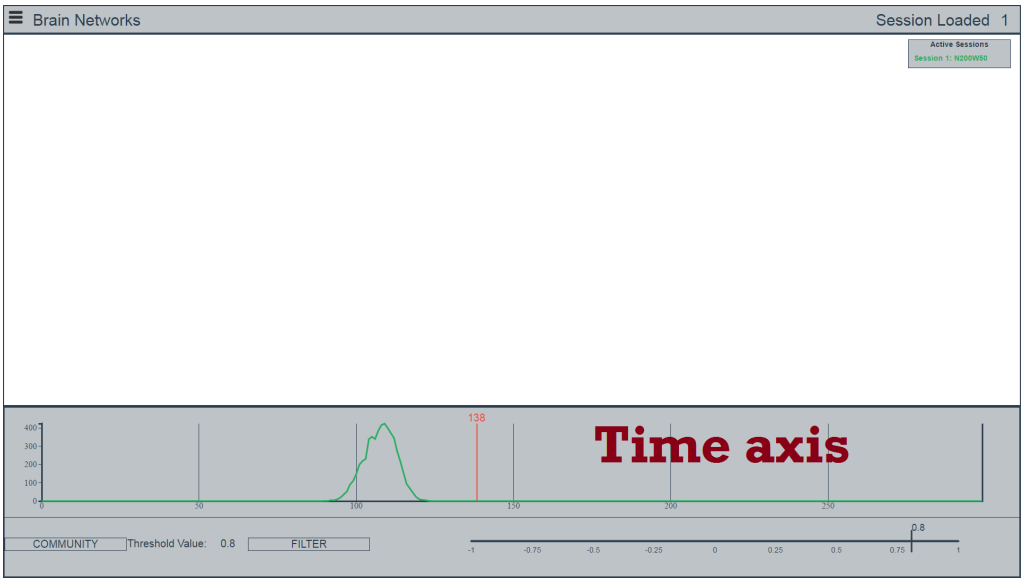


Figure 7.3: Data filtering, process that take the data loaded in the system and filter them using the threshold set by the user. It shows the time axis with the number of sedges for each time instant.

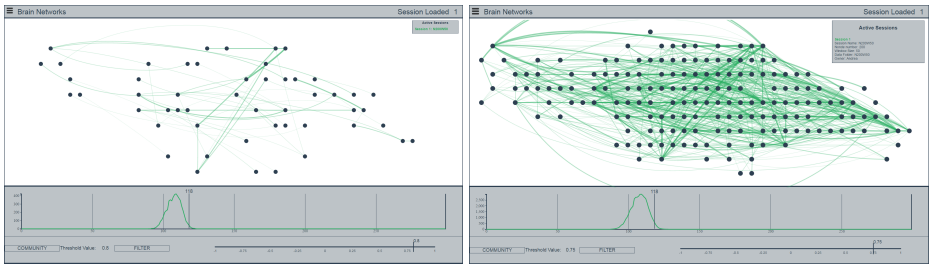


Figure 7.4: Different threshold networks, examples of network displayed with different correlation threshold,the left image has a greater threshold so less edges are present in the network, while the right image has a lower threshold and more edges are created for the network.

7.1.5 Network Exploration

Figure 7.5 shows an example of network visualization for one data session. 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 (see Figure 7.6) represents the portion of the brain visualized, this means that out of that rectangle there are no edges. The scale of

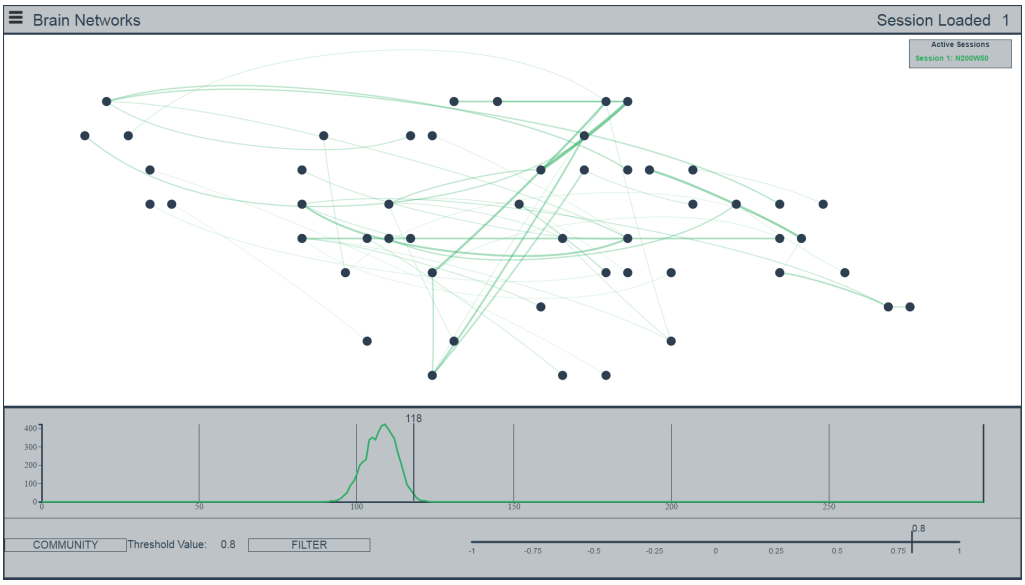


Figure 7.5: Network visualization, rendering of the network created by the filtering process, it is showed the network associated to the time instant selected in the time axis.

the network is performed autonomously by the system, moreover the network could be moved and zoomed by the user with the mouse pointer. The stroke and the opacity of the edges are 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 chaotic. 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 drawing of the network directly above the brain picture since it can results confusional and difficult to understand. Figure 7.6 shows how we implemented the map between 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. This shows the exact position of what selected in the brain. Moreover some important information are shown when the user goes over a node, like connections and correlation relative to the links, besides the nodes position in the brain.

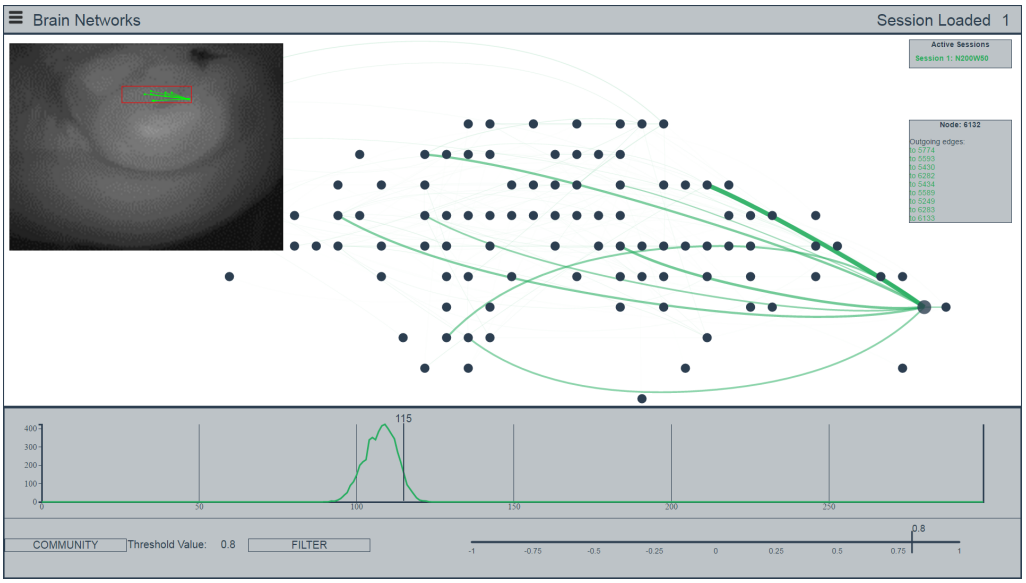


Figure 7.6: Network mapping, it shows the solution we adopted to map the network drawn on the brain. This mapping allows the identification of the absolute position of nodes in the brain.

7.1.6 Network Comparison

One of the main characteristics that our system has is the comparison between different version of networks. 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 7.7 shows how the visualization tool appears when more than one session is loaded, the networks are represented by different colors so that the users can see the differences for the time instant selected on the timeline.

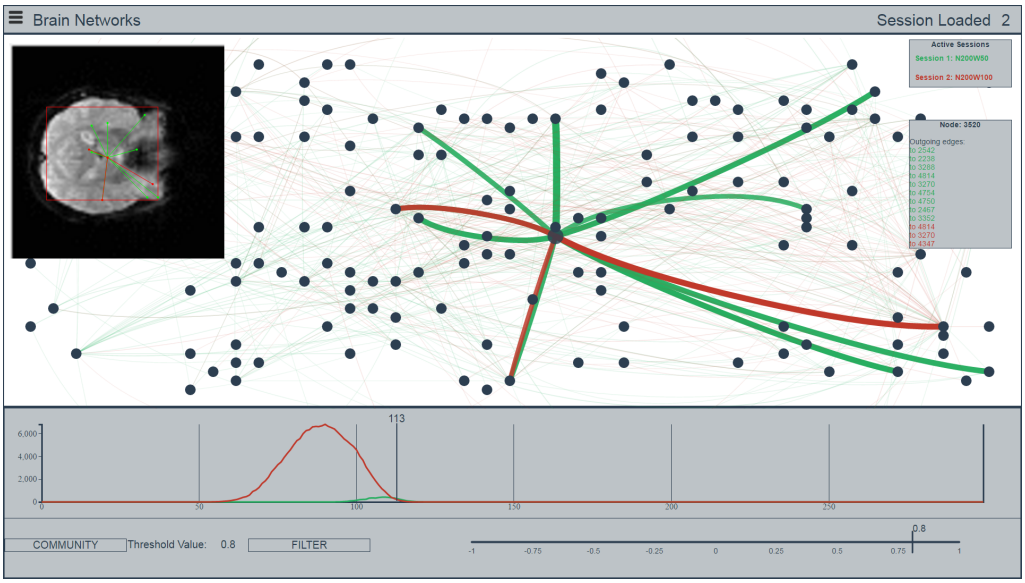


Figure 7.7: Network comparison, it shows the comparison between two different data session loaded in the system, each one is represented by a color in the network.

CHAPTER 8

Conclusions

We presented an interdisciplinary work able to compute DNs and visualize them in a clear and easy way to make the results understandable. We focused our interests especially on the development of different solutions adoptable for DNs definition, evaluating then the execution results obtained running different tests on different dataset. During the development process we encountered some bottleneck that slowed what we expected at the beginning, in fact the high computational power of the GPUs adopted is limited by the slow writing speed of the storage system 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) is really low which is what we were expecting.

8.1 Future Works

Taking advantage of the low computation time achieved 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 every step of the pipeline is executed on the same context, so the results computed by the program are already resident in the central memory and there is no need to save them on local storage. However, the mole of data computed is really high and sometimes the global memory of the machine 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. In the programs developed we avoid the filter of the data for one reason, give a clear idea of all the results computed, but if the GPU computation results are used on-line by other functions that analyses them it is possible keep in memory only the useful data necessary for the specific operation.

One other possible further work is the implementation of different similarities measures in the program so that the user can chose which one compute exploiting the hardware computational power. We have seen that the computation on hardware devices 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 defined, 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 results computed. A much more complex extension would be, instead, the generalization of the visualization from the context of DNs application, making possible the visualization of DNs applied to human being or animals in the same way we visualized the brain networks. This aspect would made the visualization tool really powerful but the realization of it is really complex since we have to find a way to bring all the data under the same format and under the same spatial domain space.

Acronyms

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

Bibliography

[1] Parallel programming and computing platform | cuda | nvidia. http://www.nvidia.com/object/cuda_home_new.html. Accessed: 03-02-2016.

[2] Rajagopal Ananthanarayanan and Dharmendra S Modha. Anatomy of a cortical simulator. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 3. ACM, 2007.

[3] Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. A taxonomy and survey of dynamic graph visualization. In *Computer Graphics Forum*, 2016.

[4] Tanya Berger-Wolf and David Kempe. 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.

[5] TY Berger-Wolf, C Tantipathananandh, and D Kempe. Community identification in dynamic social networks. *Link Mining: Models, Algorithms and Applications*, 2010.

[6] Ian Buck. Gpu computing: Programming a massively parallel processor. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 17–17. IEEE, 2007.

[7] Riccardo Cattaneo, Giuseppe Natale, Carlo Sicignano, Donatella Sciuto, and Marco Domenico Santambrogio. On how to accelerate iterative stencil loops: A scalable streaming-based approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):53, 2015.

[8] Dar-Jen Chang, Ahmed H Desoky, Ming Ouyang, and Eric C Rouchka. 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.

[9] Shi Chen, Amiyaal Ilany, Brad J White, Michael W Sanderson, and Cristina Lanzas. Spatial-temporal dynamics of high-resolution animal networks: What can we learn from domestic animals? *PloS one*, 10(6): e0129253, 2015.

[10] Petter Holme. Modern temporal network theory: a colloquium. *The European Physical Journal B*, 88(9):1–30, 2015.

[11] Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519 (3):97–125, 2012.

[12] Seongho Kim, Ming Ouyang, and Xiang Zhang. Compute spearman correlation coefficient with matlab/cuda. In *Signal Processing and Information Technology (ISSPIT), 2012 IEEE International Symposium on*, pages 000055–000060. IEEE, 2012.

[13] Daren Lee, Ivo Dinov, Bin Dong, Boris Gutman, Igor Yanovsky, and Arthur W Toga. Cuda optimization strategies for compute-and memory-bound neuroimaging algorithms. *Computer methods and programs in biomedicine*, 106(3):175–187, 2012.

[14] Daniel Llano, Chihua Ma, Kevin Stebbings, Umberto Di Fabrizio, Robert V Kenyon, and Tanya Berger-Wolf. A novel dynamic network analysis reveals aging-related fragmentation of cortical networks in mouse. Unpublished manuscript, 2016.

[15] Chihua Ma, Angus G Forbes, Daniel A Llano, Tanya Berger-Wolf, and Robert V Kenyon. Swordplots: Exploring neuron behavior within dynamic communities of brain networks. *Journal of Imaging Science and Technology*, 2015.

[16] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5): 879–899, 2008.

[17] Hans E Plesser, Jochen M Eppler, Abigail Morrison, Markus Diesmann, and Marc-Oliver Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In *Euro-Par 2007 parallel processing*, pages 672–681. Springer, 2007.

[18] Teresa M Przytycka, Mona Singh, and Donna K Slonim. Toward the dynamic interactome: it's about time. *Briefings in bioinformatics*, page bbp057, 2010.

[19] Daniel I Rubenstein, Siva R Sundaresan, Ilya R Fischhoff, Chayant Tantipathananandh, and Tanya Y Berger-Wolf. Similar but different: Dynamic social network analysis highlights fundamental differences between the fission-fusion societies of two equid species, the onager and grevy's zebra. *PloS one*, 10(10):e0138645, 2015.

[20] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[21] Vedran Sekara, Arkadiusz Stopczynski, and Sune Lehmann. The fundamental structures of dynamic social networks. *arXiv preprint arXiv:1506.04704*, 2015.

[22] Lei Shi, Chen Wang, and Zhen Wen. Dynamic network visualization in 1.5 d. In *Visualization Symposium (PacificVis), 2011 IEEE Pacific*, pages 179–186. IEEE, 2011.

[23] Stephen M Smith, Karla L Miller, Gholamreza Salimi-Khorshidi, Matthew Webster, Christian F Beckmann, Thomas E Nichols, Joseph D Ramsey, and Mark W Woolrich. Network modelling methods for fmri. *Neuroimage*, 54 (2):875–891, 2011.

[24] Damon JA Toth, Molly Leecaster, Warren BP Pettey, Adi V Gundlapalli, Hongjiang Gao, Jeanette J Rainey, Amra Uzicanin, and Matthew H Samore. 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.

[25] Eugenio Valdano, Chiara Poletto, Armando Giovannini, Diana Palma, Lara Savini, and Vittoria Colizza. Predicting epidemic risk from past temporal contact data. *PLoS Comput Biol*, 11(3):e1004152, 2015.

[26] Stef van den Elzen, Danny Holten, Jorik Blaas, and Jarke J van Wijk. 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] Yu Wang, Haixiao Du, Mingrui Xia, Ling Ren, Mo Xu, Teng Xie, Gaolang Gong, Ningyi Xu, Huazhong Yang, and Yong He. A hybrid cpu-gpu accelerated framework for fast mapping of high-resolution human brain connectome. *PloS one*, 8(5):e62789, 2013.

[28] W Hwu Wen-Mei. *GPU Computing Gems Emerald Edition*. Elsevier, 2011.