

Article

Parallel Simulations of the Sharp Wave-Ripples of the Hippocampus on Multicore CPUs and GPUs

Emanuele Torti , Simone Migliazza, Elisa Marenzi , Giovanni Danese  and Francesco Leporati *

Department of Electrical, Computer and Biomedical Engineering, University of Pavia, 27100 Pavia, Italy; emanuele.torti@unipv.it (E.T.); simone.migliazza01@universitadipavia.it (S.M.); elisa.marenzi@unipv.it (E.M.); giovanni.danese@unipv.it (G.D.)

* Correspondence: francesco.leporati@unipv.it

Featured Application: The simulation of the hippocampus CA1 area on parallel desktop-based devices.

Abstract: The simulation of realistic systems plays a crucial role in modern sciences. Complex organs such as the brain can be described by mathematical models to reproduce biological behaviors. In the brain, the hippocampus is a critical region for memory and learning. In the literature, a model to reproduce the memory consolidation mechanism has been proposed. This model exhibits a high degree of biological realism, though it is accompanied by a significant increase in computational complexity. This paper proposes the development of parallel simulation targeting different devices, namely multicore CPUs and GPUs. The experiments highlighted that the biological realism is maintained, together with a significant decrease of the processing times. Finally, the conducted analysis highlights that the GPU is one of the most suitable technologies for this kind of simulation.

Keywords: parallel processing; GPUs; multicore CPUs; simulation of complex systems



Citation: Torti, E.; Migliazza, S.; Marenzi, E.; Danese, G.; Leporati, F. Parallel Simulations of the Sharp Wave-Ripples of the Hippocampus on Multicore CPUs and GPUs. *Appl. Sci.* **2024**, *14*, 9967. <https://doi.org/10.3390/app14219967>

Academic Editors: Josie Esteban Rodríguez Condia, José Rodrigo Azambuja, Vicent Botti and Arcangelo Castiglione

Received: 6 September 2024

Revised: 24 October 2024

Accepted: 25 October 2024

Published: 31 October 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The simulation of complex biological systems plays a pivotal role in research, as it can substitute for costly, prolonged, and intricate experiments. This markedly enhances efficiency and quantification in research, facilitating the investigation of the impact on biological system operations under artificially imposed conditions. Of particular interest is the study and simulation of the human brain, given its inherent complexity and significance. Within this domain, the hippocampus, a critical region for memory and learning, represents a focal point of research, where simulations can provide invaluable insights into its functional dynamics and responses. The process of memory consolidation occurs during sleep, facilitated by a bidirectional exchange of information between the hippocampus and the cortex [1,2]. This intricate communication enables the transfer and integration of newly acquired memories, thereby solidifying them for long-term storage and retrieval. During sleep, neurons in the hippocampus and cortex that were active during daytime behavioural tasks are reactivated. This process is of great consequence in linking and modifying memories, transferring them from short-term storage in the hippocampus to long-term storage in the cortex. A pivotal element of this process is the occurrence of brief electrical events in the hippocampus, designated as sharp wave-ripples, which are high-frequency oscillations that facilitate the replay of neural activity patterns, effectively coordinating the communication between the hippocampus and cortex to consolidate and stabilise memories for long-term retention [3,4]. A recent work [5] described a computational model capable of reproducing the ripple generation mechanism consistent with a wide range of experimental data. It considers a network composed of two different types of neurons, namely excitatory and inhibitory cells. The release of the neurotransmitter glutamate by excitatory neurons increases the likelihood of other neurons firing, thereby stimulating brain activity and propagating signals. In contrast, the release of Gamma-aminobutyric

acid (GABA) by inhibitory neurons decreases the likelihood of neuron firing, regulating and balancing neural activity to prevent excessive excitation. In the hippocampal cornu ammonis region 1 (CA1), the ratio of excitatory to inhibitory cells is approximately 5:1 [6], which is maintained in the simulated network where 800 excitatory and 160 inhibitory neurons are modeled [5]. Nevertheless, the computational complexity of such models increases in proportion to the number of neurons involved. The typical approach when facing problems where computational power is needed is the adoption of high-performance computing methods, as has already been conducted in other scientific fields.

Thus, to address this issue, several methods have been developed with the objective of accelerating the simulation of biologically realistic neuronal networks [7]. These methods primarily leverage high-performance computing technologies, including supercomputers [8], multicore processors [9], Graphical Processing Units (GPUs) [10,11], Field Programmable Gate Arrays (FPGAs) [12], and custom processors [13]. However, when focusing on the simulation of the hippocampus, only some of these technologies have been adopted.

A study [14] on the Parallel Brain Simulator (PBS) demonstrates its capacity to simulate expansive hippocampal networks through the utilisation of high-performance computing, encompassing clusters and supercomputers. The model demonstrates effective scalability and supports diverse synaptic types. However, the primary limitations include considerable computational complexity and the challenges in balancing accuracy with performance, particularly as the network size increases. Furthermore, the platform's dependency on specialised hardware may restrict its accessibility for some researchers.

Another interesting study [15] presents a model network comprising simple integrate-and-fire neurons with Tsodyks–Markram synapses, devised to assess the efficacy of simulation. The hippocampal simulation involved over 40 billion synaptic connections and required substantial computational resources, including 2400 processors and 19.2 TB of memory, on the Piz Daint Cray XC40 supercomputer. Again, the primary limitations include high computational complexity, memory constraints, and the necessity for specialised hardware, which may restrict accessibility and scalability for broader research applications.

Finally, the work in [16] presents a discussion of the development and testing of hippocampal memory prostheses, with a particular focus on the utilisation of input-output models (multi-input multi-output and memory decoding models) for the simulation of hippocampal functions. The models simulate spike trains in the Cornu Ammonis region 3 (CA3) (input) and CA1 (output) regions of the hippocampus. Data were collected from neurons of 11 human subjects. Simulations were accelerated using parallel computing on a high-performance computing cluster at the University of Southern California. However, the neurons are modelled adopting an approach that does not consider the biological behaviours of the cells.

The lack of parallel and biologically realistic hippocampus models that can be simulated on desktop PCs equipped with multicore CPUs and GPUs is a notable shortcoming in the field of computational neuroscience. It is of great importance to be able to simulate the hippocampus on accessible hardware for several reasons. First, desktop PCs with multicore CPUs and GPUs are more affordable and more widely available than supercomputers, thereby facilitating greater accessibility to advanced research for a broader range of scientists and institutions. Second, these hardware platforms are capable of extensive parallel processing, which is well aligned with the complex, interconnected nature of hippocampal neural networks. In contrast to supercomputers, which may be overly sophisticated for this particular application, desktop systems can be optimized to achieve balance between performance and cost-effectiveness. Moreover, the utilisation of CPUs and GPUs allows researchers to undertake rapid iterations of their models and incorporate real-time adjustments based on experimental findings, thereby fostering a more agile research process. Last, the computational demands of simulating hippocampal dynamics often align well with the capabilities of modern GPUs, which are specifically designed to handle large-scale parallel computations efficiently. By leveraging these more accessible platforms, researchers can

also develop scalable models that can be more easily adapted and optimised for various applications, from understanding neurological disorders to developing new AI algorithms inspired by brain functions.

This paper proposes the development of parallel simulations for the model described in [5], with a particular focus on multicore CPUs and GPUs. The parallelised code has been subjected to rigorous validation against the reference implementation provided by the authors of [5], with the objective of ensuring accuracy and reliability. To evaluate the performance, experimental simulations were conducted using identical network configuration to those employed in [5], successfully reproducing two seconds of cellular activity across a range of CPU and GPU models. Furthermore, scalability tests were conducted using both smaller and larger network configurations than those originally employed, thus enabling a comprehensive evaluation of the model's scalability with varying network sizes. The results demonstrated that the GPU implementation consistently exhibited superior performance compared to the other versions of the code, achieving simulation times of less than two seconds even for the largest network configurations tested. These findings highlight the efficacy of GPU-based parallel processing in addressing intricate neural simulations, indicating that GPUs confer a substantial advantage over CPUs in terms of processing speed for large-scale models.

The paper is organized as follows. Section 2 describes the material and methods, detailing the mathematical model of the CA1 network and its parallelization targeting multicore CPUs and GPUs. Section 3 describes the experimental setup and the obtained results. Section 4 contains the discussion, together with the comparison with the state of the art. Finally, Section 5 concludes the paper, giving final remarks and describing possible future research lines.

2. Materials and Methods

This section describes the mathematical model of the CA1 network, together with the parallelization methods employed on the different hardware technologies.

2.1. Mathematical Model

The presented work stems from [5], where data had been recorded from 6–7 months old Brown Norway rats during natural sleep. The developed computational model comprises a limited portion of the CA1 pyramidal layer, which is subjected to strong excitatory inputs from the CA3 region. Here, the aim is to reproduce the activity of a group of neurons that, in vivo, could be evaluated using a single electrode, thus assuring optimal comparability between the experimental observations and the model. Two neuronal types have been considered, pyramidal and parvalbumin positive basket cells, since ripples are measured only in the pyramidal layer. More specifically, a 5:1 ratio between numerosity of neuron types is maintained, consistently with real anatomy; in particular, 800 pyramidal cells and 160 interneurons. All-to-all connectivity has been implemented, while synapses connecting the pyramidal cells to one another are weaker and in reduced number. Figure 1 shows cell types and network schematic with connections.

Local Field Potential (LFP) data have been considered and processed using MATLAB R2020a (MathWorks, Natick, MA, USA): the data underwent filtering to better observe sharp wave ripple (SWR) complexes and consequently calculate ripple duration as the duration of SWR. In addition, ripple frequency has been derived as average inter-peak interval of filtered LFP, while ripple inter-arrival times are computed as the time interval between the onset of two consecutive ripples. The CA1 hippocampus computational model considered in our work as the starting point for our optimization is composed of 160 interneurons and 800 pyramidal cells. The equations adopted to model the functional behaviour of every neuron are as follows, and all the parameters related to both neuron types have been kept the same as in [5]. Their values are shown in Table 1.

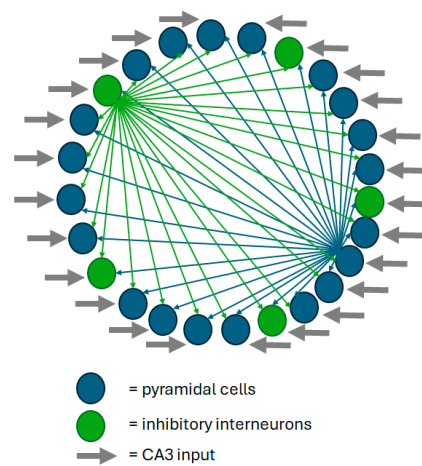


Figure 1. circles represent single cells, where the blue ones are the pyramidal cells while the green are the interneurons. CA3 input is shown as the external grey arrows, whereas arrows of the same color of cell types reproduce an illustrative example of type and numerosity of connections between neurons.

$$C\dot{v} = -g_L(v - E_L) + g_L\Delta e^{\frac{v-V_t}{\Delta}} - w + I(t) \quad (1)$$

$$\tau_w\dot{w} = a(v - E_L) - w \quad (2)$$

$$v(t) = V_{thr} \Rightarrow v(t + dt) = V_r, w(t + dt) = w(t) + b \quad (3)$$

Table 1. Model parameters and their values.

Parameter	Description	Pyramidal Cells	Inhibitory Interneurons
C	Membrane capacitance	200 pF	200 pF
g_L	Leakage conductance	10 nS	10 nS
E_L	Reversal potential	−58 mV	−70 mV
a	Coupling parameter	2	2
b	Coupling parameter	100 pA	10 pA
Δ	Sharpness of action potential initiation	2 mV	2 mV
τ_w	Membrane time constant	120 ms	30 ms
V_t	Intrinsic threshold	−50 mV	−50 mV
V_r	Resting potential	−46 mV	−58 mV
V_{thr}	Threshold	0 mV	-

In addition, cells are receiving different inputs, but always maintaining the same structure. In the following equations describing the inputs, I_{DC} is kept constant for each cell, but will assume different values from a normal distribution a priori defined, depending on the single neuron, while η_t is the stochastic Ornstein–Uhlenbeck (OU) process. β coefficients differ among cell types, namely $\beta = 80$ in the case of pyramidal cells and $\beta = 90$ for interneurons:

$$I(t) = I_{DC} + \beta\eta_t + I_{syn}(t) + I_{inp}(t) \quad (4)$$

$$\tau d\eta_t = -\eta_t dt + dW_t \quad (5)$$

$$I_{inp}(t) = I_{max} \left(1 + e^{-\frac{t-t_{on}}{k}} \right)^{-1} \left(1 + e^{-\frac{t-t_{off}}{k}} \right)^{-1} \quad (6)$$

Considering the generic cell with index n , two equations define synaptic currents, where F is the normalization factor and H is the Heaviside function. Each value of g is

derived from a Gaussian distribution, defined as having a 10% standard deviation of its mean, and values below 0 are rectified to 0. Lastly, $I_{inp}(t)$ is the net effect of CA3 synaptic excitation on CA1 cells, and t_{on} and t_{off} are mostly 50 ms apart:

$$I_{syn}(t) = \sum_{j=1}^{160} g^{j \rightarrow n} s^{j \rightarrow n}(t)(v_n - E_i) + \sum_{j=1}^{800} g^{j \rightarrow n} s^{j \rightarrow n}(t)(v_n - E_e) \quad (7)$$

$$s^{j \rightarrow n}(t) = \sum_{spikes \text{ of cell } j} F\left(e^{H(-\frac{t-t_k}{\tau_D})} - e^{H(-\frac{t-t_k}{\tau_R})}\right) \quad (8)$$

In such equations, the parameters have these values:

- $\bar{g}_{Int \rightarrow Int} = 0.0234$ nS
- $\bar{g}_{Pyr \rightarrow Int} = 0.0083$ nS
- $\bar{g}_{Int \rightarrow Pyr} = 0.0521$ nS
- $\bar{g}_{Pyr \rightarrow Pyr} = 0.001$ nS
- reversal potential for synaptic currents: $E_i = -80$ mV and $E_e = 0$ mV
- time scales of synaptic rise and decay, respectively:
 - excitatory synapses:
 - pyramidal neurons: $\tau_R = 0.5$ ms and $\tau_D = 3.5$ ms
 - interneurons: $\tau_R = 0.9$ ms and $\tau_D = 3$ ms
 - inhibitory synapses:
 - pyramidal neurons: $\tau_R = 0.3$ ms and $\tau_D = 3.5$ ms
 - interneurons: $\tau_R = 0.3$ ms and $\tau_D = 2$ ms
- $I_{max} = 210$ pA for pyramidal cells
- $I_{max} = 700$ pA in the case of interneurons
- simulations performed with a 1-step Euler algorithm ($\Delta t = 0.001$ ms)

2.2. Parallel Simulations of the CA1 Network

This section details the different parallel implementations targeting multicore CPUs and GPUs. Prior to the description of these parallel algorithms, the serial code is introduced since it serves both as a base for the multicore and GPU versions and as baseline to evaluate the performance.

2.2.1. The Serial Algorithm

The simulation of the hippocampus neural network starts by creating the connections between the different neurons under the constraints described in Section 2.1. Indeed, the connections and the synaptic weights are selected from a Gaussian distribution with zero mean and a variance of 10%. Then, all the non-time dependent values are computed together with the background noise. The inputs from the CA3 section in form of voltages and currents are also computed. Then, the equations described in Section 2.1 are solved with a 1-step Euler algorithm with an integration step of 0.001 ms. Thus, serial code considers the simulation of 2 s of cellular activity with a time step of 0.001 ms. At this point the simulation of the cellular activity can start. A main *for* loop iterates over these time steps. The first operations are the update of the synaptic currents, which can be written as a set of four matrix-vector multiplications, one for each of the four connection types, namely excitatory-excitatory (EE), excitatory-inhibitory (EI), inhibitory-inhibitory (II) and inhibitory-excitatory (IE). The matrices store the terms related to the synaptic conductance and the normalization factor of the spikes, while the vector contains the difference between the actual potential of a neuron and its inversion potential. The next step is the update of the neuronal voltages. It is performed by two *for* loops, one for each kind of cell. The operations performed inside these two loops are described by Equations (1)–(8). Figure 2 summarizes the main steps of the serial algorithm.

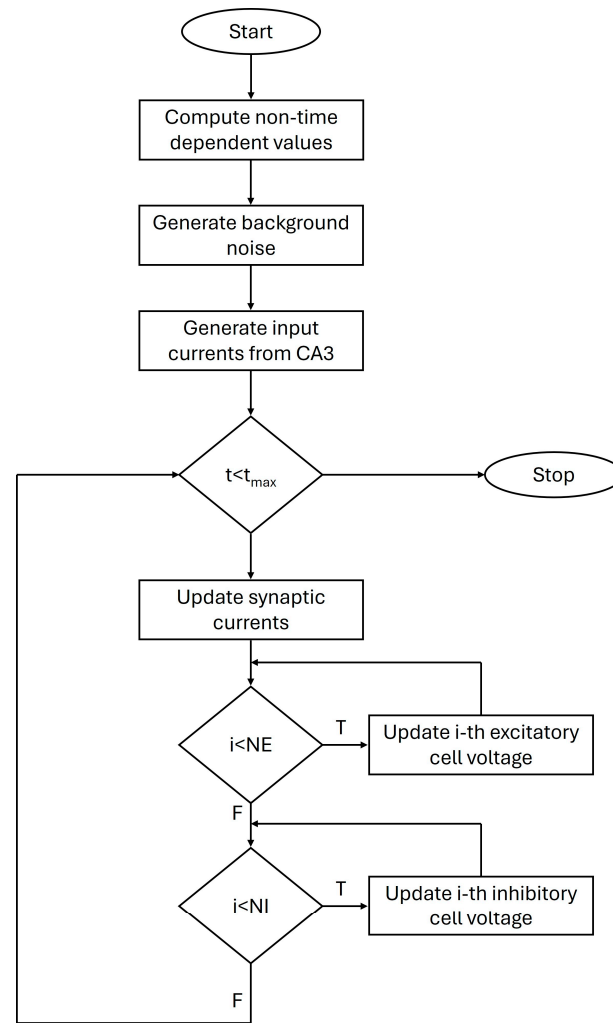


Figure 2. the serial algorithm flow. The terms *NE* and *NI* represent the number of excitatory and inhibitory neurons, respectively.

The operations performed inside the main *for* loop are detailed in Algorithm 1.

Algorithm 1: Pseudocode of the main for loop

Input: Initial values of conductance and synaptic weights for each connection

```

1: for  $t = 0$  to  $t_{max}$ 
2:   #update the synaptic conductances with matrix-vector multiplications
3:    $g_{EE} = \overline{g_{EE}} * w_{EE}$ 
4:    $g_{EI} = \overline{g_{EI}} * w_{EI}$ 
5:    $g_{IE} = \overline{g_{IE}} * w_{IE}$ 
6:    $g_{II} = \overline{g_{II}} * w_{II}$ 
7:   Compute Equations (7) and (8)
8:   for  $i = 0$  to  $NE$ 
9:     compute Equations (1)–(3) to update the excitatory cells voltage
10:  end
11:  for  $i = 0$  to  $NI$ 
12:    compute Equations (1)–(3) to update the inhibitory cells voltage
13:  end
14: end
  
```

The main for loop starts in line 1 and is iterated over the time steps. Lines 3 to 6 are the matrix-vector multiplications used to update the synaptic conductances that will be used

to update the synaptic currents. In this case, the subscripts EE , EI , IE , and II indicates the excitatory to excitatory, the excitatory to inhibitory, the inhibitory to excitatory, and the inhibitory to inhibitory connections, respectively. After these computations, line 7 evaluates Equations (7) and (8). The next step is the update of excitatory and inhibitory cells voltage, which is performed using Equations (1)–(3) with the two for loops in lines 8 to 13.

2.2.2. The Multicore Algorithm

The multicore algorithm exploits the OpenMP (Open Multi Processing) paradigm. It is an Application Programming Interface (API) that facilitates multi-platform shared memory multiprocessing programming in C/C++. It is specifically designed to facilitate the development of parallel applications on multicore processors by providing a set of compiler directives, library routines, and environment variables. OpenMP provides a means for developers to write parallel code that can effectively utilise the capabilities of multicore systems by distributing tasks across multiple processing cores.

One of the key advantages of OpenMP is its simplicity and ease of integration into existing codebases, which allows for incremental parallelisation without the necessity for extensive code restructuring. It supports both fine-grained and coarse-grained parallelism, thereby enabling the parallel execution of loops, sections, and tasks while maintaining a high level of control over thread creation, synchronisation, and workload distribution. The OpenMP shared memory model permits threads to communicate by reading and writing shared variables. Additionally, it provides mechanisms such as critical sections and atomic operations to circumvent race conditions.

In multicore systems, OpenMP can markedly enhance performance by optimising the utilisation of available cores, thereby reducing execution times for compute-intensive applications. Its portability across different architectures also ensures that code written with OpenMP can run efficiently on a wide range of multicore processors, making it a crucial tool for developers aiming to optimise software for parallel execution.

The serial code described in Section 2.2.1 has been annotated with OpenMP directives to indicate to the compiler which parts can be parallelized, without exploiting any numerical library. In particular, all the operations inside the main *for* loop are intrinsically parallel since they are in the form of matrix-vector multiplications or equations describing the voltage update of single cells. It must be stressed that, for each time step of the simulation, the update of the voltage of each cell is dependent only on the network state at the previous time step, the actual inputs from the CA3 layer, and the background noise. Therefore, the four matrix-vector multiplications and the *for* loops related to the excitatory and inhibitory cells can be performed in parallel. These code parts have been annotated with the `#pragma omp parallel` for directive to indicate that the iterations of the *for* loops should be equally divided on the available core of the processor. Moreover, the *private* and *shared* clauses have been used to indicate the variables that are private to the single thread and the structures that are shared among all the processing elements. Indeed, referring to Algorithm 1, the matrix-vector multiplications described in lines 3–6 have been parallelized with the aforementioned directives. Moreover, the *for* loops in lines 8 and 11 have been annotated to be parallelized by the compiler.

Finally, since the number of spikes occurring at each time step is needed to validate the simulation against the results of [5], a *reduction* clause is used to accumulate the results of all the threads in a single variable that can be seen from the master thread.

2.2.3. The GPU Algorithm

The GPU version of the hippocampus network simulation relies on CUDA (Compute Unified Device Architecture). It is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on NVIDIA GPUs. It permits developers to utilise the substantial parallel processing capabilities of GPUs to accelerate compute-intensive applications. CUDA provides extensions to standard programming languages, including C, C++, and Fortran, enabling developers to write code that executes

on the GPU. This is particularly well-suited for tasks that can be parallelised, such as matrix operations, image processing, and scientific simulations.

CUDA capitalises on the underlying architecture of the GPU, including thousands of highly efficient cores that are optimised for parallel execution. In contrast to traditional CPUs, which are optimized for sequential processing, GPUs are capable of handling thousands of threads simultaneously, making CUDA an optimal choice for problems that can be decomposed into numerous smaller tasks that can be executed in parallel.

The typical steps of a CUDA program involve:

1. GPU memory allocation: the first step is to allocate memory on the GPU through the *cudaMalloc* routine.
2. Data transfer: this step is responsible for transferring the required data from the host (CPU) memory to the device (GPU) memory. This step is of paramount importance, as GPUs possess their own dedicated memory space, and operations are conducted on data stored within this memory. The *cudaMemcpy* routine performs these memory transfers.
3. Kernel launch: the fundamental component of a CUDA program is the *kernel*, which is a function that is executed on the GPU. The kernel is initiated from the host code, with the number of threads and blocks previously specified. Each thread within a block executes operations on a portion of the data in parallel.
4. Results transfer: once the execution of the kernel is complete, the results are transferred from the device memory to the host memory. This enables the CPU to access and process the data as required. These transfers are again managed by the *cudaMemcpy* routine.
5. GPU memory de-allocation: subsequently, the allocated GPU memory is released by the *cudaFree* routine and the program terminates.

CUDA's ability to utilize the parallelism inherent in GPUs can lead to significant performance gains over traditional CPU-only implementations, particularly in applications that involve heavy numerical computations and data processing. By allowing developers to directly program the GPU, CUDA provides fine-grained control over performance, enabling optimizations that are critical in high-performance computing domains.

The efficient transfer of data between the CPU and GPU is a crucial aspect of optimising GPU performance, given that data must be moved between these two components both before and after the execution of a computation. The PCIe bus, which connects the two, has a limited bandwidth in comparison to the GPU's internal memory, which creates the potential for memory transfer bottlenecks. The implementation of efficient transfer strategies allows us to reduce latency and increase throughput, thereby ensuring that the GPU is utilised to its fullest capacity without unnecessary idle time.

Therefore, special effort has been put in minimizing memory transfers during the hippocampus CA1 network simulation. The adopted strategy consists of transfers from host to device of all the non-time dependent data after they are computed at the beginning of the simulation. Then, the main for loop iterating over the time steps start. The computation of the synaptic contribution, performed by matrix-vector multiplications, is implemented by exploiting the cuBLAS library, which contains highly optimized CUDA kernels for linear algebra operations. In particular, the *cublasDgemv* routine performs the matrix-vector product. At the beginning of the for loop, this routine is activated four times to compute the EE, EI, II, and IE contributions. After that, kernels for the excitatory and for the inhibitory neurons are activated. In this case, the number of CUDA threads generated is equal to the number of neurons managed by each kernel.

When the main for loop ends, the results are retrieved from the GPU memory and stored in suitable file for offline analysis. The flow of the GPU algorithm, shown in Figure 3, also highlights the memory transfers. The three blue dashed arrows represent the output of the serial functions that are transferred from the host to the device memory. These transfers are conducted collectively, preceding the main for loop. Conversely, the red dashed lines illustrate the data transfer from device to host memory, emphasizing the role of the kernel

in generating these outcomes. Once more, the memory transfers are grouped together and executed at the conclusion of the main *for* loop.

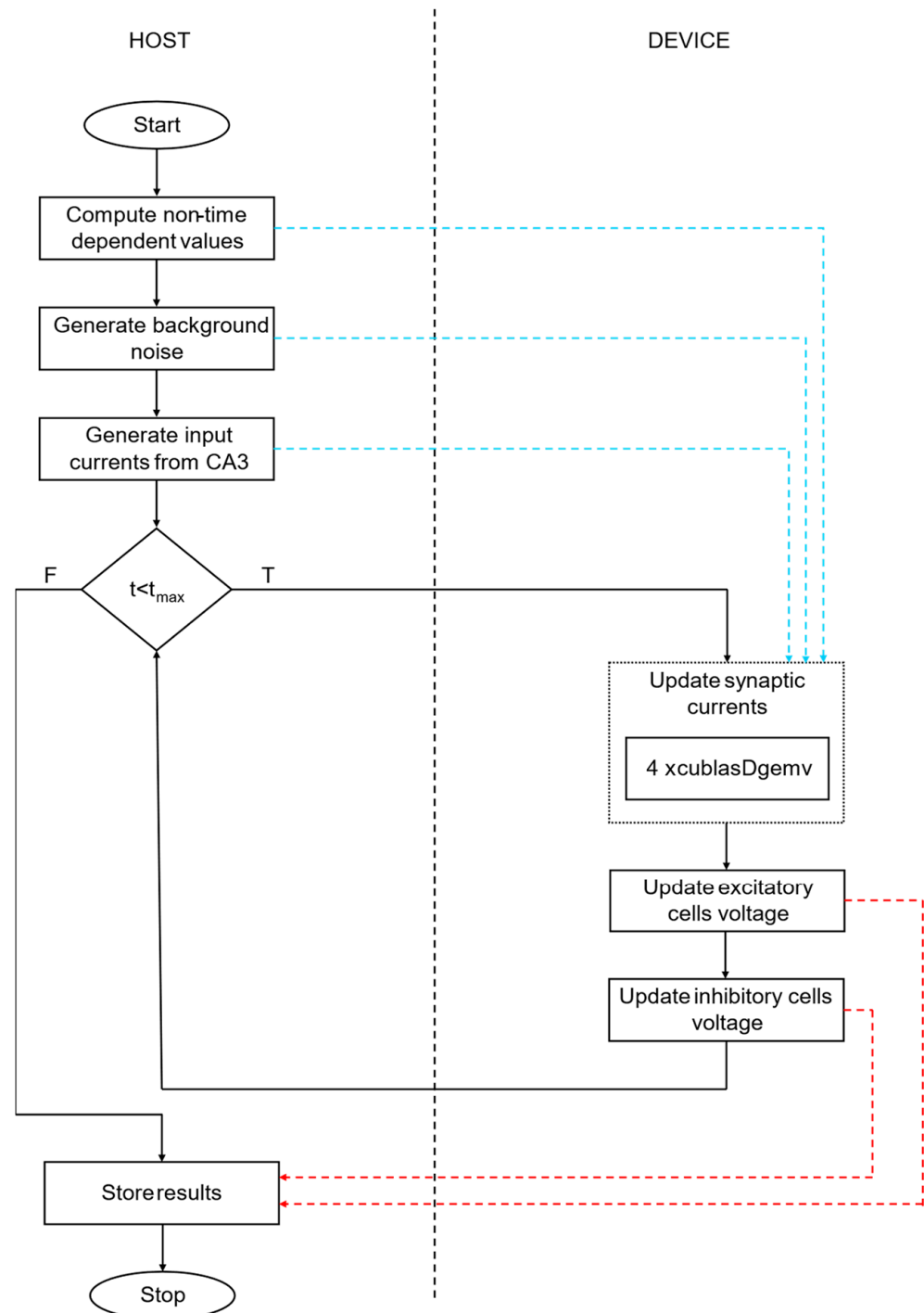


Figure 3. Parallel code flow. The blue dashed arrows represent the memory transfers from host to device, while the red dashed arrows indicate the memory transfers from device to host.

3. Experimental Results

This section describes the main features of the systems used to validate the performance of the serial and parallel codes, together with the executed experiments and their results.

3.1. Hardware Features of the Test Systems

The serial and the multicore versions of the code have been validated and tested considering two different processors, namely an Intel Xeon Silver 4314 and an Intel i9-13900K. These processors have been designed to meet a variety of computing requirements, and each offers a distinctive set of hardware features. The Xeon Silver 4314 is designed for use in enterprise-grade servers and data centres, where reliability, scalability, and multi-threaded performance are of critical importance. The processor features multiple cores and threads that are optimized for parallel processing, extensive cache memory, and support for error-correcting code (ECC) memory, which enhances data integrity in mission-critical environments. Additionally, the Xeon Silver 4314 is compatible with advanced virtualisation features, making it an appropriate choice for tasks such as virtualisation, database management, and large-scale computing.

Conversely, the Intel Core i9-13900K is a high-performance desktop processor designed for programmers, gamers, and content creators. The processor in question boasts higher base and turbo clock speeds, which are designed to deliver exceptional single-threaded performance and responsiveness in gaming and creative applications. Additionally, the i9-13900K incorporates Intel's hybrid architecture, comprising a combination of Performance-cores (P-cores) and Efficient-cores (E-cores), which enables it to manage a range of tasks with optimal efficiency, from demanding gaming to background operations.

The main features of these two processors are summarized in Table 2.

Table 2. Hardware features of the two adopted processors.

Feature	Intel Xeon Silver 4314	Intel i9 13900K
# Core/threads	16/32	24 (8 P-cores, 16 E-cores)/32
Base clock speed [GHz]	2.4	3.0 (P-cores) 2.2 (E-cores)
Max turbo frequency [GHz]	3.4	5.8 (P-cores)
Cache [MB]	24	36
Thermal Design Power [W]	235	253
RAM [GB]	768	64

Concerning the CUDA version of the hippocampus CA1 network simulation, two GPUs have been considered: NVIDIA A16 and NVIDIA RTX 4090 (NVIDIA, Santa Clara, CA, USA). These GPUs are both powerful, yet they are designed for markedly disparate use cases, offering distinctive hardware features to meet particular needs. The NVIDIA A16 GPU features a multi-GPU architecture, whereby a single card incorporates four discrete GPUs, each comprising its own set of CUDA cores, memory, and other processing units. This configuration enables the A16 to effectively manage multiple concurrent virtual desktops or workstations on a single card, offering up to 64 users per card with dedicated GPU resources. The independence of the four GPUs guarantees that workloads are isolated and can be separately managed, making the A16 an optimal choice for high-density virtual desktop infrastructure (VDI) environments where scalability and resource allocation are paramount. The device is constructed on the NVIDIA Ampere architectural framework and is intended to facilitate the provision of high-density, high-performance virtual desktops and workstations, with the capacity to support up to 64 concurrent users per card. The A16 is particularly well-suited to scenarios where multiple users require efficient sharing of GPU resources, making it an optimal choice for large-scale deployment in data centres.

In contrast, the NVIDIA RTX 4090 is a consumer-oriented GPU aimed at gamers and content creators who demand the highest levels of graphical performance. As one of the most powerful GPUs currently available, the RTX 4090 is based on NVIDIA's Ada Lovelace architecture, delivering exceptional performance in gaming, 3D rendering, AI processing, and other graphics-intensive tasks. It features a high number of CUDA cores, significant VRAM, and support for advanced ray tracing and AI features, making it the optimal choice for users seeking the best possible performance in desktop graphics. Table 3 compares the features of the two adopted GPUs.

Table 3. Hardware features of the two adopted GPUs.

Feature	NVIDIA A16	NVIDIA RTX 4090
Architecture	Ampere	Ada Lovelace
# CUDA cores	7168 total, 1792 per GPU	16,484
RAM [GB]	64 total, 16 per GPU	24
Memory bandwidth [Gb/s]	320	1008
Thermal Design Power [W]	250	450

These two devices have been selected to represent the different ranges of products developed by NVIDIA. The NVIDIA A16 is a GPU specifically developed for scientific computing, while the NVIDIA RTX 4090 is designed for rendering and for general purpose computing. By evaluating the parallel code on these two devices, the performance of the different architectural philosophies developed by NVIDIA are characterized for the neuroscientific simulations.

3.2. Experimental Setup

All the serial and parallel codes described in the previous section have been developed using the Microsoft Visual Studio 2022 Integrated Development Environment (IDE). For the serial and OpenMP codes, the compiler flags have been carefully configured to achieve the best performance in terms of processing speed. Optimisation has been set to maximise speed (*/O2*), enabling a range of optimisations such as inline expansion and loop unrolling that prioritise execution speed. To further improve performance, Inline Function Expansion (*/Ob2*) was enabled, allowing the compiler to inline functions wherever beneficial, reducing the overhead associated with function calls. To take advantage of processor-specific capabilities, the Enhanced Instruction Set has been configured to use Advanced Vector Extensions (AVX), allowing the use of SIMD instructions that accelerate data processing tasks. In addition, the floating-point model has been set to Fast (*/fp*), which allows aggressive floating-point optimisations that improve speed, albeit with a slight trade-off in accuracy.

For parallel OpenMP code, OpenMP support (*/openmp*) was enabled, allowing the compiler to recognise and efficiently handle OpenMP directives, thereby optimising the parallel parts of the code. To further maximise the performance of the whole program, program optimisation (*/GL*) was enabled, which facilitates link-time code generation (LTCG), enabling cross-module optimisation to optimise the program as a whole rather than in isolated sections. Finally, Link Time Code Generation (*/LTCG*) was used in the linking phase to ensure that the linker performs optimisations across module boundaries, further improving execution speed. By carefully configuring these compiler flags, the project exploits the full potential of the hardware, ensuring that both serial and parallel code paths are optimised for maximum processing speed, ultimately resulting in superior execution performance.

Concerning the CUDA code, to ensure the best performance targeting both the NVIDIA A16 and RTX 4090 GPUs, the compiler flags and settings have been carefully configured to optimise processing speed. Compute Capability was set according to the target GPU architecture. For the A16, based on the Ampere architecture, and the RTX 4090, based on the Ada Lovelace architecture, the *-arch=compute_80* flag was used for the A16 and *-arch=compute_89* for the RTX 4090. These flags ensure that the compiled code takes full advantage of the specific architectural features and instructions available on these GPUs. To further optimise performance, the Fast Math option (*-use_fast_math*) was enabled. This setting allows the CUDA compiler to use faster, less precise math operations, which can significantly improve performance in scenarios where a slight reduction in precision is acceptable. Parallel Thread Execution (PTX) optimisations have been fine-tuned by enabling link-time optimisations (*-dlto*). This allows the compiler to optimise across multiple files and modules, ensuring the final executable is as efficient as possible, especially for large, complex CUDA projects. Finally, memory transfers between the host and device have been streamlined by enabling asynchronous transfers using streams and setting the *-Xptxas*

-dlcm=ca flag to cache global loads in L1, which is critical to minimising data transfer overhead, especially in high throughput applications.

By carefully configuring these compiler flags and optimisations, CUDA code is fine-tuned to take full advantage of the NVIDIA A16 and RTX 4090 GPUs. This results in significant performance improvements, ensuring that code runs as efficiently as possible on these high-performance GPUs, whether in data centre environments or high-end desktop systems.

The serial and OpenMP codes have been tested on all the processors described in Section 3.1, while the CUDA algorithms have been tested on the NVIDIA RTX 4090 connected to the Intel i9 processor and on the NVIDIA A16 connected to the Intel Xeon Silver 4314.

The hippocampus CA1 network was simulated to reproduce 2 s of activity. This time was chosen according to the simulation conducted in [5].

3.3. Biological Validation

All the codes described in this paper have been validated against the results reported in [5]. In particular, the input from the CA3 area of the hippocampus has been modelled as bell-shaped currents, with a maximum value of 700 pA for the interneurons and of 210 pA for the pyramidal cells. These bell-shaped currents have a duration of 100 ms and are repeated every 130 ms. Figure 4 shows the two input currents traces of 1 s.

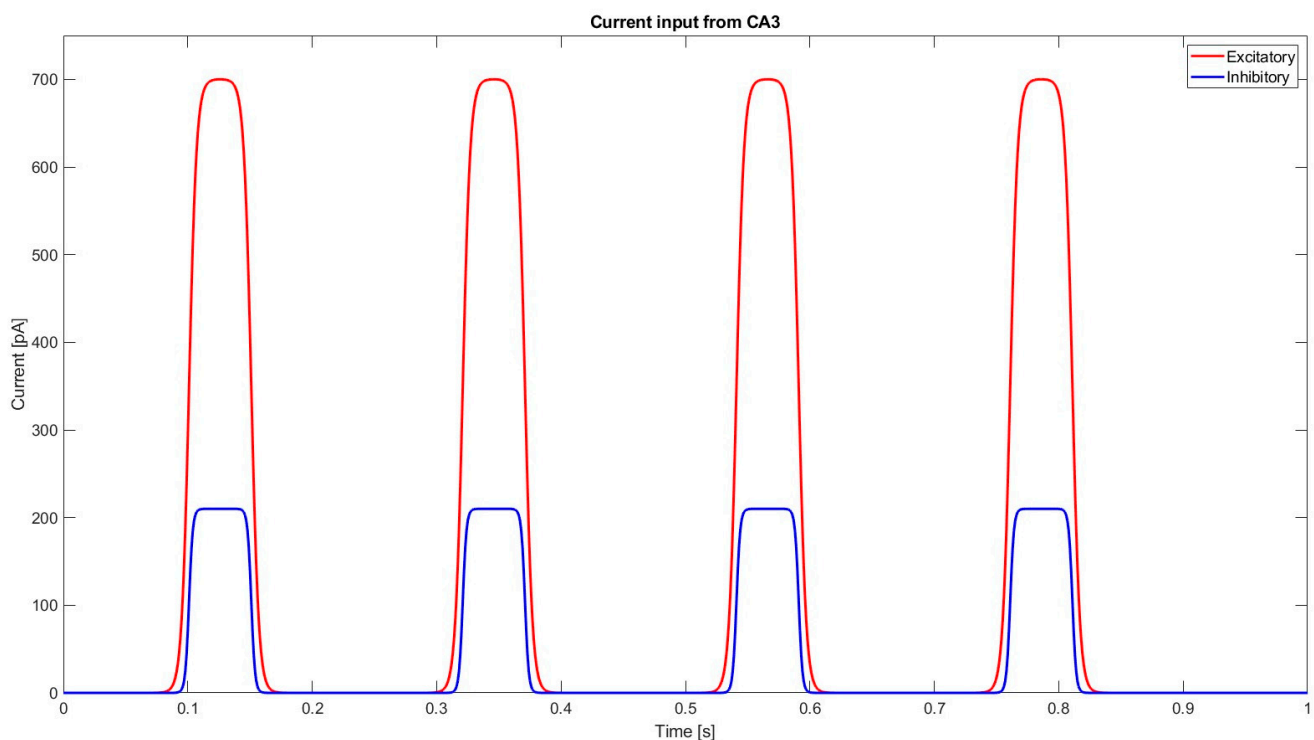


Figure 4. Input traces of the input currents.

The excitatory and inhibitory neurons show different responses to the input currents. Indeed, the interneurons features a ripple frequency normally distributed around 163.5 Hz with a standard deviation of 20.6 Hz. On the other hand, the pyramidal cells do not show ripples. As an example, Figure 5 shows the voltage of a pyramidal cell and of an interneuron in response to the input currents shown in Figure 4. These results confirm that the outputs produced by the developed model are the same as in [5].

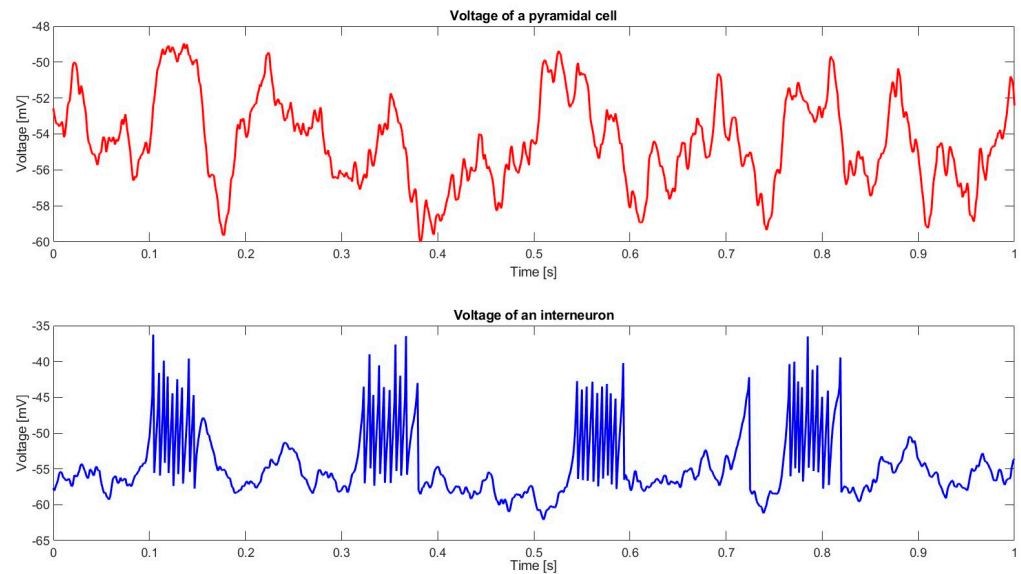


Figure 5. Voltage traces of two cells included in the network.

3.4. Performance Evaluation

The processing time of all the serial and parallel codes have been measured on the system described in Section 3.1. In addition to the experiment concerning the same network described in [5] i.e., considering 800 pyramidal cells and 160 interneurons, two other networks have been considered to evaluate the scalability of the simulation. In particular, a smaller network with half the neurons (i.e., 400 excitatory and 80 inhibitory cells) and a bigger network with 1200 pyramidal cells and 240 interneurons have been simulated. In the following, the smallest network will be denoted as Network1, the original as Network2, and finally the biggest network will be referenced as Network3. All the simulations were repeated 10 times and the mean processing times, together with the standard deviations, have been computed. The serial processing of the three networks on the Intel i9 13900K took 192.25 s, 740.54 s, and 983.76 s, respectively. The same networks took 552.84 s, 2141.91 s, and 2504.97 s on the Intel Xeon Silver 4314. The memory footprint of the three networks is 8.7 GB, 15.2 GB and 21.2 GB, respectively. These memory footprints impose a limitation on the GPUs that can carry out the simulations.

For the OpenMP codes, the experiments considered a variable number of threads, from 16 to 64, to find the optimal number of processing elements for each processor. Figure 6 shows the mean processing times of the Intel i9 13900K and of the Intel Xeon Silver 4314 for the three Networks for the serial and parallel codes, and with a variable number of threads. In all these cases, the standard deviation was less than 2%.

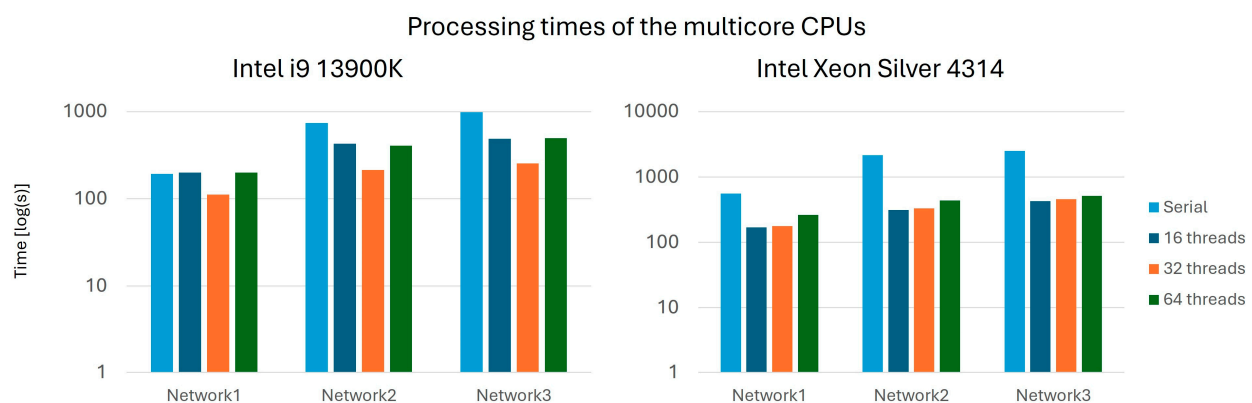


Figure 6. Processing times of the multicore CPUs with varying number of threads. The y-axis is in logarithmic scale.

The same methodology, apart from varying the number of threads, was repeated for the two GPUs described in Section 3.2. Network3 has been simulated only on the NVIDIA RTX 4090 board due to memory constraints. Figure 7 reports the mean processing time for these devices for the three networks. Again, the standard deviation is low since it is always less than 2%.

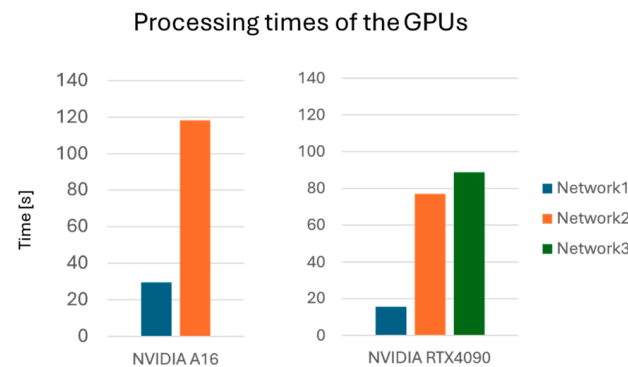


Figure 7. Processing times of the GPUs.

Finally, the speed-up between the serial processing on the serial processors and the best OpenMP for each multicore CPU and the GPUs have been computed to characterize the impact of the parallelization of the hippocampus CA1 network simulation. Table 4 reports these best processing times, together with the speed-up indicated between brackets.

Table 4. Best processing times and speed-up. All the processing times are reported in seconds.

Network	Serial		OpenMP		NVIDIA RTX 4090	NVIDIA A16
	Intel i9	Intel Xeon	Intel i9	Intel Xeon		
Network1	192.95	552.84	111.96 (1.72×)	169.40 (3.26×)	15.58 (12.38×)	29.51 (18.73×)
Network2	740.54	2141.91	213.25 (3.47×)	310.62 (6.90×)	77.01 (9.74×)	118.32 (18.10×)
Network3	983.76	2504.97	253.48 (3.88×)	425.37 (5.89×)	88.76 (11.08×)	-

4. Discussion

Considering the OpenMP performance reported in Figure 6, the Intel i9 processor shows a decrease in processing time as the number of threads increases from 16 to 32, but this decrease is not linear. However, there is no improvement when increasing to 64 threads. On the other hand, the Xeon Silver processor performs best with 16 threads, with processing times increasing as the number of threads increases.

This behavior is maintained by both processors for the three networks. In conclusion, the Intel i9 processor performs best with 32 threads. This suggests that using more than 32 threads can lead to inefficiencies, probably due to synchronization overheads or memory contention at least with these dimensions of the problem. The Intel i9 appears to be better suited to workloads requiring moderate levels of parallelism, offering significant performance gains up to 32 threads. However, its performance does not improve and sometimes deteriorates when scaling beyond this point. Conversely, the Xeon Silver, while generally slower, handles higher thread counts more consistently, but also struggles beyond 32 threads. These observations suggest that for optimal performance, thread counts must be carefully matched to both problem size and processor architecture to avoid over-parallelization, which can lead to diminishing returns.

Analyzing the GPUs performance reported in Figure 7, the following considerations can be made. For Network1, the RTX 4090 completes the task in 15.58 s, while the A16 takes longer at 29.512 s. This suggests that the RTX 4090 exhibits superior performance

for smaller workloads, potentially attributable to its elevated clock frequency and a more sophisticated architectural design optimized for high-throughput computing.

As the problem size increases, the processing time of the RTX 4090 rises to 77.01 s. In contrast, the A16 requires 118.32 s to process the same workload. This serves to further emphasize the performance gap between the two GPUs, with the RTX 4090 continuing to demonstrate superior efficiency and speed. The A16, despite its design for high-density, multi-user environments, appears to encounter greater challenges as the problem size increases. This observation reflects its emphasis on versatility for multi-user environments rather than on the raw performance on a single task.

With regard to the largest problem size, the RTX 4090 records a processing time of 88.76 s. However, no corresponding value is available for the A16, since the available memory is not enough to perform this simulation. This indicates that while the A16 is effective in scenarios that require moderate parallelism and multi-user environments, it may not be as well-suited for extremely large, computationally intensive tasks that demand the substantial processing power provided by the RTX 4090.

In conclusion, the RTX 4090 demonstrates superior performance compared to the A16 across a range of tested problem sizes, particularly when the workload is increased. The A16 is capable in specific environments, yet it is unable to match the RTX 4090's performance in high-computational tasks. These findings illustrate that the RTX 4090 is well-suited for scenarios that require maximum GPU throughput and computational speed, whereas the A16 may be more appropriate for environments that emphasize workload distribution and resource sharing across multiple users.

By analyzing the speed-ups, it can be noted that all the parallel versions accelerate the serial processing. In particular, focusing on the Intel i9 processor, the speed-up increases with the problem size. Indeed, the smallest speed-up is related to Network1 and is about $1.7\times$, while it increases up to $3.8\times$ for Network3. Similar considerations can be conducted for the Intel Xeon Silver 4314 CPU. However, the speed-up ranges from about $3.2\times$ to $6.9\times$. Again, the smallest speed-up is related to Network1. In this case, the best speed-up is related to Network2, while there is a slight decrease for Network3.

The analysis of the speed-up for the GPUs algorithm led to the following considerations. The NVIDIA RTX 4090 has the best performance in terms of processing times, but the speed-up ranges from about $9.7\times$ to $12.3\times$, while the NVIDIA A16 shows a constant speed-up of about $18\times$ for Network1 and Network2.

A comparison between all these parallel algorithms on different devices can be made by introducing a figure of merit that combines the product between the thermal design power and the processing time divided by the number of cores. This figure of merit can be interpreted as the energy dissipated divided by the total number of cores. The values are reported in Table 5.

Table 5. Values of the figure of merit.

Network	Intel i9	Intel Xeon Silver 3414	NVIDIA RTX 4090	NVIDIA A16
Network1	1180.23	2488.03	0.43	4.27
Network2	2248.01	4562.31	2.12	17.11
Network3	2672.08	6247.59	2.44	-

These values highlight that the GPU technology is the most effective for this kind of simulation. Indeed, there are three orders of magnitude between the values obtained for the GPUs and the ones achieved by the multicore CPUs.

It is worth noticing that a direct comparison between the proposed work and the state of the art would not be fair since the works in [14–16] adopt different mathematical models to describe the hippocampus CA1 area, together with the adoption of supercomputing facilities rather than desktop-based solutions.

5. Conclusions

This paper described the development of a parallel simulation of the ripple activity of the hippocampus CA1 area. The parallel codes have been validated against the original model described in [5]. The performance has been evaluated on different multicore CPUs and GPUs, considering three networks of increasing size (i.e., number of elements to simulate). Ten simulations for each device and for each network have been run and profiled to obtain the mean processing times and the standard deviations.

The analysis of the results highlighted that all the parallel codes outperform the serial processing, maintaining the correctness of the biological behaviors. Moreover, the GPUs achieved the best performance in terms of processing times, with the NVIDIA RTX 4090 emerging as the most suitable device for this kind of simulations.

Finally, a direct comparison with the state of the art revealed to be unfair, since the models and the computing facilities are too different from the ones considered in this paper. Thus, to the best of the authors' knowledge, this is the first paper proposing the parallel simulation of the hippocampus CA1 area reproducing the wave-ripples.

Future research will be focused on integrating the models of the other areas to reproduce the functionality of the whole hippocampus.

Author Contributions: Conceptualization, E.T. and F.L.; methodology, E.T.; software, S.M. and E.T.; validation, E.T., S.M. and E.M.; data curation, E.T. and E.M.; writing—original draft preparation, E.T.; writing—review and editing, G.D., E.M. and F.L.; supervision, F.L.; project administration, F.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to privacy.

Conflicts of Interest: The authors declare no conflicts of interest.

References

- Diekelmann, S.; Born, J. The Memory Function of Sleep. *Nat. Rev. Neurosci.* **2010**, *11*, 114–126. [\[CrossRef\]](#) [\[PubMed\]](#)
- Schönauer, M.; Pawlizki, A.; Köck, C.; Gais, S. Exploring the Effect of Sleep and Reduced Interference on Different Forms of Declarative Memory. *Sleep* **2014**, *37*, 1995–2007. [\[CrossRef\]](#) [\[PubMed\]](#)
- Buzsáki, G.; Horváth, Z.; Urioste, R.; Hetke, J.; Wise, K. High-Frequency Network Oscillation in the Hippocampus. *Science* **1992**, *256*, 1025–1027. [\[CrossRef\]](#) [\[PubMed\]](#)
- Ylinen, A.; Bragin, A.; Nadasdy, Z.; Jando, G.; Szabo, I.; Sik, A.; Buzsáki, G. Sharp Wave-Associated High-Frequency Oscillation (200 Hz) in the Intact Hippocampus: Network and Intracellular Mechanisms. *J. Neurosci.* **1995**, *15*, 30–46. [\[CrossRef\]](#) [\[PubMed\]](#)
- Malerba, P.; Krishnan, G.P.; Fellous, J.-M.; Bazhenov, M. Hippocampal CA1 Ripples as Inhibitory Transients. *PLoS Comput. Biol.* **2016**, *12*, e1004880. [\[CrossRef\]](#) [\[PubMed\]](#)
- Bezaire, M.J.; Soltesz, I. Quantitative Assessment of CA1 Local Circuits: Knowledge Base for Interneuron-Pyramidal Cell Connectivity. *Hippocampus* **2013**, *23*, 751–785. [\[CrossRef\]](#) [\[PubMed\]](#)
- Pastur-Romay, L.A.; Porto-Pazos, A.B.; Cedron, F.; Pazos, A. Parallel Computing for Brain Simulation. *Curr. Top. Med. Chem.* **2017**, *17*, 1646–1668. [\[CrossRef\]](#) [\[PubMed\]](#)
- Casali, S.; Marenzi, E.; Medini, C.; Casellato, C.; D'Angelo, E. Corrigendum: Reconstruction and Simulation of a Scaffold Model of the Cerebellar Network. *Front. Neuroinform.* **2019**, *13*, 51. [\[CrossRef\]](#) [\[PubMed\]](#)
- Eichner, H. Neural Simulations on Multi-Core Architectures. *Front. Neuroinform.* **2009**, *3*, 476. [\[CrossRef\]](#) [\[PubMed\]](#)
- Knight, J.C.; Nowotny, T. Larger GPU-Accelerated Brain Simulations with Procedural Connectivity. *Nat. Comput. Sci.* **2021**, *1*, 136–142. [\[CrossRef\]](#) [\[PubMed\]](#)
- Florimbi, G.; Torti, E.; Masoli, S.; D'Angelo, E.; Leporati, F. Granular Layer Simulator: Design and Multi-GPU Simulation of the Cerebellar Granular Layer. *Front. Comput. Neurosci.* **2021**, *15*, 630795. [\[CrossRef\]](#) [\[PubMed\]](#)
- Wang, R.M.; Thakur, C.S.; van Schaik, A. An FPGA-Based Massively Parallel Neuromorphic Cortex Simulator. *Front. Neurosci.* **2018**, *12*, 213. [\[CrossRef\]](#) [\[PubMed\]](#)
- Bogdan, P.A.; Marcinnò, B.; Casellato, C.; Casali, S.; Rowley, A.G.D.; Hopkins, M.; Leporati, F.; D'Angelo, E.; Rhodes, O. Towards a Bio-Inspired Real-Time Neuromorphic Cerebellum. *Front. Cell Neurosci.* **2021**, *15*, 622870. [\[CrossRef\]](#) [\[PubMed\]](#)

14. Liu, X.; Zeng, Y.; Zhang, T.; Xu, B. Parallel Brain Simulator: A Multi-Scale and Parallel Brain-Inspired Neural Network Modeling and Simulation Platform. *Cogn. Comput.* **2016**, *8*, 967–981. [[CrossRef](#)]
15. Gandolfi, D.; Mapelli, J.; Solinas, S.M.G.; Triebkorn, P.; D’Angelo, E.; Jirsa, V.; Migliore, M. Full-Scale Scaffold Model of the Human Hippocampus CA1 Area. *Nat. Comput. Sci.* **2023**, *3*, 264–276. [[CrossRef](#)] [[PubMed](#)]
16. She, X.; Robinson, B.; Flynn, G.; Berger, T.W.; Song, D. Accelerating Input-Output Model Estimation with Parallel Computing for Testing Hippocampal Memory Prostheses in Human. *J. Neurosci. Methods* **2022**, *370*, 109492. [[CrossRef](#)] [[PubMed](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.