Leveraging the Versal Adaptive SoC Heterogeneity for Large-Scale Neural-Mass Brain Modeling

Towards Decoding the Seizure Storm

Project Report, AMD Open Hardware Contest 2024

Amirreza Movahedin

TU Delft

Erasmus MC

Leveraging the Versal Adaptive SoC Heterogeneity for Large-Scale Neural-Mass Brain Modeling

Towards Decoding the Seizure Storm

by

Amirreza Movahedin

TU Delft
Erasmus MC

Contents

1 Introduction

For centuries, understanding the brain and how it works has been of great concern for scientists. The National Academy of Engineering of US has classified reverse engineering the brain as one of the grand challenges of the 21st century [9]. Many efforts have been done to better understand different aspects of the brain, and due to the recent advancements in computational tools, brain simulation has been one of the leading research areas in neuroscience. Brain simulations can be performed for different purposes, for neuroscientists to unlock the mystery of neurons, the building block of the brain. Or for doctors to use results from simulation in medical settings, to understand and tackle brain disorders. The endeavor to build better and more biologically accurate Artificial Intelligence systems can also benefit from brain simulations.

The field of computational neuroscience, which studies the organization of the brain and how they process information, has had many successful efforts in simulating the brain on different levels, from detailed modeling and simulation of a single neuron to large-scale brain network simulation. Nowadays, the whole-brain simulation is effectively being used in addressing brain diseases, with one of its prime examples being in the study of epilepsy [29]. Epilepsy affects more than 50 millions people around the world, and is characterized by recurrent spontaneous seizure attacks. Many cases of epilepsy (around 70%) can be contained with the use of drugs, however the rest might need surgical treatment. The Virtual Epileptic Patient (VEP) [16] is an effort to create a personalized digital twin for the epileptic patient's brain that can help with the surgery. Since epilepsy is a condition involving several connected brain structures, the VEP can help the surgery by estimating the regions that are affected. Using VEP to identify only the affected parts of the brain can help the surgery to go without causing any neurological problems, and as a result reducing its risk. VEP and similar whole-brain network models such as the ones meant for Alzheimer's, Parkinson, or Schizophrenia, are being used extensively in the medical setting, providing new insights for scientists and personalized medicine for affected patients [16].

In another application, whole-brain network simulation can be used in a control unit of a brain interface system [30, 23]. In these examples, there are physical invasive or non-invasive connections to the brain for different applications. For example, a closed-loop control system can perform deep brain stimulation to suppress high-amplitude epileptic activity. However, there is the challenge of how the system can analytically calculate the stimulus parameters it is gonna perform. Whole-brain network simulation can be used as part of the closed-loop control in these systems to more accurately and effectively find the appropriate parameters for the stimulation that suppresses the seizure attacks.

1.1. Motivation

The whole-brain network models such as VEP rely on patient-specific parameters extracted from the individual's brain imaging data. To build these personalized models, the following steps are taken [16]:

1. The whole-brain model structure, meaning the high-level brain regions that are of concern of the application, are specified using the unique anatomical structure of the subject.

2. The connectivities between the specified brain regions and other parameters are mapped to the whole-brain model. The connectivites can be inferred using different methods, from statistical analysis on the brain activity data to using imaging techniques.
3. Relevant clinical parameters of the model (such as the strength of the connections) are inferred.

The last step is usually done by simulating the model with different parameters and fitting it in small increments to the actual data taken from the patient. In other words, a leaning mechanism runs the simulation many times and for each time it checks how close the output of the model is to the actual brain data, and according to this, the parameters of the model are tweaked to get it closer to the actual model. This process, which is referred to as model fitting, requires a large number of simulation which could take a long time to finish, making achieving the goal of having a personalized, patient-specific, digital-twin of the brain challenging. Furthermore, as mentioned earlier in this chapter, this digital brain twin can also be used in closed-loop control system that interface with brain, and require quick and real-time simulations. In conclusion, **having a high-performance platform that can run such whole-brain network models will significantly help with building and utilizing patient-specific digital brain twins.**

1.2. Report Organization

In Chapter 2 the background information and relevant topics, problem formulation, and related works are presented. In Chapter 3, the problem at hand is analyzed in terms of memory and computation. Additionally, three design candidates are presented and compared. In Chapter 4, the details of the implemented systems are discussed in depth. In 5, the performance results of the implemented systems in addition to the related works are presented and compared. And finally, in Chapter **??**, concluding remarks and suggested future work directions are presented.

2

Background and Related Work

2.1. The Human Brain

The human brain is arguably the most complex system that we know of. It consists of tens of billions of neurons, interacting with each other in a complicated way [11]. There are different types of neurons, however almost all of them have the same structure. A neuron is composed of soma (or the cell body), dendrites, and axon as illustrated in Figure 2.1. The dendrites act as the input of the neuron, receiving the spikes from other neurons through connections called synapses. The soma is the processing part of the neuron, and axon can be seen as the output. The brain consists of a network of individual neurons, allowing them to communicate with each other to perform different functionalities.

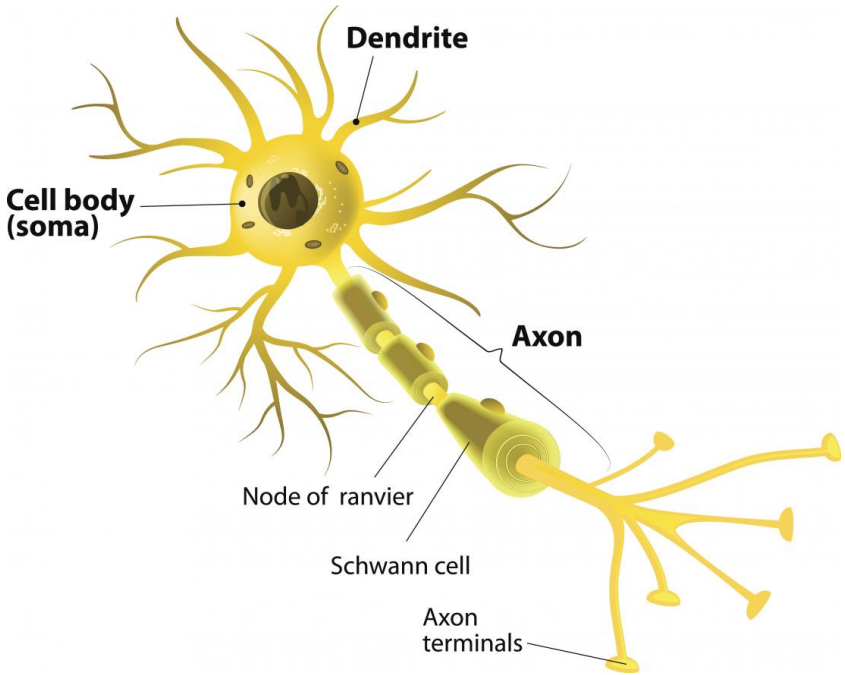


Figure 2.1: Structure of a Neuron [28]

The brain, as a whole, is shaped like a walnut, divided into left and right hemispheres. Through observing the effects of injuries and strokes, we know that different areas in the brain perform different processes [11]. For instance, there are different regions in the brain for tactile sensation, movement control, hearing, vision, or speech. Figure 2.2 shows these different regions of the brain and their main functionality.

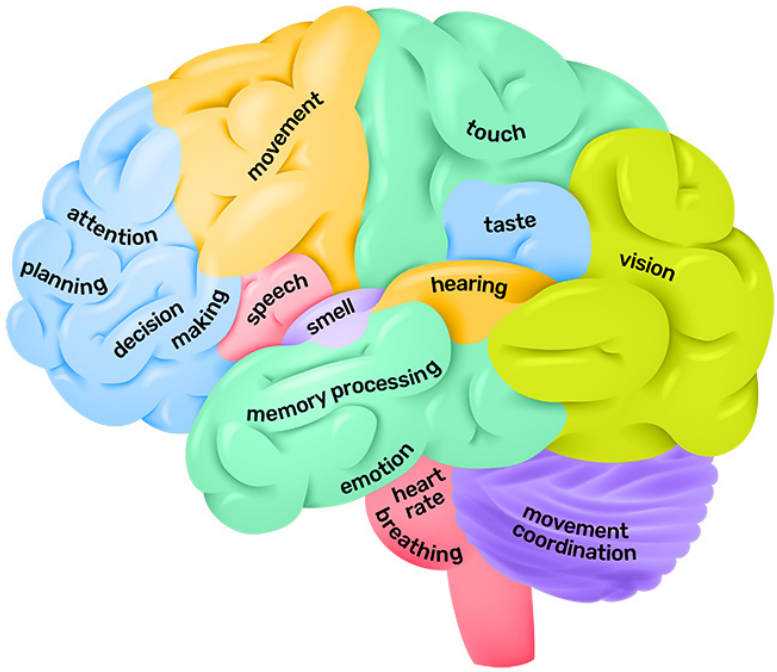


Figure 2.2: Brain Regions and Their Functionality [24]

2.2. Neural-Mass Models and Whole-Brain Simulation

Neurons, as a physical system, are non-linear elements that are capable of producing spikes. These spikes are the way that neurons communicate with each other. Over the history of computational neuroscience, brain modeling has being performed at different levels of abstraction [21]. Early efforts of modeling the brain were focused on the simulation of a single neuron [12, 10]. Generally, a model of a neuron consists of a set of non-linear differential equations with variables describing different characteristic of the neuron. The single-neuron models provided foundational understanding of the behavior of a single neuron. However, they did not offer much insight on how a population of neurons interact with each other to perform a certain task.

In an effort to understand the activity patterns of a population of neurons, neural-mass [31, 4] and neural-field [15, 1] models were formulated. Neural-mass models looked at the activity of a mass of neurons instead of just a single neuron, were less chaotic and irregular activities are found compared to just one neuron [8]. Although these models had a great success in theoretically explaining many neural activities, they had limited applicability in the clinical settings [21]. Many neuronal phenomenon which were observed in practice could not be formulated mathematically and solved analytically within the boundaries of these models.

With brain imaging (such as fMRI) being used more for cognition studies, the need to incorporate the data gathered by these studies into the brain modeling and simulation increased [21]. This means that patient-specific imaging data could be combined with brain simulations to enable more precise medical practices [25, 7]. In this fashion of simulation, the nodes (which represent the activity of a collection of neurons in a brain area and are described by differential equations) are connected to each other based on a connectivity pattern. These connections have different parameters, and since we want the closest model to the actual brain possible, different connection parameters are explored to find the best fit to the real data. In other words, many values for the connection parameters are simulated, the result of the simulation is compared to the extracted real data from the patient's brain, and using fitting algorithms (such as gradient decent or Bayesian inference) the parameters are learned.

The Virtual Brain (TVB) [22] is one of the leading frameworks in providing whole-brain modeling.

TVB uses biologically realistic connectivity data to perform full brain network simulations. In the whole-brain network simulation, the brain is divided into different regions (represented by a node) and the activity and connections of these regions are modeled. The brain can be divided into these regions in many ways [20], each for a certain reason and use. Figure 2.3 shows the default dataset of the brain in the TVB with 74 nodes. Each division of the brain can be summarized into two matrices, a weight matrix and a distance matrix. These matrices are square, and their dimension depends on the number of nodes in the model. The weight matrix shows the existence and strength of connections between the nodes, and the distance matrix shows the distance between each pair of nodes.

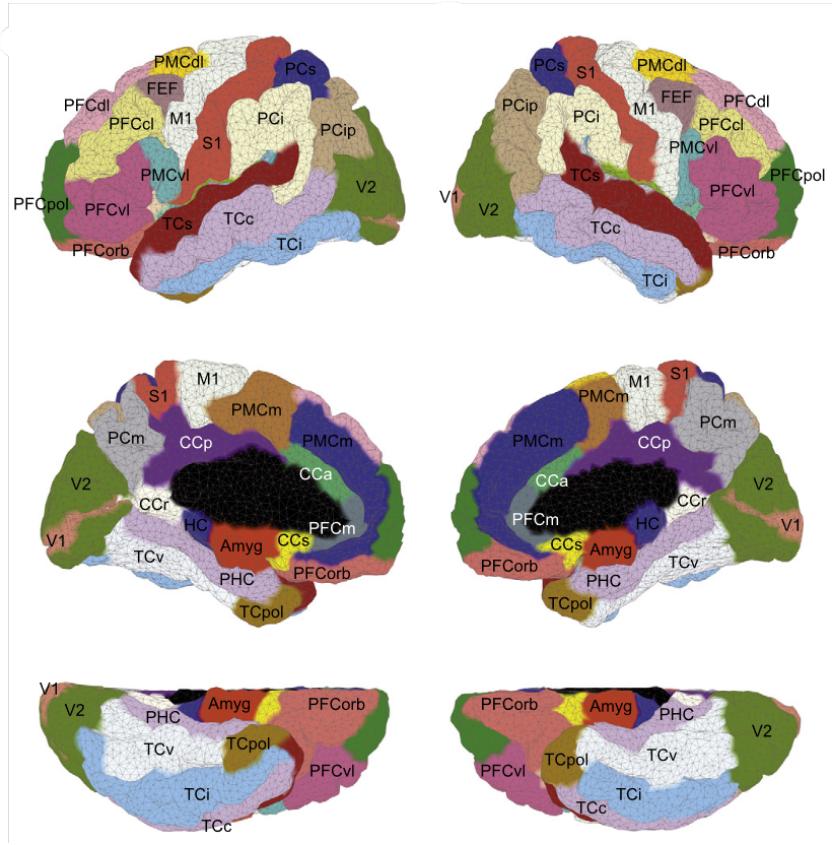


Figure 2.3: TVB Default Brain Dataset [26]

In order to perform the whole-brain network simulation, one must solve the differential equation of the nodes of the brain, for instance the ones that are shown in Figure 2.3. In their general form, the differential equations in the TVB are: 1) coupled, due to the existence of the connections between centers, and 2) delayed, due to the signal propagation speed being limited. In general, numerical methods, such as different orders of Runga-Kutta, are used to solve these coupled delayed differential equations [18].

2.3. Problem Formulation

The problem at hand is the acceleration of **N**eural-**M**ass **M**odels (NMMs). As mentioned in Section 2.2, NMMs model the populations of neurons (referred to in this document as *centers*) and the information transfer between them. Each center is modeled with a number of variables (referred to as *state variables*), and its behavior independent of other centers is usually modeled with differential equations and it is referred to as its local dynamics. The effect of the centers on each other are called the *couplings*, and are modeled as input variables to the differential equation of the centers. A centers' effect on another center takes time to arrive, as the in the physical brain signals travel at limited speeds. As a result, our problem boils down to solving a series of *coupled delayed differential equations*. NMMs generally consist of N centers and each center has M state variables. For each center $i \in [1, N]$, we

can define the state variables as $\vec{X}_i \in \mathbb{R}^M$. Centers are connected to each other in a network and with a certain distance. We define connectivity and delay matrices $W \in \mathbb{R}^{N\times N}$ and $D \in \mathbb{R}^{N\times N}$ respectively.

Matrix W shows whether two centers are connected to each other, and if they are, how strong is their connection. W_{ij} shows the strength of connection from center j to center i. If this value is zero, it means that there is no connection from center j to center i. It is worth noting that matrix W is not necessarily a symmetrical matrix. Matrix D shows the delay between 2 centers usually with a unit of seconds or milliseconds. The delay between 2 centers is calculated by dividing the distance between them by the speed of signal propagation in the brain. This matrix is symmetrical, since the distance from center i to j is the same as the distance from center j to i.

NMMs can generally be described as shown in Equations 2.1 and 2.2.

$$\frac{d\vec{X}_i(t)}{dt} = F\left(\vec{X}_i(t),\, \vec{C}_i(t)\right) + u(t) + Noise \tag{2.1}$$

$$\vec{C}_i(t) = K_{post}\left(\sum_{j=1}^{N} W_{ij} K_{pre}\left(\vec{X}_j(t - D_{ij}), \vec{X}_i(t)\right)\right) \tag{2.2}$$

- The local dynamics of the centers are represented with $F\left(\vec{X}_i(t),\, \vec{C}_i(t)\right) = \begin{bmatrix} f_1(X_{i0}(t),\, C_{i0}(t)) \\ f_2(X_{i1}(t),\, C_{i1}(t)) \\ \dots \end{bmatrix}$. In order to make this platform general for all sorts of NMMs, and more accurate for each patient in addition to, this vector of functions can be replaced by a Multilayer Perceptron (MLP) that can approximate F, meaning $MLP\left(\vec{X}_i(t),:\, \vec{C}_i(t)\right) \approx F\left(\vec{X}_i(t),\, \vec{C}_i(t)\right)$. This is possible due to the fact that MLPs are universal function approximators [13]. The MLPs that represent the local dynamics of the centers are trained using NeuralODE [6] techniques [3].
- The effect of other centers on center i is referred to as the *coupling input* to center i. This is represented with $\vec{C}_i(t) \in \mathbb{R}^M$ in Equation Equation 2.1. As shown in Equation Equation 2.2, the coupling input of node i at time t depends on the previous state variable values of other centers (according to the delay between center i and other centers) and the strength of their connection. Functions $K_{pre} \in (\mathbb{R}^N, \mathbb{R}^N) \to \mathbb{R}$ and $K_{post} \in \mathbb{R} \to \mathbb{R}^N$ are pre- and post-synapse functions respectively. They scale the signals from other centers to more realistically represent their strength when reaching the receiving center [26].
- In $MLP\left(\vec{X}_i(t),\, \vec{C}_i(t)\right)$, the coupling input $\vec{C}_i(t)$ is usually a linear additive. This allows us to rewrite the MLP function as $MLP\left(\vec{X}_i(t)\right) + \vec{C}_i(t)$ to help with the speed of calculation. Even when the coupling input is not a linear additive to the MLP function, the MLP can be trained with the new structure using some workarounds to ensure the validity of the MLP output.
- The input stimulus to the center i is shown with $u(t)$. Additionally, the noise present in the system is also taken into account. This noise term is usually a Brownian motion approximated by a standard normal random variable.

We aim to solve this differential equation with the Forward Euler method. In order to do so, we define the step size h and timestep-delay matrix $d \in \mathbb{R}^{N\times N}$ which is the result of dividing the elements of matrix D by step size h. This means that matrix d shows the delays between the centers in simulation timesteps. When solving the differential equation with the Forward Euler method, we also discretize time (t). So from this point on, the variable t refers to the t-th timestep of the Forward Euler method. Equations 2.3 and 2.4 show the Forward Euler integration for the NMM problem. In these equations, for the sake of simolicity, the effect of centers on each other happens through only one state variable. This means that $\vec{X}_i$ consist of one *global* variable that can be seen by all centers and multiple local variables that can only be seen by center i itself. Additionally, the entry point of the coupling input for each center happens through only one of the M state variables. This basically implies that the value of the coupling input to each center is only added to one of the M state variables. Furthermore, the input and noise terms ($u(t)$ and $\xi(t)$ respectively) are ignored for now as they are not essential to the

functionality of the system.

$$\vec{X}_i(t+1) = \vec{X}_i(t) + h \cdot \Big(MLP(\vec{X}_i(t)) + C_i(t) \Big) \tag{2.3}$$

$$C_i(t) = K_{post} \left(\sum_{j=1}^{N} W_{ij} K_{pre} \left(X_j(t - d_{ij}), X_i(t) \right) \right) \tag{2.4}$$

2.4. Related Work

In this project we try to implement the whole-brain simulators using TVB-style models on the Versal Adaptive SoC. The related work to our project is mainly the work done by TVB team, which comes in different versions of the TVB tool.

2.4.1. TVB on CPU

Multiple version of TVB is available in CPU, with the most notable of them are as follows:

1. The original TVB [22] is a Python-based tool for large-scale brain modeling available to neuroscientists. The numerical algorithms behind the oroginal TVB is extracted from the main tool by the TVB team and is open to public and is referred to as `tvb_algo` [32]. This code is also in Python and shows the basic algorithms and calculations performed in the TVB system. Based on this system, we developed a code in C++ where the operations done using the `numpy` library in Python are replaced by normal low-level C++ code (`tvb_algo_c`) [19]. This code was used to validate the result of the Versal system throughout the development process, in addition to a purely sequential version of the TVB system. The local dynamics of the centers are calculated using an MLP in the C++ version, something that was missing in the Python code. Furthermore, the coupling calculations are done using sparse representation which improves the performance of the simulation.
2. The Fast TVB [27] is a specialized and optimized C implementation of TVB for a specific neural-mass model. The Fast TVB sacrificed the generality of the original TVB for higher performance only for a single model. Using the CPU resources efficiently, the Fast TVB allows for simulating even very large models in a reasonable time.
3. TVB C++ [17] is another faster version of the original TVB. TVB C++ is more general compared to the Fast TVB, providing the flexibility of the original TVB to some extent.

2.4.2. TVB on GPU

The original TVB is also implemented as a JAX-based [5] Python package for execution on the GPU [33]. JAX is a Python library developed by Google for array computation with acceleration in mind. JAX can be used for high-performance numerical and large-scale machine learning computations. The `vbjax`, which is the JAX-based version of TVB, maps the numerical calculations of the original TVB to the GPU. Additionally, the `vbjax` uses an MLP for local dynamic evaluation.

The main idea behind the acceleration performed in `vcjax` is batching many simulation instances and calculating them all at the same time. As mentioned in Section 2.2, large amount of simulations must be performed in order to fit the whole-brain model into the data taken from the patient's brain. All of these simulations are only different in the parameter values of the brain model (the values of the weight matrix), and share all other aspects of the simulation. In other words, this means that all of the simulations running at the same time share the same control sequence in their calculation, and only the values used in the calculations are different. This allows for many simultaneous simulations to be executed on the GPU at the same time, increasing the performance of the system.

3

Design

In this chapter the design that is implemented in the Versal Adaptive SoC is discussed. Although many design candidates were considered and analyzed to find the best fit for the Versal SoC, the design described in this chapter was found to be the most suitable for the device.

3.1. The Versal Adaptive SoC Design

The Versal Adaptive SoC design is a dataflow-style architecture that moves the data in an efficient way. In this design, we divide the system into 2 different parts: 1) MLP evaluation and the FE Solver, and 2) Coupling calculations. For the first part, we introduce the *MLPFE* engines, that are responsible for evaluation of the MLP and performing the Forward Euler to calculate the next timestep value. For the second part of the problem we introduce *CC* engines, are responsible for the calculation of the couplings for the centers. The details of the design are as follows:

1. Each center is assigned to a single MLPFE engine that is responsible for its MLP evaluation in addition to calculation of its next timestep value using Forward Euler method. If there are E MLPFE engines, each engine is assigned $\frac{N}{E}$ centers. For MLP calculations, the previous step state variables and the neural network parameters are needed. The values for the centers that are assigned to an MLPFE engine are stored on the memory block dedicated to the engine. The MLP calculation for each center does not have any dependency on other centers' variables.
2. The general coupling calculation process is challenging in terms of memory needs and access patterns. The idea of this design for addressing this challenge is to group together the connectivities with the same delay. In other words, this design uses the fact that we can rewrite the coupling calculation formula as shown in Equations 3.1 and 3.2. The C_{in} values are coupling inputs for center number i in the delay range n. This design suggests that these values can be calculated in parallel.

$$C_i(t) = K_{post}\left(\sum_{j=1}^{N} W_{ij} K_{pre}\left(X_j(t-d_{ij}), X_i(t)\right)\right) = K_{post}\left(\sum_{n=1}^{CCE_N} C_{in}(t)\right) \tag{3.1}$$

$$C_{in}(t) = \sum_{j=1}^{N} W_{ij} K_{pre}\left(X_j(t-d_{ij}), X_i(t)\right) \textbf{ For } d_{ij} \in [D_{n-1}, D_n) \tag{3.2}$$

3. There are E CC engines in this design, and the engines are chained together in a way that engine #1 can send data to engine #2, and engine #2 can send data to engine #3 and so on. Each engine calculates the couplings associated with a set of delays in the matrix D. We take all the unique values in D (with considering the sparsity pattern of matrix W), sort them, and divide those values between the CC engines. At each timestep, all the calculated state variables enter the CC engine #1. Each state variable might or might not contribute to a connectivity with the delay that engine

#1 is responsible for. If it is, the coupling value is calculated in the engine with that value. As the simulation advances, the state variables are passed from engine to engine until they are no longer needed for coupling calculation, which is the moment they are sent to the off-chip memory. This chain of CC engines, in terms of memory, can be seen as a very big shift register.

4. Let's look at an example. Imagine there are 3 CC engines. Engine #1 is responsible for delays from 0 to 20 timestep, engine #2 for delays of 20 to 50, and engine #3 for delays of 50 to 100. At a certain timestep t_0, the state variables $\vec{X}(t_0)$ for all nodes are calculated and they enter engine #1. From timestep t_0 to $t_0 + 20$, the state variables $\vec{X}(t_0)$ stay in engine 1 since they are less than 20 timesteps old. Any connectivity that contribute to the coupling input of any node i and has a delay of less than 20 is calculated in engine #1 at the right timestep. When the simulation reaches $t_0 + 20$, the values $\vec{X}(t_0)$ are transferred from engine #1 to engine #2. From $t_0 + 20$ until $t_0 + 50$, the connectivities that $\vec{X}(t_0)$ contribute to and have a delay of 20 to 50 are calculated at the right timestep. This trend continues until the simulations reaches the timestep of $t_0 + 100$, where no value of $\vec{X}(t_0)$ no longer contributes to any connectivity and they are sent to the off-chip memory.
5. While the couplings of each timestep are calculated in these engines, the MLPFE engines evaluate the local dynamic of all the nodes and wait for the coupling inputs from the CC engines. After receiving the calculated coupling values, the MLPFE engines then calculate the next step of the state variables.
6. Optionally, bypass lanes can be implemented between non-adjacent engines, and from engines to the off-chip memory. These bypass lanes can prevent data that are not needed in a certain engine not to be stored in that engine or data that are no longer needed not to be stored on the on-chip memory anymore.

The AI Engines of the Versal SoC are designed for dataflow-style processing. The MLPFE engine described in this design is a good candidate to reside in the AI Engines. Additionally, the coupling calculation in this design has a dataflow-style, and it is suited for implementation on both the PL or the AIE. Figure 3.1 shows the general block diagram of this design.

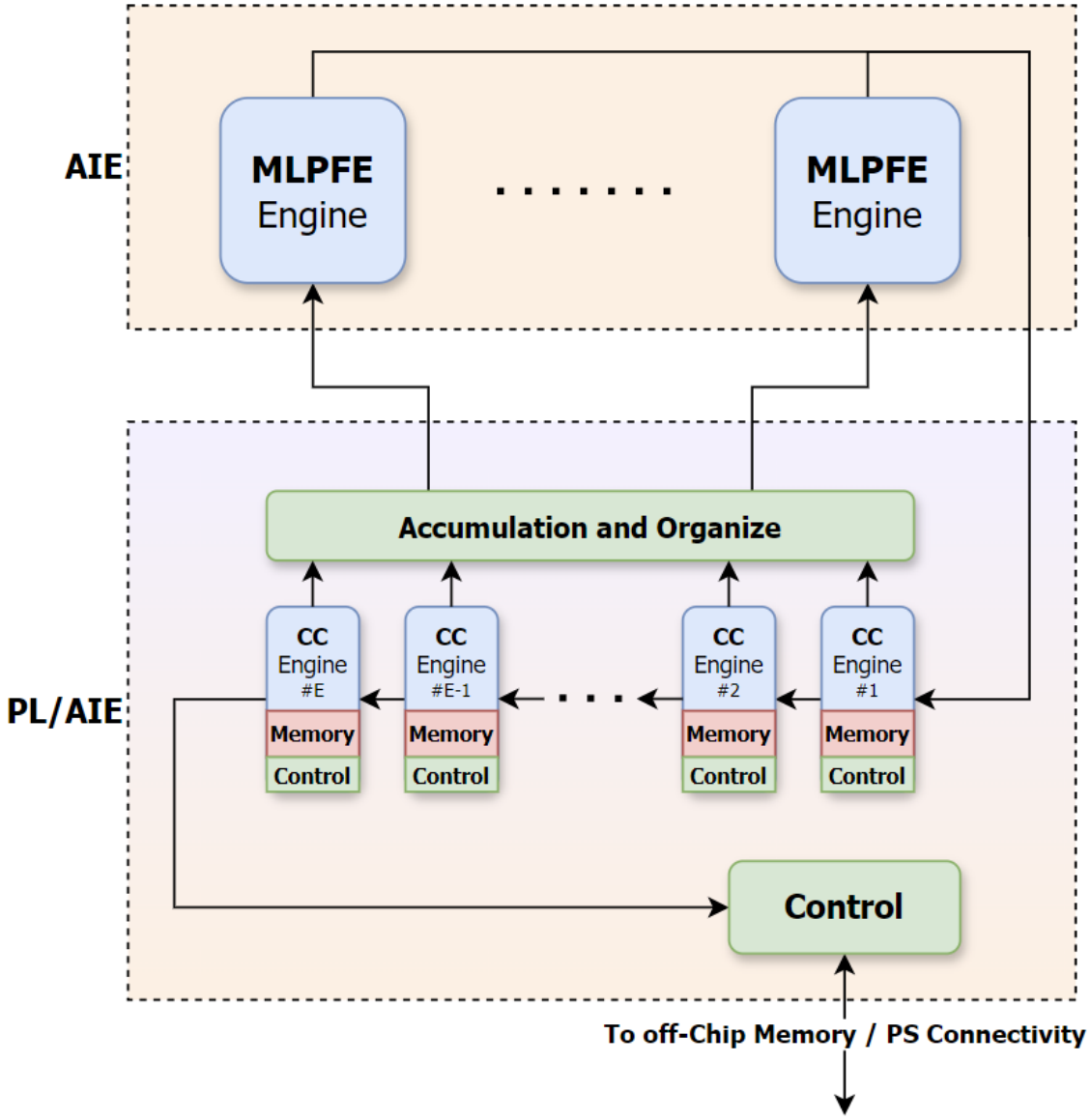


Figure 3.1: The Versal Adaptive SoC Design Block Diagram

4

Implementation

In this chapter, we dive into the implementation details of the Versal Adaptive SoC system. The implemented design is shows in Figure 3.1, and has the option to implement the CC engines in the AIE or the PL. We implemented both versions of this design: 1) A system which all of its components are implemented in the AIE (AIE-Only), and 2) A system which the coupling calculation part of it is implemented in the PL (Heterogeneous). We anticipate the Heterogeneous implementation to have a better performance compared to the AIE-Only system. However, the AIE-Only system benefits from the fast development cycle of the AI Engines, in addition to the usability of it in devices that have the AMD XDNA™ Architecture [2]. This makes the AIE-Only version of the system not limited to the Versal SoC, and brings the acceleration of whole-brain neural-mass models to a wider range of devices.

4.1. AIE-Only System

As mentioned in Section 3.1, the implemented design uses dataflow-style coupling calculations. This makes this process to be appropriate to be mapped to the AIE of the Versal SoC.

The MLPFE engines are kernels that have two input and two output streams, in addition to runtime parameters. The two input streams are for the initial states and the coupling values, and the output streams are for the state variables that go to the first CC engine and all of the state variables that go to the off-chip memory. MLPFE engines in this system are working in parallel and independently from each other. However, their outputs need to be grouped together to be sent to the first CC engine and also the off-chip memory. After some trials with a concatenation tree mechanism that joins the output streams of the MLPFE engines, using the *Packet Switching* construct of the AIE proved to be much more faster.

The CC engines, as mentioned in Section 3, are connected to be able to pass the history of the state variables to each other. Additionally, as shown in Equaiton 3.1, the calculated values in each of the CC engines need to be added together and sent to the MLPFE engines. To do so, we experimented with reduction trees and packet switching, but using the wide *Cascade Stream* connections in the AIE performed the best in adding all the outputs together.

In addition to the cascade stream, a smarter mechanism regarding data availability was used in this implementation to improve its performance. At each timestep of the simulation, the data needed for the all the CC engines except for the first one to calculate the coupling input for the next timestep is already calculated, and it is either residing within the CC engine itself or the engine before it. This data can be passed without waiting for the previous timestep to be finished. However, the first engine has to wait for the input from the MLPFE engines to be able to calculate the couplings.

Figure 4.1 shows the block diagram of the AIE-Only implementation.

Some details regarding this implementation are as follows:

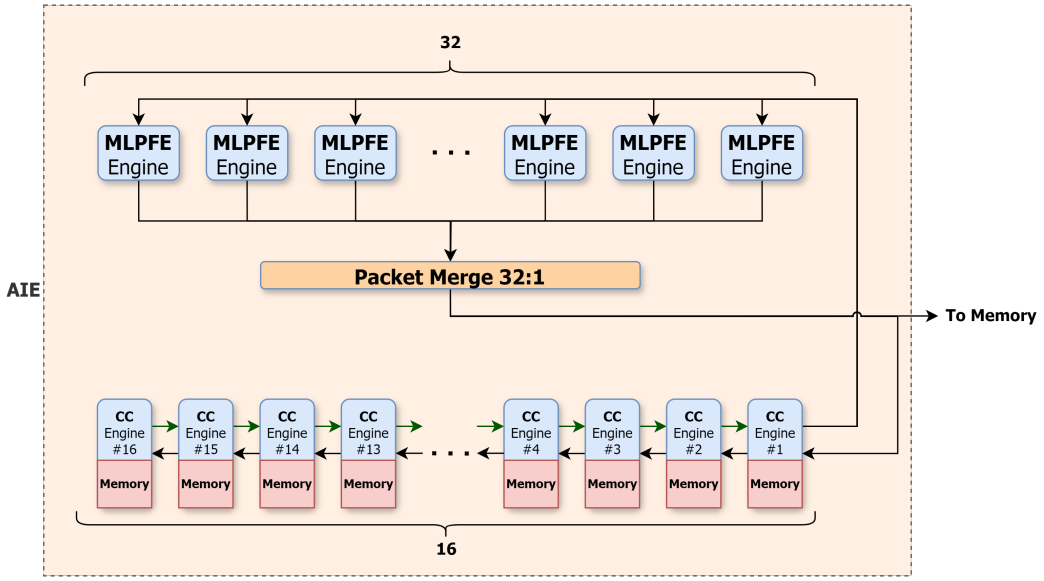


Figure 4.1: Block Diagram of the Second Improved Implementation of the AIE-Only System

1. The calculation part of the MLPFE and the CC engines happens more or less in parallel which helps improving the performance of the system.
2. A cascade stream chain (the green arrows in Figure 4.1) starting from the last CC engine is responsible for moving the accumulated coupling to the next engine. Each CC engine upon receiving data from the cascade stream, adds its own calculated couplings to the received data and outputs the result to the next engine.
3. With this design, the coupling calculation processes of the CC engines, the coupling data transfer, and the accumulation of the coupling data are overlapping to a good extent. Additionally, this implementation provides good scalability if we want to increase the number of CC engines, something that was challenging in tree-based or packet switching reduction mechanisms.

In order to further improve the performance of this system, more CC engines can be added to the implementation. However, adding more CC engines just to one engine-chain proved to be not that beneficial. A better and more efficient way to add more CC engines is to have multiple engine-chains that run in parallel such as shown in Figure 4.2.

1. The centers are divided into 4 groups, and each group has its own MLPFE engines and CC engine chain. These groups are working completely independently from each other, apart from the part where the output of all of the MLPFE engines enter the first CC engine of all of the chains of the system.
2. The coupling calculation workload is divided into 4 CC engine chains. However it is possible that the chains don't have equal workload assigned to them. This is resolved to some extent by re-organizing the centers in a way that the CC engine chains have as close of computation workload as possible to each other.

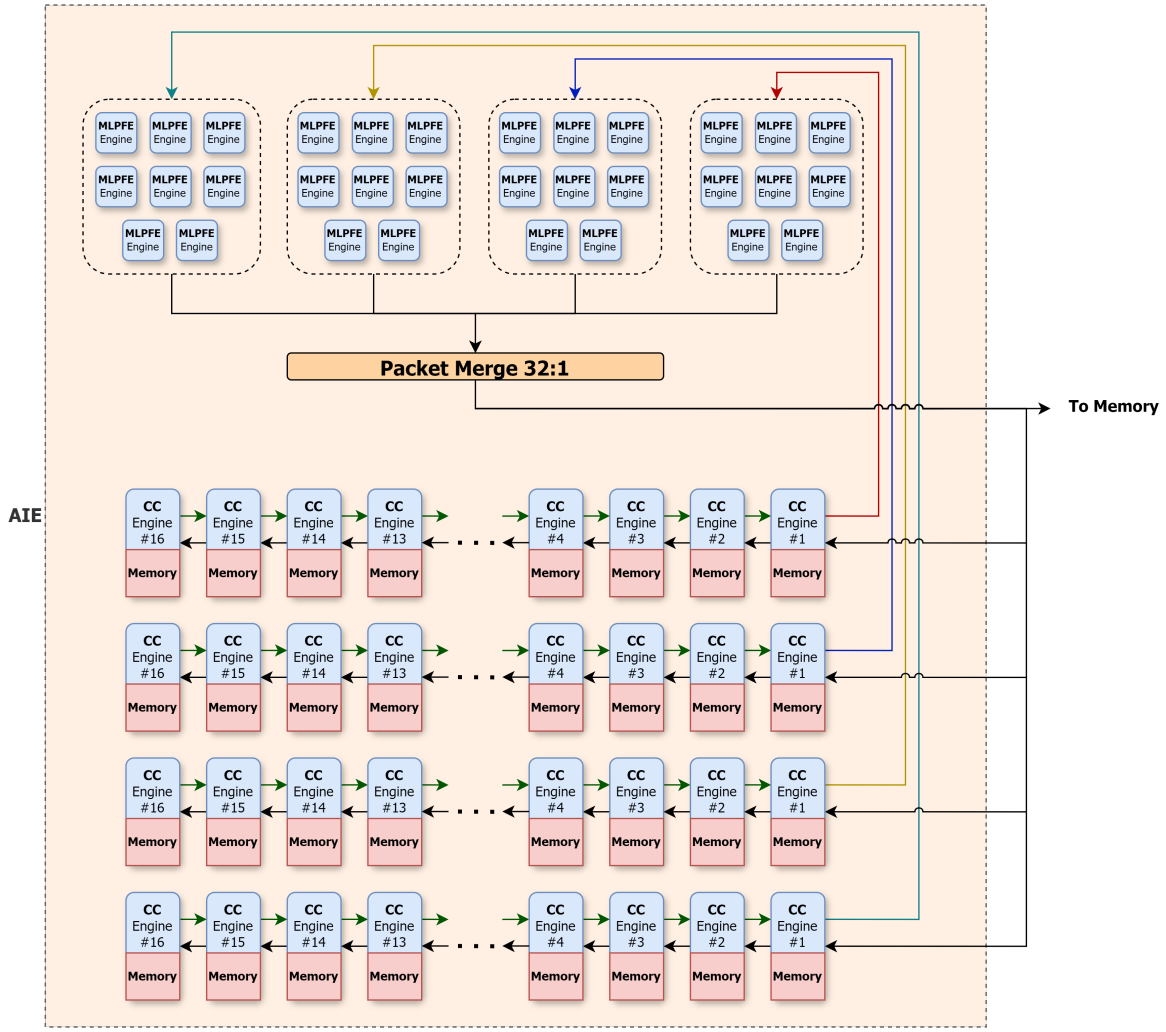


Figure 4.2: Block Diagram of the Final Implementation of the AIE-Only System

4.2. Heterogeneous System

In this section, we discuss the implementation details of the Heterogeneous system. As mentioned in Section 3.1, in the Heterogeneous system the CC engines are implemented in the PL where the MLPFE engines are residing in the AIE, benefiting from the heterogeneous characteristics of the Versal platform. In this implementation, the CC engines and other blocks in the PL were implemented using Vitis HLS which could provide resource- and performance-wise efficient implementation, with a fast development cycle.

Figure 4.3 shows the general block diagram of the Heterogeneous system.

As can be seen in Figure 4.3, the CC engines in the PL calculate the couplings for each center from a certain delay range, send the calculated values to an addition tree which then feeds the results into the Data Distributor block. The Data Distributor block, based on how the centers are divided among the MLPFE engines, puts the coupling results into the appropriate coupling buffer of the engines. Each MLPFE engines reads the coupling input values from their corresponding coupling buffer, calculates the next step state variables of the centers they are responsible for, and puts the newly calculated values into its state variable buffer. The Data Gatherer block then reads the new state variable values from all the buffers, and sends them into the first CC engine in addition to the off-chip memory.

In the implementation of the Heterogeneous system, the MLPFE engines are vectorized and perform

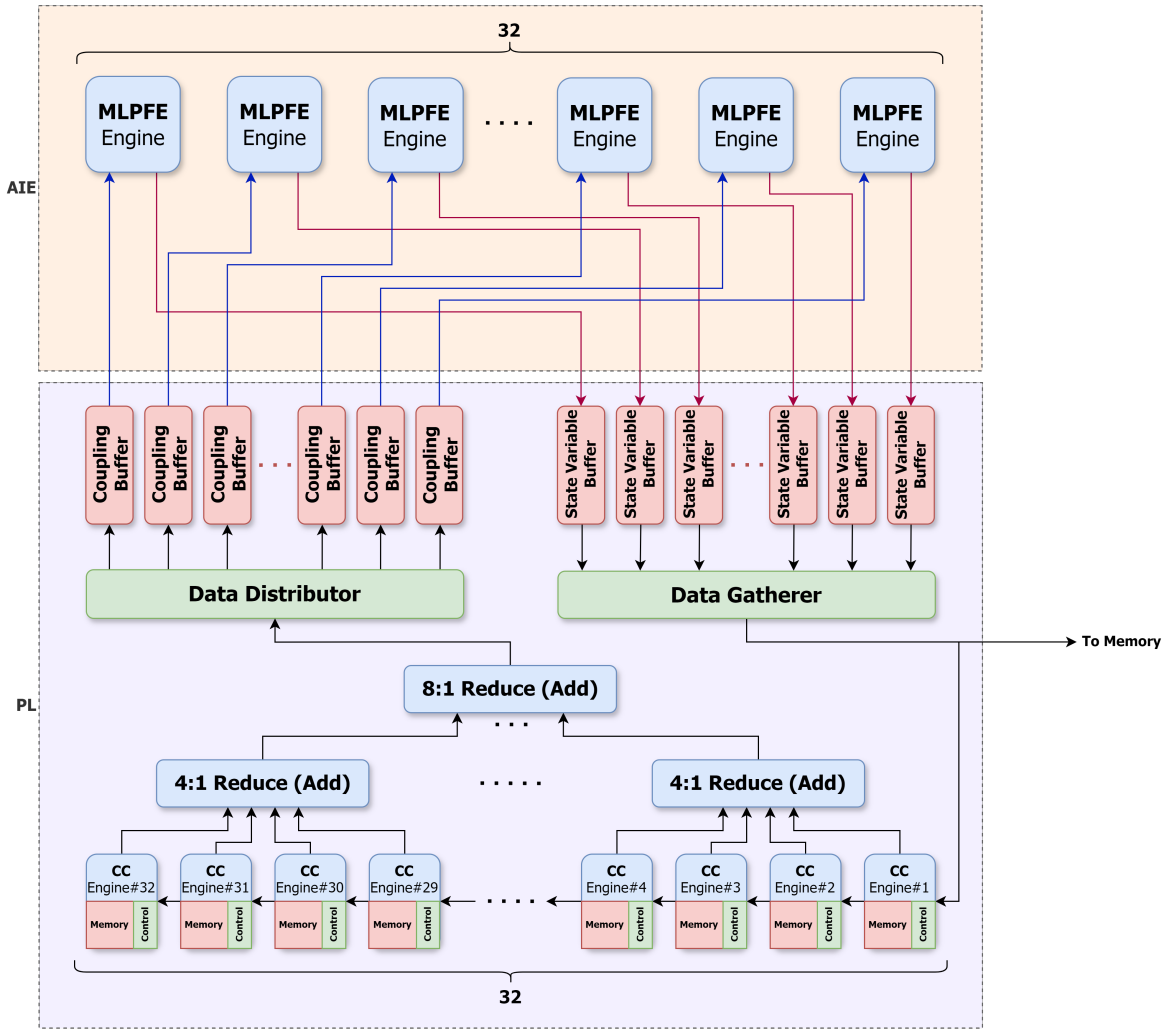


Figure 4.3: Block Diagram of the Implementation of the Heterogeneous System

16 single-precision floating point operations at the same time. This vectorization provides a speedup of around $8\times$ for the MLPFE engines compared to the non-verctorized implementation of the engines.

As mentioned earlier, the CC engines reside in the PL. Each engine is assigned an URAM chunk to store the state variables history the connectivity data. The rest of the memory resources, namely the BRAMs, of the PL are used for buffers in between the connections, in addition to some slack to allow for feasible FPGA place and routing.

Each CC engine is implemented as a Finite State Machine (FSM), with 3 important states: 1) `INIT`, 2) `CALC_TX`, and 3) `RECEIVE` shown in Figure 4.4. In the `INIT` state, the engine reads the connectivity and initial states data from the off-chip memory, and when it is done it goes to the `CALC_TX` state. In this state, the engine calculates the coupling values for each centers and sends it to the 4:1 Reduce block. Additionally, in this state the engine sends the oldest state variables of each center to the next CC engine. It is worth noting that there are buffers in between all the connections to prevent stalls happening in the transmitter side. After the engine calculates and sends all the coupling and state variables values, it enters the `RECEIVE` state where it reads the new state variables from the input buffer and writes it to its local memory. After all the values are read, the engine checks whether it has reached the end of the simulation or not, and if it is the end of the simulation it would go to the `INIT` state and if the simulation is not done, it would enter the `CALC_TX` state to perform the calculations for the next timestep.

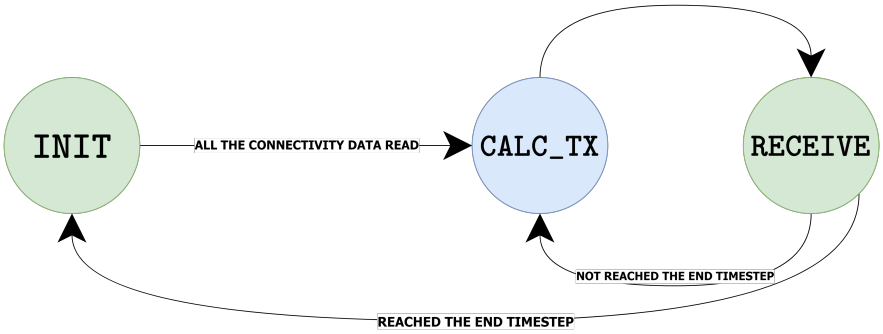


Figure 4.4: CC Engines FSM

All of the states of the CC engine FSM is implemented as a dataflow pipelines. The pipeline of the `INIT` and `RECEIVE` states are fairly straightforward, where they read data from either off-chip memory or a buffer and write them to the local memory. On the other hand, the pipeline of the `CALC_TX` state, as shown in Figure 4.5, is more complicated. The operations seen in Figure 4.5 is performed for each calculation point of the simulation. The calculation points are the points in the weight matrix (W) where the value is not zero. Each calculation point's data (which collectively is the connectivity data), including its row, column, and delay value are stored in separate memory blocks within each CC engine. The addition of none, one, or many of the results of these calculation points would be the coupling input for each center within a CC engine.

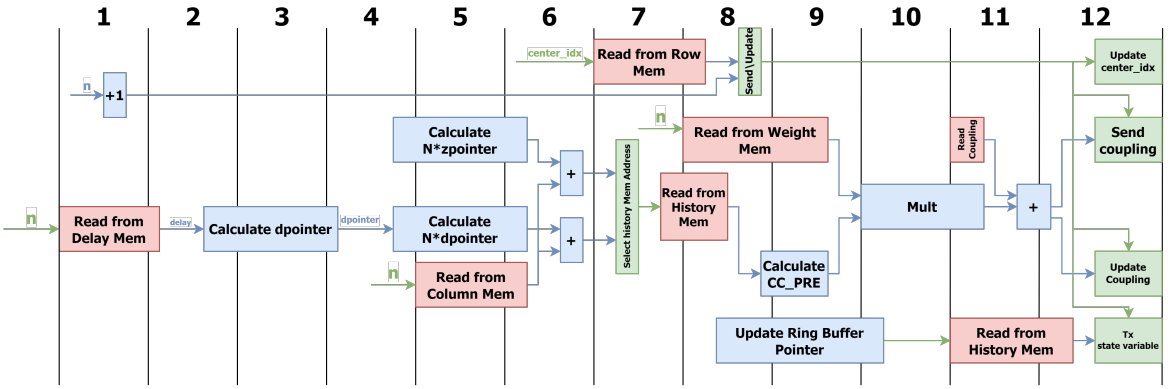


Figure 4.5: CC Engines `CALC_TX` State Pipeline

The details of the stages of the pipeline are as follows:

1. In this stage, the `delay` value of the calculation point is read from the Delay memory. Additionally, the incremented value of `n`, which is the index of the calculation point, is calculated for control purposes.
2. In this stage, the read process of `delay` is finished and it is used to start the calculation of the pointer (`dpointer`) to the ring buffer that holds the history values (History memory).
3. The `dpointer` calculation continues in this stage.
4. With `dpointer` calculated, the calculation of the possible History memory addresses begins. Theses possible addresses are the delayed value of a center which is based on `dpointer`, or it is the most recent value of a center which uses `zpointer`. (`zpointer` is the pointer to the current head of the state variable ring buffer is used.) Additionally, the `col` value is read from the Column memory to be used in later stages for address calculation.
5. The calculations and the accessing process continues in this stage.
6. With the `col` value read from the memory, it is added to the base addresses calculated in the past 2 stages. The results of these 2 additions enter the next stage of the pipeline.

7. In this stage, based on the timestep that the simulation is in, the appropriate History memory address is selected and reading process from the History memory starts. Additionally, `row` value is read from the Row memory for control purposes. The address of the Row memory is `center_idx`, which is the index of the current center that its coupling input is being calculated.
8. In this stage, the value read from the History memory enters the `CC_PRE` calculations which is the pre-synapse function (K_{Pre}) mentioned in Section 2.3. Additionally, the `row` and the `n+1` values are used to determine whether there should be a transmission at the last stage of the pipeline. Furthermore, the process of calculating the address to the oldest state variable in the ring buffer (the value that needs to be shifted out) starts. The reading process from the Weight memory also starts in this stage.
9. In this stage, the output of the `CC_PRE` calculation and the `weight` value read from the Weight memory enter the multiplication process.
10. The pipelined multiplication process continues in this stage. The calculation of the address of the oldest state variables in the History memory finished as well.
11. With the the multiplication finished, its result is added to the stored `coupling` variable. This variable holds the value of the coupling input of the center `center_idx`. Additionally, the process of reading the oldest value in the History buffer starts in this stage.
12. In the last stage of the pipeline, based on the control signal calculated in stage 8, the value of `coupling` is transmitted and updated, the oldest state variable of center `center_idx` is transmitted, and also the value of `center_idx` is updated.

The `CALC_TX` state in the CC engines are fully pipelined, and the described pipeline run with minimum interval in between. This means if a center has to calculate 100 calculations points, the whole process would take 102 clock cycles. In addition to the CC engines, the reduction blocks (4:1 and 8:1 addition) shown in Figure 4.3 are also implemented as dataflow pipelines. This is also true for Data Distributor and Gatherer blocks.

4.3. Deployment

In order to run the system in the Versal SoC, the simulation information is needed. A pre-processing script, written in Python takes in the simulation parameters, such as weights, delays, signal propagation speed, MLP parameters, or timestep size, and generate configuration files based on these inputs and the architecture of the target system (for example number of engines). These configuration files are written to a certain location of the memory before starting the simulation, and the system upon booting up reads the information from this address of the memory and initializes the engines based on that.

5 Results

In this chapter, we present the results of the state of the art related works, in addition to the AIE-only and Heterogeneous systems. The performance of the systems are discussed and compared, and the reasons behind the differences in the results are analyzed.

The Neural-Mass Model used for benchmarking the systems developed by us use the connectivity data from The Virtual Brain (TVB) [22]. The TVB dataset contains three different models with different number of centers and connectivity data shown in Table Table 5.1. As described in Section 2.2, the NMM divides the brain into a number of centers which are connected to each other in a certain way, and each have a number of state variables and local dynamics. The TVB dataset defines the number of centers (in more detail, the regions of the brain) and how they are connected to each other. For a NMM simulation, we also have to specify the number of state variables in addition to the local dynamics (the structure and parameters of the MLP). For the results presented in this chapter, we use the state variable number of 2 ($M = 2$) and a MLP architecture with 1 hidden layer of 64 neurons ($H = 1$ and $L = 64$). The MLP was trained to approximate the local dynamics of a Simple 2D Oscillator model [14], and the single hidden layer was sufficient to accurately calculate the local dynamics for this model [3]. It is worth noting that because of the large size of the TVB998 model, a subset of this dataset with fewer centers (for example 600 centers - TVB600) is mostly used for benchmarking. This subset is not accurate in terms of neuroscience validity, but is used for performance measurement purposes only. Additionally, all of the reported numbers in this chapter are for a simulation running for 3000 timesteps with 0.05 step size, unless specified.

	TVB76	***TVB192***	***TVB998***
Number of Centers	76	192	998
Sparsity (%)	72.99	90.42	96.41
Number of Calculation Points	1560	3532	18736

Table 5.1: TVB Dataset

5.1. Related Work

5.1.1. TVB on CPU

Table 5.2 shows the performance results of the original TVB (sequential) system. As mentioned in Section 2.4.1, the sequential system is a C++ port of the back-end algorithm used in the original TVB with MLP being used for the local dynamics (the `tvb_algo_c`). Additionally, this system is running on a machine with a 16-core Intel Core i7-13700KF cpu with $30MB$ of cache running at maximum of $5.4GHz$.

The performance numbers for the Fast TVB and TVB C++ were extracted from [17], which because of the limitation of the Fast TVB tool, only a certain model with 379 centers was tested. Table 5.3 shows the execution time of the Fast TVB and TVB C++ running for 100,000 timesteps and a step size of 0.1. Additionally, these version of TVB don't include MLP for their local dynamics calculations.

Benchmark	Original TVB System (`tvb_algo_c`)
TVB76	101,417.34 μs
TVB192	259,133.89 μs
TVB600	848,660.03 μs
TVB998	1,510,870.00 μs

Table 5.2: `tvb_algo_c` Performance

Benchamrk	Fast TVB	TVB C++
Reduced Wong Wang with 379 centers	5.1 s	6.4 s

Table 5.3: Optimized TVB on CPU Performance

5.1.2. TVB on GPU

Tables 5.4 shows the performance results of the GPU version of TVB (`vbjax`) when running a model for 10,000 timesteps on an Nvidia RTX6000 GPU. The model that `vbjax` is running has the same number of centers as the main benchmarks, but the connections are dense (all-to-all) connections. This doesn't affect the performance of the GPU version of TVB since the `vbjax` does not perform sparse matrix operations. The batch size in Table 5.4 refers to the number of simulation instances the GPU runs at the same time.

Benchmark	Batch Size							
	1	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
76 Centers (Dense)	0.35 s	0.48 s	0.48 s	0.48 s	0.49 s	0.67 s	1.83 s	3.61 s
192 Centers (Dense)	0.38 s	0.49 s	0.51 s	0.53 s	1.05 s	2.34 s	4.53 s	8.96 s
600 Centers (Dense)	0.37 s	0.62 s	0.95 s	2.04 s	3.87 s	7.52 s	14.1 s	27.4 s
998 Centers (Dense)	0.45 s	0.93 s	1.8 s	3.35 s	6.31 s	12.1 s	23.2 s	45.5 s

Table 5.4: TVB on GPU (`vbjax`) Performance

5.2. AIE-Only System

Table 5.5 shows the performance results of all of the implementations of the AIE-only system.

# CC Engines	TVB76	TVB192	TVB600	TVB998
64 (16/Chain)	15,766.13 μs	55,016.46 μs	-	-
128 (32/Chain)	15,351.98 μs	40,043.77 μs	-	-
160 (40/Chain)	15,204.91 μs	39,682.47 μs	205,046.73 μs	-

Table 5.5: AIE-Only Systems Performance

5.3. Heterogeneous System

Table 5.6 shows the performance results of the hybrid system with 32 CC and 64 MLPFE engines.

Benchmark	Heterogeneous System (32 CC / 64 MLPFE)
TVB76	4,132.02 μs
TVB192	7,522.42 μs
TVB600	21,099.30 μs
TVB998	33,828.95 μs

Table 5.6: Heterogeneous System Performance

5.4. Discussion

In this section we talk about and compare the performance results from the different systems mentioned earlier. First, the results of the AIE-only and Heterogeneous systems are compared to the CPU versions of the TVB. Next, an extensive comparison between the final Veral Adaptvice SoC system (the Heterogeneous system) and the GPU version of the TVB is presented.

5.4.1. AIE-Only and Heterogeneous Systems vs TVB on CPU

As can be seen from the numbers presented earlier in this chapter in Tables 5.2, 5.3, 5.5, and 5.6, both the AIE-Only and Heterogeneous systems outperform all of the CPU versions of TVB (Original, Fast, and C++ TVBs). Although the CPU runs on a much faster clock frequency compared to the Versal SoC, the custom, application-specific dataflow design of the AIE-Only and Heterogeneous systems made them to perform better in terms of speed. Additionally, in terms of power consumption, the Versal systems consume almost half the power of the CPU, making them much more energy efficient as well. With AIE-Only system outperforming a high-end CPU in terms of speed, power consumption, and energy efficiency; devices that benefit from the AMD XDNA™ Architecture can use this system for accelerating the brain modeling applications.

5.4.2. Heterogeneous System vs TVB on GPU

In order to compare the Versal SoC system against the GPU version of TVB, we take a look at their throughput, latency, and power (energy).

Throughput

When we talk about the throughput of the system, we refer to how many simulation timesteps (or iterations) can the system perform in a second ($\frac{iters}{s}$). Figures 5.1 and 5.2 show the throughput performance results of the Heterogeneous system and the GPU version of TVB respectively. The dashed lines in Figure 5.2 are the throughput results of the Heterogeneous system and are used for comparison.

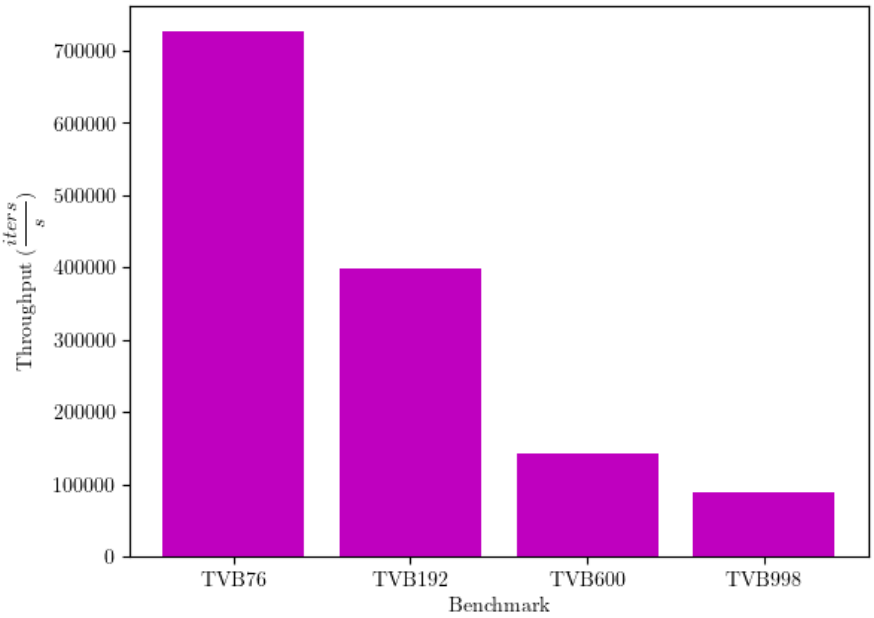
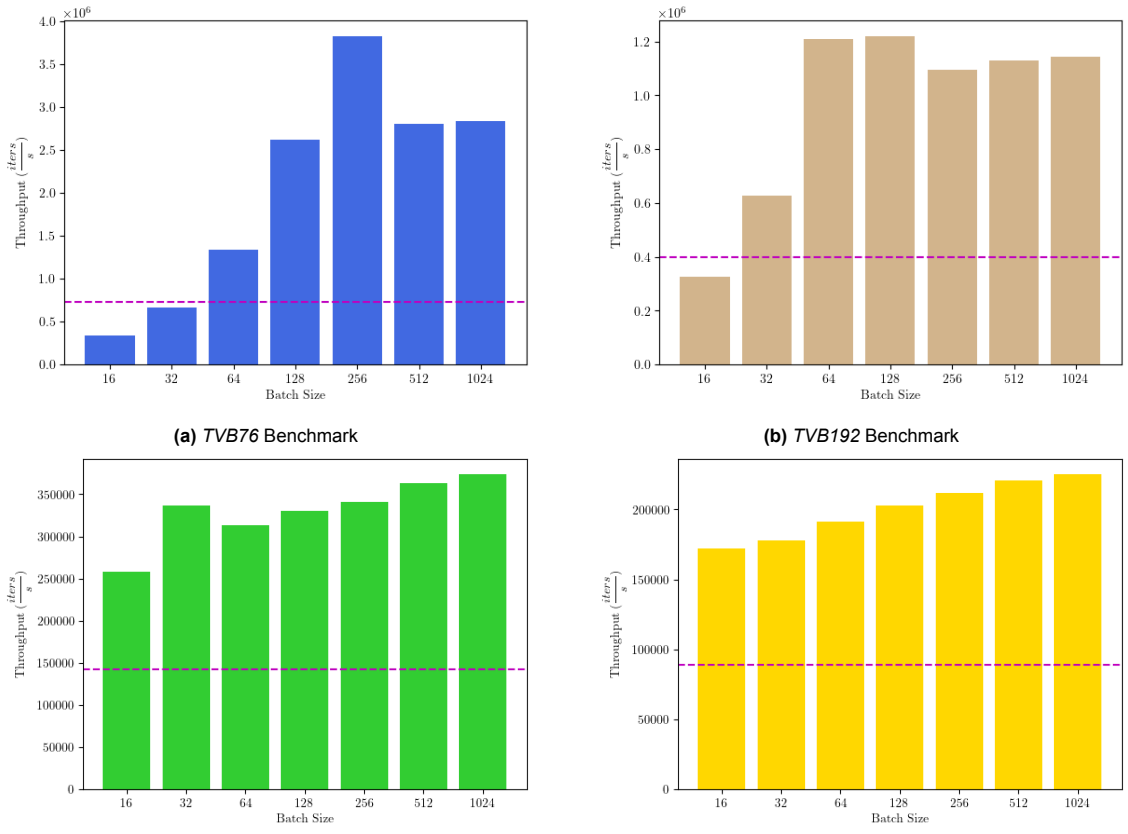


Figure 5.1: Throughput Performance of the Heterogeneous System

Firstly, as can be seen from both Figures 5.1 and 5.2, the throughput of the systems decrease as the number of centers increases. This is expected as with more centers in the model, more calculations are needed for each timestep. However, with smaller batch sizes in the GPU, the throughput for all of the benchmarks remains more or less the same. This shows that the GPU can be saturated with larger



(a) *TVB76* Benchmark

(b) *TVB192* Benchmark

(c) *TVB600* Benchmark

(d) *TVB998* Benchmark

Figure 5.2: Throughput Performance of the TVB on GPU

batch size or larger models.

Additionally, when looking at Figure 5.2, we can see that the GPU version of TVB (at its most suitable batch size) performs around $2.5 - 7\times$ the Versal SoC Heterogeneous system in terms of throughput. This performance gap is mainly due to the batching of the simulations performed by the GPU version of TVB. Furthermore, the batching of simulation instances helps with hiding the overhead that comes with the host-GPU communication.

Currently our Heterogeneous system does not support running multiple simulations at the same time. However, if we want better throughput performance for the Heterogeneous system by incorporating simulation batching, the following measures can be done:

1. The on-chip memory assigned to each of the simulation instances is reduced, allowing for more concurrent independent coupling calculation processes. The model fitting process performs many simulation with the same connectivity structure, meaning that the sparsity pattern and delay values of the connectivity matrices are the same, and only the values of the weight matrix is changed from simulation to simulation. This implies that the simulation instances can share the structural data, only requiring separate memory for the weight and history values.
2. As the reduction of the on-chip memory assigned to each simulation causes limitation on the largest model that the system can handle, off-chip memory can be used to compensate this memory shortage. Incorporating the off-chip memory can cause performance flop in the simulations, but as we scale-out by running the simulations at the same time, the throughput performance increases. Additionally, smart measures such as pre-fetching or exploiting the delay of accessing the off-chip memory for the delay inside the simulation can be used to make up for some of the performance loss caused by incorporating the off-chip memory.

Latency

The first step of building a patient-specific whole-brain model is to perform the model fitting. In this step, as mentioned earlier, we require to perform large number of simulations to train the parameters of the model. However, when the system is done with its training, it can be used in many settings including real-time applications (such as the ones in [23, 30]). In this case, what matters is the how fast the system performs a single simulation, or its latency. Figure 5.3 shows the latency performance of the Heterogeneous system and the GPU version of TVB when running different models for 3000 timesteps. Since the execution time of the GPU system also includes the data transfer and some initialization' time, the numbers shown in Figure 5.3 is an estimation of the actual computation time of the GPU version of TVB. In order to get this estimation, we ran the simulation for 100000 and 220000 timesteps, calculated the different between their execution times, and normalized that number to 3000 timesteps by dividing the difference by 40.

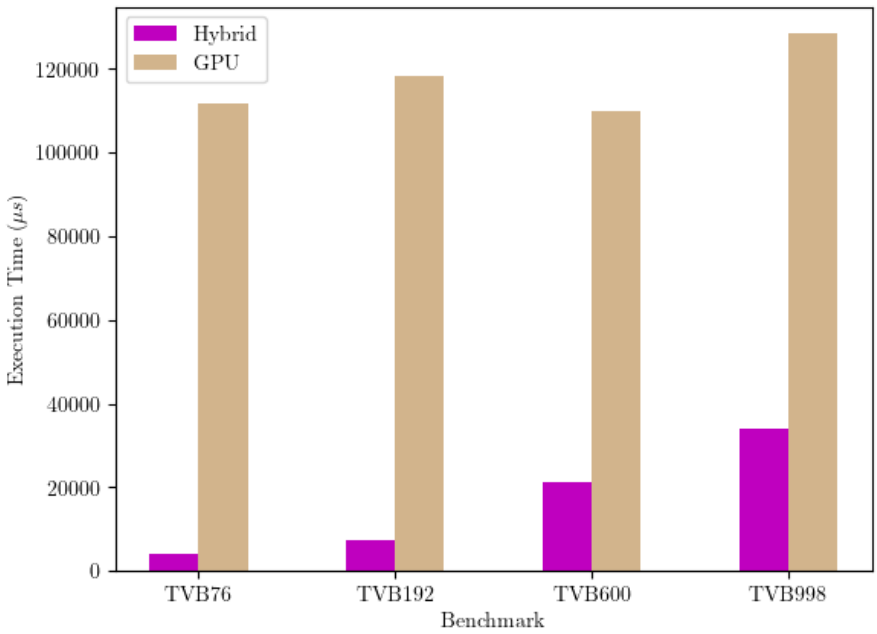


Figure 5.3: Latency Performance of the Heterogeneous and TVB on GPU Systems

As can be seen in Figure 5.3, the Heterogeneous system on the Versal SoC performs on average around $20\times$ better compared to the GPU system. The better performance of the Heterogeneous system in terms of latency shows the effect of the custom memory hierarchy in the PL and the effective implementation of the MLPFE and CC engines. This lower latency of the Versal SoC system makes it much more suitable for the edge's real-time and streaming applications where the whole-brain neural-mass modeling is part of a time-sensitive critical path.

Power, Energy

In terms of power, the Quadro RTX6000 used by is for benchmarking the GPU version of TVB operates at a maximum of 260 Watts. Table 5.7 shows the approximate power consumption of the GPU when running each of the benchmarks. These numbers were taken by monitoring the power consumption of the GPU while running the simulation using `nvidia-smi` command line tool. Based on the reported numbers in Table 5.7, it is fair to assume an average power consumption of around **162 and 236 Watts** for single and batched simulation cases respectively.

The power consumption of the Versal SoC heavily depends on the utilization of the device. AMD provides a power estimation tool as part of Vivado that can be used to estimate the power consumption of the device based on different utilization parameters of the design. As shown in Figure 5.4, the power consumption value of the Heterogeneous system lies around **43.63 Watts**. Since the entirety of

Benchmark	Single Simulation	Batched Simulation (Batch Size)
TVB76	119 W	230 W (256)
TVB192	137 W	224 W (128)
TVB600	172 W	243 W (1024)
TVB998	219 W	246 W (1024)
Average	162 W	236 W

Table 5.7: TVB on GPU Power Consumption

the system is used for any of the simulations regardless of their size, the power consumption value is independent on the input of the system.

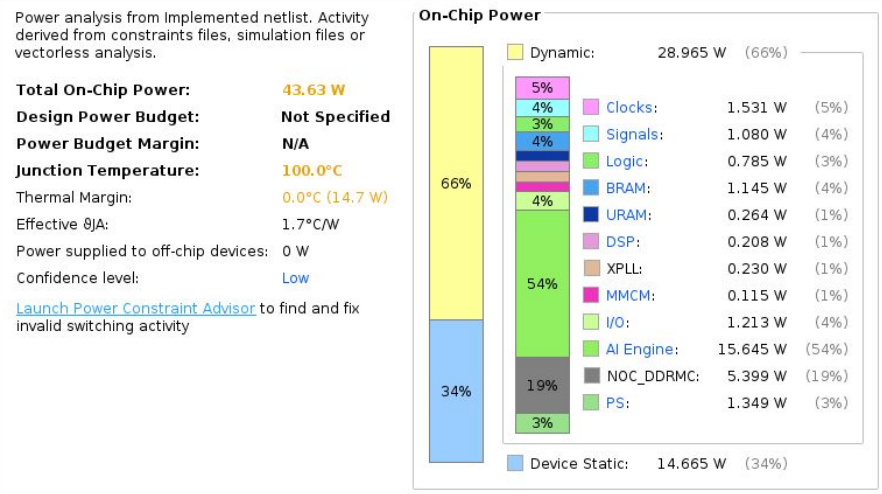


Figure 5.4: Power Consumption and Categorization of The Heterogeneous System

Although the Heterogeneous system consumes less power, in order to compare the power efficiency of the Versal SoC and the GPU systems, we take a look at the *Energy* and *Performance per Watt* values of these systems when running different models. We use the single and batched simulations for energy and performance per Watt calculations respectively. Additionally, the numbers reported in Figure 5.3 are used for energy consumption calculations. Figures 5.5 and 5.6 show the energy and performance per Watt results respectively.

As expected, the Heterogeneous system consumes less energy compared to the GPU version of TVB, as shown in Figure 5.5 when running single simulation. The energy consumption difference is around two orders of magnitude, making the Versal SoC system much more energy efficient compared to the GPU system. In terms of performance per Watt, as can be seen in Figure 5.6, the Heterogeneous system has around $2.2\times$ higher performance efficiency in the best case compared to the GPU system. Although the GPU version of TVB performs better in terms of throughput compared to the Heterogeneous system, the much lower power consumption of the Heterogeneous systems makes it more performance efficient.

5.5. Conclusion

In this chapter, the performance results from different versions of TVB on CPU, GPU, and Versal SoC were presented and compared. We saw that how two different versions of Versal Adaptive SoC implementations (AIE-only and Heterogeneous) were benchmarked outperformed the CPU versions of TVB both in terms of speed and power consumption. The final Versal SoC system (the Heterogeneous system) was also compared to the high-performance GPU version of TVB in Section 5.1.2. We showed that the Heterogeneous system on Versal Adaptive SoC performs better in terms of latency

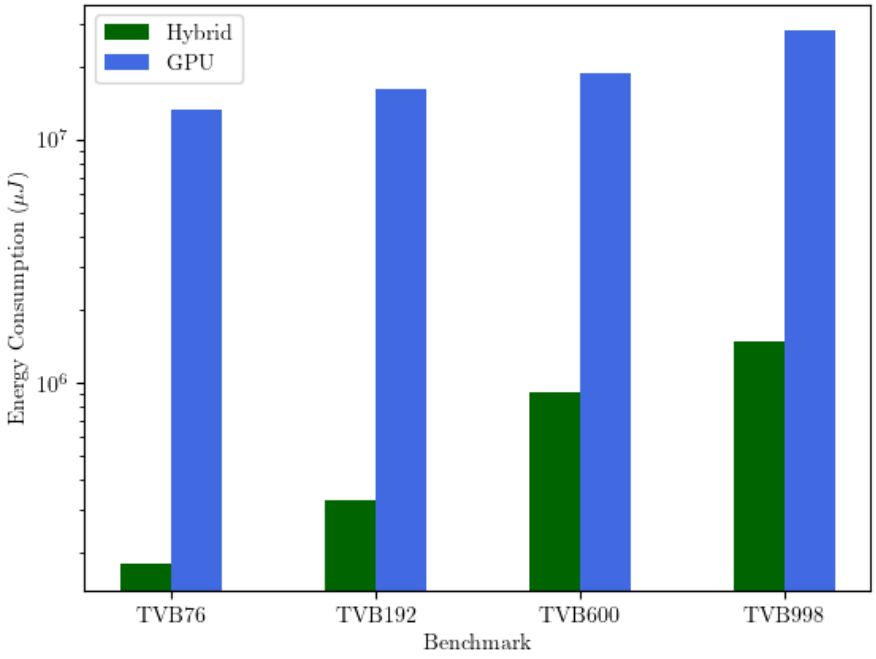


Figure 5.5: Energy Consumption for Heterogeneous and GPU Systems

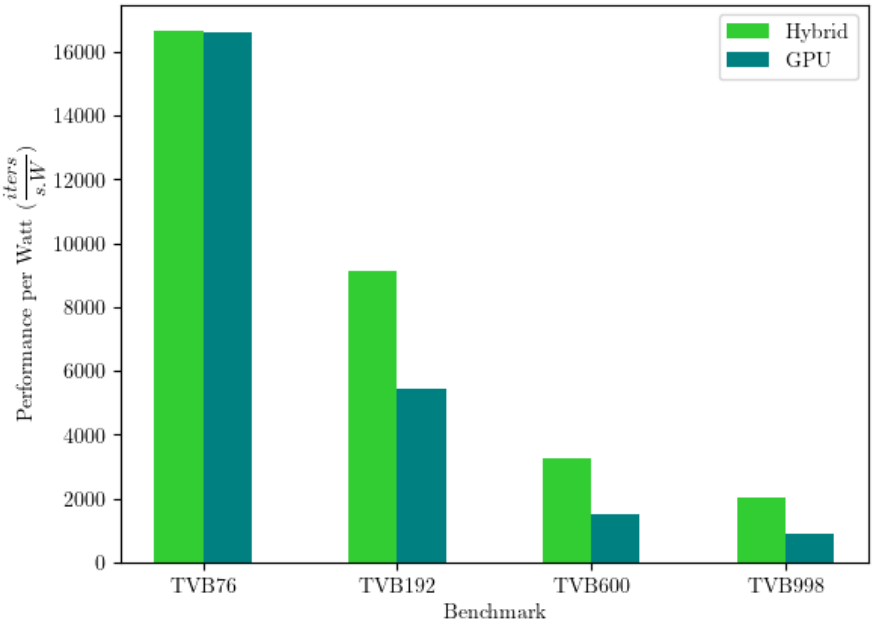


Figure 5.6: Performance per Watt for Heterogeneous and GPU Systems

($20\times$), energy consumption ($100\times$), and performance per Watt ($2\times$) compared to the GPU version of TVB; which makes it a great candidate for model fitting and real-time and low-power applications that require whole-brain network simulations on the edge.

References

[1] Shun□ichi Amari. “Dynamics of pattern formation in lateral-inhibition type neural fields”. In: *Biological Cybernetics* 27 (1977), pp. 77–87. URL: https://api.semanticscholar.org/CorpusID:2811608.

[2] AMD. *AMD XDNA™ Architecture*. 2024. URL: https://www.amd.com/en/technologies/xdna.html (visited on 06/14/2024).

[3] Nina Baldy et al. “Efficient Inference on a Network of Spiking Neurons using Deep Learning”. In: *bioRxiv* (2024). URL: https://api.semanticscholar.org/CorpusID:267312741.

[4] R. L. Beurle. “Properties of a mass of cells capable of regenerating pulses”. In: *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 240 (1956), pp. 55–94. URL: https://api.semanticscholar.org/CorpusID:85372469.

[5] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[6] Tian Qi Chen et al. “Neural Ordinary Differential Equations”. In: *Neural Information Processing Systems*. 2018. URL: https://api.semanticscholar.org/CorpusID:49310446.

[7] Gustavo Deco and Morten L. Kringelbach. “Great Expectations: Using Whole-Brain Computational Connectomics for Understanding Neuropsychiatric Disorders”. In: *Neuron* 84 (2014), pp. 892–905. URL: https://api.semanticscholar.org/CorpusID:12197385.

[8] Gustavo Deco et al. “The Dynamic Brain: From Spiking Neurons to Neural Masses and Cortical Fields”. In: *PLoS Computational Biology* 4 (2008). URL: https://api.semanticscholar.org/CorpusID:494344.

[9] National Academy of Engineering. *Grand Challenges*. 2024. URL: https://www.engineeringchallenges.org/challenges.aspx (visited on 06/21/2024).

[10] Richard FitzHugh. “Impulses and Physiological States in Theoretical Models of Nerve Membrane.” In: *Biophysical journal* 1 6 (1961), pp. 445–66. URL: https://api.semanticscholar.org/CorpusID:18307924.

[11] Hermann Haken. *Brain Dynamics: Synchronization and Activity Patterns in Pulse-Coupled Neural Nets with Delays and Noise*. Google-Books-ID: 8elDAAAAQBAJ. Springer Science & Business Media, Nov. 22, 2006. 249 pp. ISBN: 978-3-540-46284-2.

[12] A. L. Hodgkin and A. F. Huxley. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. In: *The Journal of Physiology* 117.4 (1952), pp. 500–544. DOI: https://doi.org/10.1113/jphysiol.1952.sp004764. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1952.sp004764. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1952.sp004764.

[13] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert L. White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2 (1989), pp. 359–366. URL: https://api.semanticscholar.org/CorpusID:2757547.

[14] E.M. Izhikevich. “Which Model to Use for Cortical Spiking Neurons?” In: *IEEE Transactions on Neural Networks* 15.5 (Sept. 2004), pp. 1063–1070. ISSN: 1045-9227. DOI: 10.1109/TNN.2004.832719. URL: http://ieeexplore.ieee.org/document/1333071/ (visited on 11/23/2023).

[15] Viktor Jirsa and Hermann Haken. “Field Theory of Electromagnetic Brain Activity.” In: *Physical review letters* 77 5 (1996), pp. 960–963. URL: https://api.semanticscholar.org/CorpusID:12810321.

[16] Viktor Jirsa et al. “Personalised virtual brain models in epilepsy”. In: *The Lancet Neurology* 22.5 (May 1, 2023), pp. 443–454. ISSN: 1474-4422. DOI: 10.1016/S1474-4422(23)00008-X. URL: https://www.sciencedirect.com/science/article/pii/S147444222300008X (visited on 06/14/2024).

[17] Ignacio Mart'in et al. "TVB C++: A Fast and Flexible Back-End for The Virtual Brain". In: 2024. URL: https://api.semanticscholar.org/CorpusID:270095206.

[18] Michael Mascagni and Arthur Sherman. "Numerical Methods for Neuronal Modeling". In: 1989. URL: https://api.semanticscholar.org/CorpusID:14962939.

[19] Amirreza Movahedin. *tvb-algo-c*. 2023. URL: https://github.com/AmirrezaMov/tvb-algo-c (visited on 11/30/2023).

[20] Wieslaw Lucjan Nowinski. "Evolution of Human Brain Atlases in Terms of Content, Applications, Functionality, and Availability". In: *Neuroinformatics* 19 (2020), pp. 1–22. URL: https://api.semanticscholar.org/CorpusID:220843511.

[21] Anagh Pathak, Dipanjan Roy, and Arpan Banerjee. "Whole-Brain Network Models: From Physics to Bedside". In: *Frontiers in Computational Neuroscience* 16 (May 26, 2022). Publisher: Frontiers. ISSN: 1662-5188. DOI: 10.3389/fncom.2022.866517. URL: https://www.frontiersin.org/articles/10.3389/fncom.2022.866517 (visited on 06/14/2024).

[22] Leon Paula et al. "The Virtual Brain: a simulator of primate brain network dynamics". In: *Frontiers in Neuroinformatics* 7.10 (2013), pp. 10–11.

[23] Alexander Pei and Barbara G. Shinn-Cunningham. "Closed-Loop Current Stimulation Feedback Control of a Neural Mass Model Using Reservoir Computing". In: *Applied Sciences* (2023). URL: https://api.semanticscholar.org/CorpusID:256174191.

[24] The University of Queensland - Queensland Brain Institute. *Lobes of the brain*. 2024. URL: https://qbi.uq.edu.au/brain/brain-anatomy/lobes-brain (visited on 07/17/2024).

[25] Petra Ritter et al. "The Virtual Brain Integrates Computational Modeling and Multimodal Neuroimaging". In: *Brain connectivity* 3 2 (2013), pp. 121–45. URL: https://api.semanticscholar.org/CorpusID:7680252.

[26] Paula Sanz-Leon et al. "Mathematical framework for large-scale brain network modeling in The Virtual Brain". In: *NeuroImage* 111 (May 2015), pp. 385–430. ISSN: 10538119. DOI: 10.1016/j.neuroimage.2015.01.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S1053811915000051 (visited on 11/29/2023).

[27] Michael Schirner et al. "Brain simulation as a cloud service: The Virtual Brain on EBRAINS". In: *NeuroImage* 251 (2022). URL: https://api.semanticscholar.org/CorpusID:246493232.

[28] Medical News Today. *All you need to know about neurons*. 2024. URL: https://www.medicalnewstoday.com/articles/320289 (visited on 07/18/2024).

[29] Huifang E. Wang et al. "Virtual brain twins: from basic neuroscience to clinical use". In: *National Science Review* 11.5 (May 2024), nwae079. ISSN: 2053-714X. DOI: 10.1093/nsr/nwae079.

[30] Junsong Wang et al. "Suppressing epileptic activity in a neural mass model using a closed-loop proportional-integral controller". In: *Scientific Reports* 6 (2016). URL: https://api.semanticscholar.org/CorpusID:4539967.

[31] HR Wilson and JD Cowan. "Excitatory and inhibitory interactions in localized populations of model neurons". In: *Biophysical journal* 12.1 (Jan. 1972), pp. 1–24. ISSN: 0006-3495. DOI: 10.1016/s0006-3495(72)86068-5. URL: https://europepmc.org/articles/PMC1484078.

[32] Marmaduke Woodman. *tvb-algo*. 2020. URL: https://github.com/maedoc/tvb-algo (visited on 02/20/2020).

[33] Marmaduke Woodman. *vbjax*. 2024. URL: https://github.com/ins-amu/vbjax/tree/main (visited on 06/20/2024).