Heterogeneous Acceleration of Neural-Mass Models towards Digital Brain Twins

Porting The Virtual Brain on the Versal ACAP

Master's Thesis Report

Amirreza Movahedin

Delft University of Technology

TU Delft

Heterogeneous Acceleration of Neural-Mass Models towards Digital Brain Twins

Porting The Virtual Brain on the Versal ACAP

by

Amirreza Movahedin

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday July 15, 2024 at 9:30 AM.

Student number:	5753244	
Project duration:	November 20, 2023 - July 15, 2024	
Thesis committee:	Dr. ir. C. Strydis,	TU Delft, supervisor
	Dr. ir. J.S.S.M. Wong,	TU Delft
	Dr. M. Negrello,	Erasmus MC
	Ir. L. Landsmeer,	TU Delft

Preface

First and foremost, I am grateful to God for his grace and blessing, not only during this thesis but throughout my entire life.

I would like to thank my supervisor Christos Strydis for giving me the opportunity to work on this topic. He was a great motivation and support throughout the time I was working on this thesis. I also would like to thank Mario Negrello, Lennart Landsmeer, and Freddie Renyard from Erasmus MC for their help and guidance. Additionally, I would like to thank Marmaduke Woodman from The Virtual Brain team for his assistance.

Furthermore, I want to thank my friends in Delft. Andrea, Jyo, and Pietro from kitchens 5A and 121; and Alexander, Andrea, Antonio, Christian, Gustavo, Jiacong, Marnix, Raffaele, Riccardo, and Tony from the 10th floor and the amazing movie nights. I found a family away from home with you, and I appreciate you all greatly.

Last but not least, I would like to thank my family and friends back home: My parents, whose presence has been a great support and inspiration for me my entire life, and I am extremely grateful for what they did and do for me; My brother, sister-in-law, and beautiful niece Ayeh who make my life much more beautiful; And my dear friends back home who are and have always been there for me throughout ups and downs of life.

Amirreza Movahedin
Delft, July 2024

Summary

The human brain is arguably the most complex system we know of. For centuries, understanding the brain and the way it works has been of great concern to scientists. In 1952, the first mathematical model of a neuron was developed by Hodgkin and Huxley. This work pioneered the field of computational neuroscience, which studies the organization of the brain and the way it processes information. Modeling and simulation of neurons and small brain regions at the cellular level offer great insights into the details of brain mechanics, however, they are far away from clinical applications.

The Virtual Brain (TVB) is a simulation framework that takes a different approach to brain modeling. TVB reduces complexity at the micro level to achieve the macro level of brain modeling and simulation. This approach in brain modeling is referred to as *Large-scale Brain Network* or *Neural-Mass* Modeling. Additionally, TVB incorporates individual brain-imaging data with the models to achieve personalized, patient-specific digital brain twins. The digital brain twins can be used in clinical settings such as neurosurgeries and real-time brain-interface systems. However, building and utilizing such large-scale brain models requires high performance and low latency in terms of computation.

Today, heterogeneous architectures, due to their enhanced performance, energy efficiency, and better flexibility are being utilized increasingly in computing systems ranging from low-power edge devices to high-performance cloud infrastructures. Versal Adaptive Compute Acceleration Platform (ACAP) is a high-performance, heterogeneous computing platform developed by Xilinx/AMD. Versal ACAP offers an array of vector processors, specialized in artificial intelligence (AI) and signal processing workloads, next to the traditional programmable logic with extensive memory resources. The large-scale neural-mass simulation problem at hand has heterogeneous computational needs, which makes it interesting to see how it maps to the Versal ACAP. In this work, we explore the benefits and challenges of using heterogeneous systems (specifically, the Versal ACAP) in the high-performance workload of large-scale neural-mass simulation.

We designed and implemented a dataflow-style system for large-scale brain network simulation on the Versal ACAP. Two different versions of this system, namely *AIE-Only* and *Heterogeneous*, were developed in order to explore the capabilities of the Versal ACAP when accelerating our application. Compared to the current GPU version of TVB, the Heterogeneous implementation performed on average around $4\times$ slower in terms of throughput. However, the Versal ACAP Heterogeneous implementation delivered around $13\times$ lower latency, two orders of magnitude better energy efficiency, and around $2\times$ better power efficiency compared to the GPU version of TVB. Although the Heterogeneous implementation falls short in terms of throughput compared to the GPU version, its lower latency and energy efficiency make it suitable for real-time applications that might use large-scale neural-mass modeling such as virtual surgery or brain interface devices.

Contents

1 Introduction

For centuries, understanding the brain and the way it works has been of great concern to scientists. The National Academy of Engineering of the US has classified reverse engineering the brain as one of the grand challenges of the 21st century [19]. Many efforts have been made 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 address brain disorders. The endeavor to build better and more biologically accurate Artificial Intelligence systems can also benefit from brain simulations.

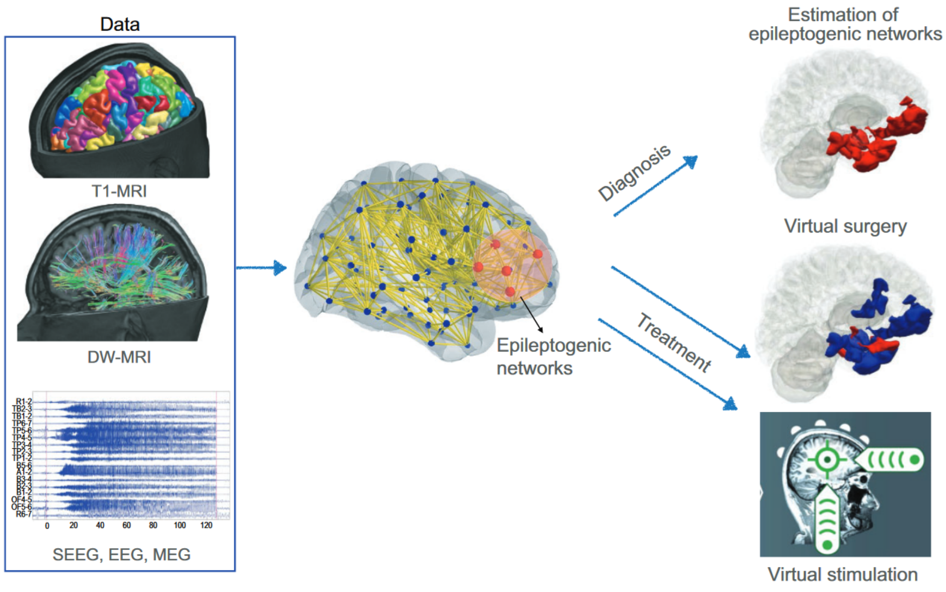


Figure 1.1: Workflow of The Virtual Epileptic Patient (VEP) [57] (with modifications)

The field of computational neuroscience, which studies the organization of the brain and the way it processes 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, coarse-grained brain simulation is effectively being used in addressing brain diseases, with one of its prime examples being in the study of epilepsy [57]. Epilepsy affects more than 50 million 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) [31] is an effort to create personalized digital twins for epileptic patients' brains that can help with the surgery. Since epilepsy is a condition involving several connected brain structures, the VEP can help neurosurgeons estimate the regions that are afflicted as shown in Figure 1.1. Using VEP can help reduce neurological problems and as a result, reduce surgery risk. VEP and similar large-scale network models such as the ones meant for Alzheimer's, Parkinson's, or schizophrenia, are being used extensively in the medical setting, providing new insights for scientists and personalized medicine for affected patients [57].

In another application, large-scale network simulation can be used in a control unit of a brain-interface system [59, 45]. In these examples, there are medically 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, a main challenge is how the system can analytically calculate the stimulus parameters that it is going to perform. 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 strategy that suppresses the seizure attacks.

1.1. Motivation

Large-scale 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 [33]:

1. The whole-brain model structure, meaning the high-level brain regions that are of concern to 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 connectivities can be inferred using different methods, from statistical analysis of 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 learning mechanism runs the simulation many times, and each time it checks how close the output of the model is to the actual brain data. According to this, the parameters of the model are tweaked to get it closer to the actual model or to obtain a distribution over likely values of these parameters. This process, which is shown in Figure 1.2, is referred to as model fitting, requires a large number of simulations that could take a long time to finish, making achieving the goal of having a personalized, patient-specific, digital twin of the brain quite challenging. Furthermore, as mentioned earlier, this digital brain twin can also be used in a closed-loop control system that interfaces with the brain and requires quick and real-time simulations. In conclusion, **having a high-performance platform that can run such brain network models will significantly help with building and utilizing patient-specific digital brain twins.**

1.2. Research Question and Thesis Goals

There are many tools and frameworks that utilize high-performance computing for brain simulation [22, 51]. However, the use of heterogeneous acceleration platforms for large-scale brain modeling is an interesting research area that has not been explored before. Therefore, the research question of this thesis can be formulated as follows:

What are the benefits and challenges of utilizing heterogeneous systems in high-performance computing, specially in the application of large-scale brain network model fitting and simulation?

Based on this research question, the following goals are defined for this thesis:

1. Understanding the application of large-scale brain network simulation and its computation implications.
2. Understanding the target heterogeneous platform and its tool stack.
3. Design of an application-specific design for the brain simulation problem for implementation on the target device.
4. Implementation of the designed accelerator for whole-brain simulation on the target device, comparable in throughput, latency, and power with the current latest work.

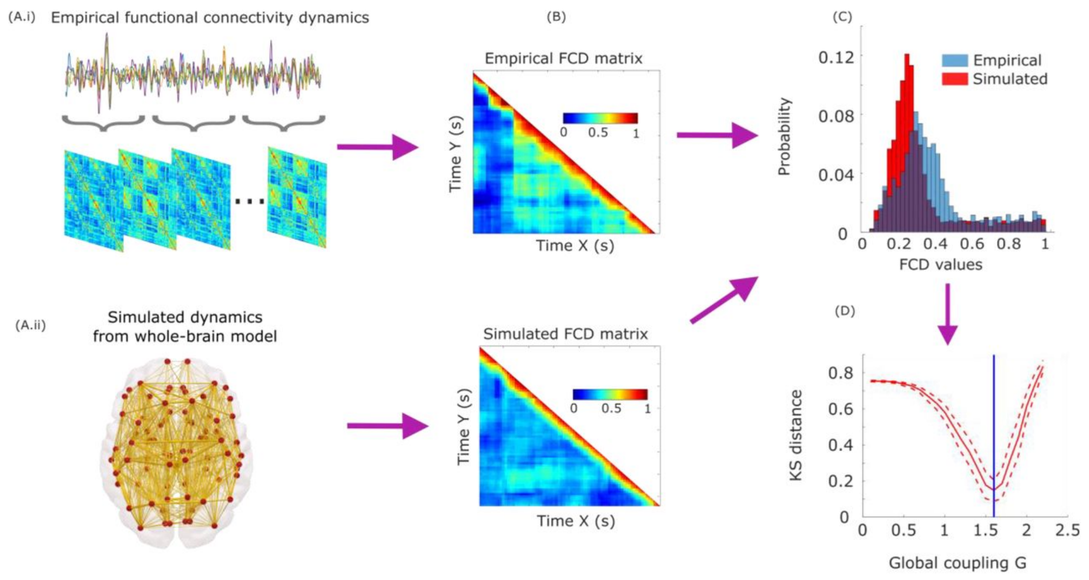


Figure 1.2: Overview of a Model Fitting Procedure [35]. **(A.i)** Patient-specific data are obtained from empirical functional MRI. **(A.ii)** Simulation is performed with a certain set of model parameters. **(B)** The data of both simulated and empirical processes are used to construct Functional Connectivity Dynamics (FCD) matrices. This time-versus-time matrix shows how the connectivity structure of different brain regions that share functional properties change over time. (**C**) Histogram of the distribution of empirical and the simulated FCD matrices are calculated. The similarity between the empirical and simulation histograms is calculated using Kolmogorov-Smirnov (KS) distance. (**D**) Based on the calculated KS distance, the simulation model parameters are changed and steps A.ii to C are repeated. After a large number of simulations, the set of model parameters that is the most similar to the empirical data (lowest KS distance) can be regarded as the best fit for that specific patient.

1.3. Thesis Methodology

In order to understand the benefits and challenges of using heterogeneous systems in the context of brain modeling, we first have to understand the problem and the scope of it. Related work and literature concerning large-scale brain network modeling, neural-mass models, numerical methods for computational neuroscience, (artificial) neural networks, and brain simulation platforms in addition to the documents and information regarding the target compute platform will be studied.

In the next step, the boundaries of the problem that requires acceleration are set. Then, the problem will be analyzed in its different aspects and possible optimizations possible at the algorithmic level will be performed. The target computation platform will also be considered throughout this analysis. With the scope and boundaries of the problem found and formulated, possible architectures for the problem are explored. These architectures will also consider the target platform to make a design that best fits the device. The requirements of each of the possible architectures will be analyzed and compared to the resources of the target device.

In the next step, the system is implemented on the target device. The system will be benchmarked throughout the implementation process and based on the performed benchmarks, improvements will be made to the system. In the end, the results of the implemented system in terms of speed, power, and energy will be benchmarked and compared to the state-of-the-art related work.

1.4. Thesis Organization

In Chapter 2, 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 Chapter 5, the performance results of the implemented systems in addition to the related works are presented and compared. Finally, in Chapter 6, 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 86 billion neurons, interacting with each other in a complicated way [24]. 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 axons 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 the axons 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.

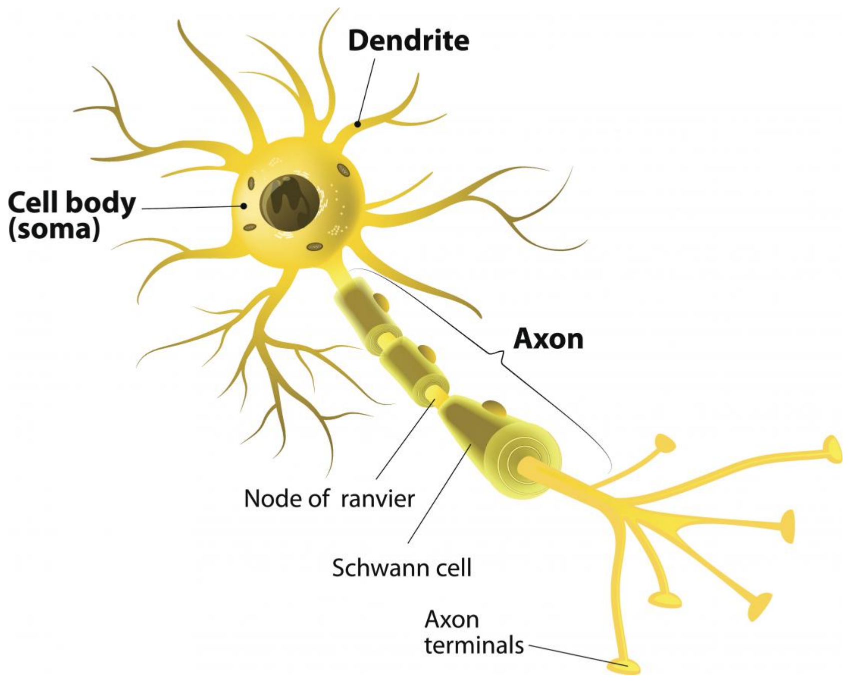


Figure 2.1: Structure of a Neuron [54]

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 [24]. 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.

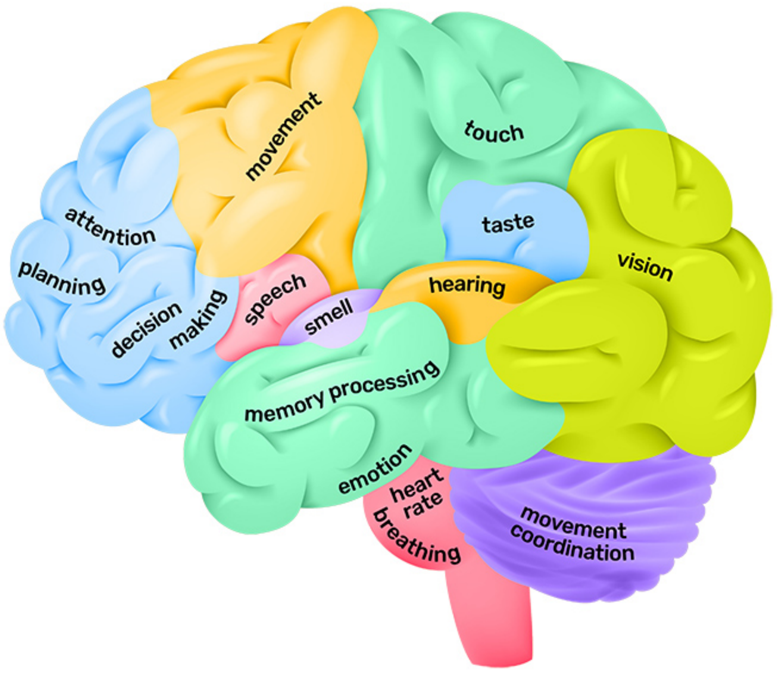


Figure 2.2: Brain Regions and Their Functionality [47]

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. The spikes produced by a neuron can have different characteristics [30], encoding information in every aspect of them. Figure 2.3 illustrates some of the most important types of a neuron's spiking activity in response to simple DC-current inputs.

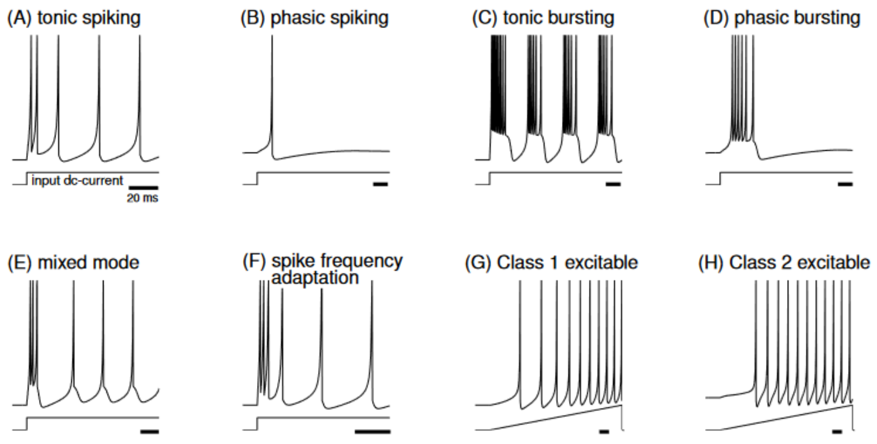


Figure 2.3: Some Different Characteristics of Spiking Neurons [30] (with modifications)

Over the history of computational neuroscience, brain modeling has been performed at different levels of abstraction as shown in Figure 2.4 [43]. Early efforts of modeling the brain were focused on the simulation of a single neuron [26, 20]. These models describe how electrical potentials in neurons are initiated and propagated. Generally, a model of a neuron consists of a set of non-linear differential equations with variables describing different characteristics of the neuron. A good model at this level of abstraction is capable of producing a wide range of spiking behavior shown in Figure 2.3. The single-neuron models provided a foundational understanding of the behavior of a single neuron. However, they did not offer much insight into how a population of neurons interact with each other to perform a certain task.

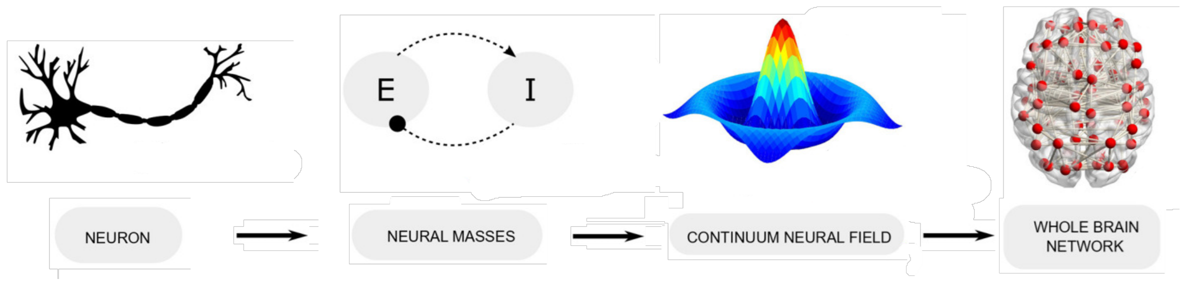


Figure 2.4: Evolution of Computational Neuroscience [43] (with modifications)

In an effort to understand the activity patterns of a population of neurons, neural-mass [60, 10] and neural-field [32, 1] models were formulated. Neural-mass models looked at the activity of a mass of neurons instead of just a single neuron, where less chaotic and irregular activities that are found compared to just one neuron [18] allowed for less computationally expensive brain simulations. Although these models had great success in theoretically explaining many neural activities, they had limited applicability in clinical settings [43]. Many neuronal phenomena that 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 [43]. This means that patient-specific imaging data could be combined with brain simulations to enable more precise, personalized medical practices [49, 16]. In this fashion of simulation, the nodes (which represent the activity of a collection of neurons in a brain area) 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 descent or Bayesian inference) the parameters are learned. The large amount of simulation needed for model fitting is one of the main reasons that we have for accelerating this application.

The nodes are described by differential equations, with the impact of couplings from other areas, external inputs, and noise [17]. In addition to the connection parameters, the parameters related to the local dynamics of the nodes are also tunable, and the result of each simulation run is a time-series [43]. For any application that large-scale brain modeling is used for, two questions need to be asked:

1. What local dynamics model best fits the requirements of the application?
2. How the nodes of the model are connected to each other?

The Virtual Brain (TVB) [44] is one of the leading frameworks in providing large-scale brain modeling. TVB uses biologically realistic connectivity data to perform full brain network simulations. This framework provides the simulation tools, connectivity atlases, integrator, local models, and in general everything that is needed for large-scale brain simulation. TVB assumes a standard model for each of the nodes as shown in Equation 2.1 [57].

$$\dot{\psi}\left(x_i,t\right) = L\left(\psi\left(x_i,t\right)\right) + \int_{\Gamma_l} g_{ij} S\left(\psi\left(x_j,t-\tau_{ij}\right)\right) dx_j + \int_{\Gamma_g} G_{ij}\eta_{ij} S\left(\psi\left(x_j,t-\tau_{ij}\right)\right) dx_j + w(t) \tag{2.1}$$

In Equation 2.1, $\psi\left(x_i,t\right)$ is the neural activity at time t and location x_i, L is the local dynamics of the node as a function of the neural activity, g_{ij} is the coupling input from nearby nodes (local connectivities), and G_{ij} is the coupling input from the distant nodes (global connectivities). S is the non-linearity that is applied to the connectivities coming from other nodes, and $w(t)$ represents the dynamic noise. Additionally, both the local and global connectivities include delays in their calculations (τ_{ij}). This delay is due to the limited speed of signal propagation in the brain, and it depends on the distance between the two nodes and the speed at which the signal travels in the brain. This signal propagation speed can be both fixed or distributed [39], which is more biologically accurate. In the large-scale brain network simulations, the speed of travel is assumed to be fixed as it is sufficiently accurate for the purpose of the modeling. Furthermore, based on the abstraction level at which the simulation is being performed, the local connectivity (g_{ij}) in Equation 2.1 can be ignored and assumed to be absorbed in the local neural-mass dynamics.

Equation 2.1 shows the neural activity of a certain location at a certain time. As mentioned earlier, in large-scale brain network modeling, 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 [41], each for a certain reason and use. Figure 2.5 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.

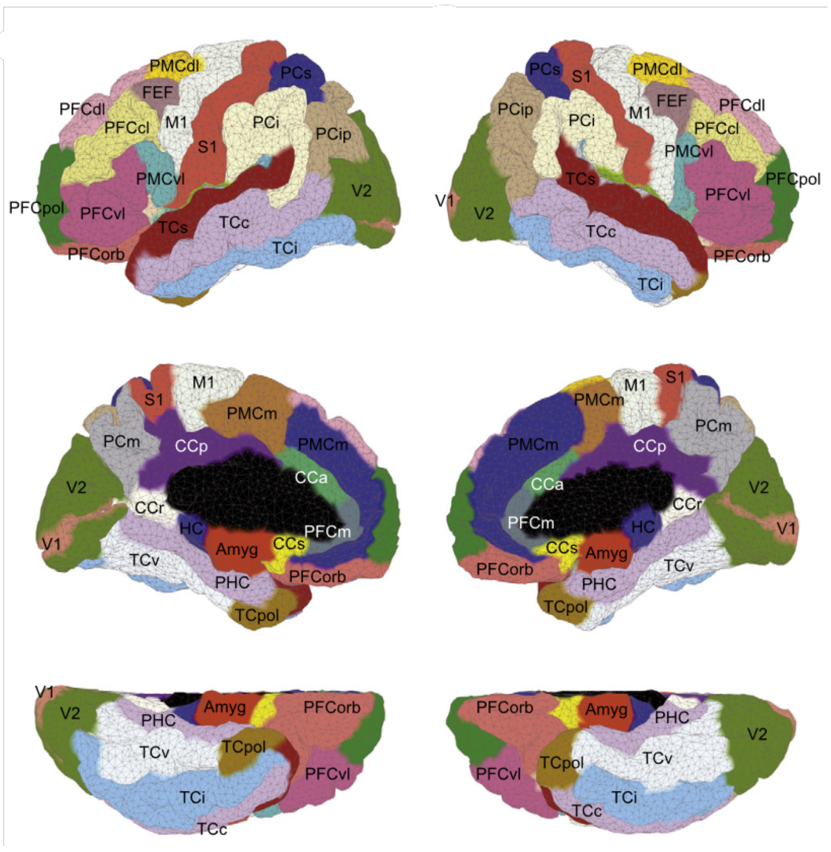


Figure 2.5: TVB Default Brain Dataset [50]

In order to perform large-scale brain network simulation, one must solve the differential equation shown in Equation 2.1 for each of the nodes of the brain, for instance, the ones that are shown in Figure 2.5. 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 [37].

2.3. Multi-Layer Perceptrons and NeuralODE

Multi-Layer Perceptrons (MLPs) are a family of neural networks with layers of artificial neurons (or perceptrons) connected densely together, as shown in Figure 2.6. The inputs to each perceptron are multiplied by weights and added together with a bias value. The result of this summation enters a non-linearity function which produces the output of the perceptron. Many of these perceptrons are put together in a layered structure shown in Figure 2.6 to form a multi-layer feed-forward neural network called an MLP. The parameters of the MLP (θ), meaning the weights and biases of the perceptrons, are calculated through a training process called *backpropagation*. In this process, we start with a labeled training dataset, initial weight and bias values, and a loss function (L) which we try to minimize through this process [42]. For a batch of training data, we calculate the output of the network and evaluate the loss function for the calculated and expected values. In the next step, we look at the gradient of the loss function with regards to the network parameters ($\frac{\partial L}{\partial \theta}$) to see the effect of the parameters on the loss function and change the parameters towards minimizing the loss function. This process is repeated for all the training datasets, leaving us with the final parameters of the network. Hyperparameters of the network, such as the number and size of the hidden layers, training batch size, and learning rate have an important role in the training quality of the MLPs, and tuning them to find the best suite of hyperparameters is also part of the training process.

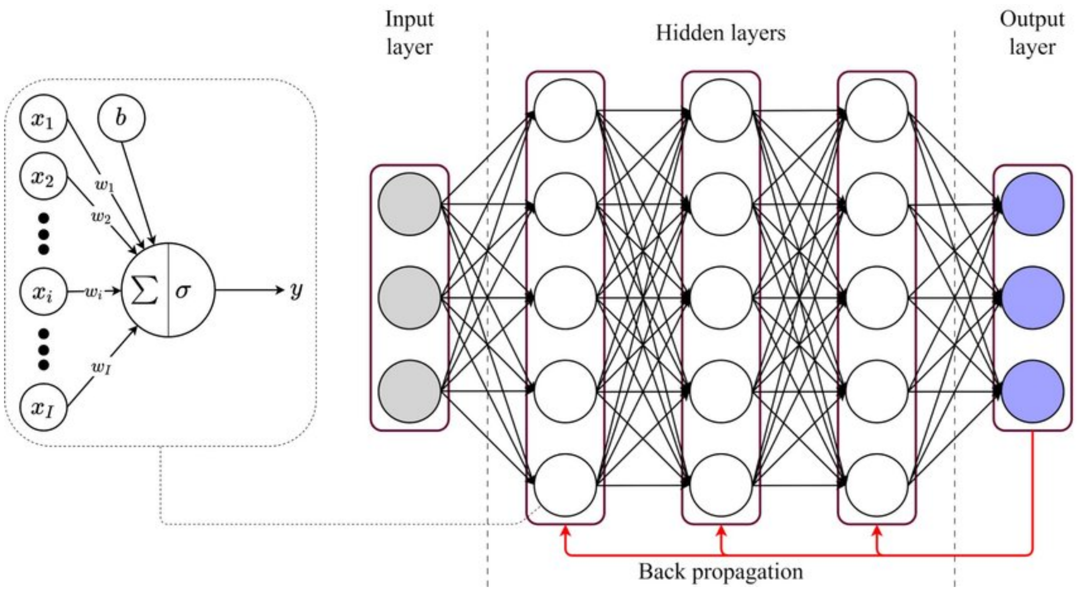


Figure 2.6: MLP Structure [56]. Each perceptron (on the left) has inputs (x_1-x_I) which are multiplied by their corresponding weights (w_1-w_I) and added together and a bias value (b). The result is then passed through an activation function (σ), which produces the output of the perceptron. Many of these perceptron are organized together in a layered structure as shown on the right side.

MLPs are universal function approximators [27], meaning that any continuous function $f : \mathbb{R}^N \to \mathbb{R}^M$ can be approximated by an MLP with at least one hidden layer, and input and output layers with N and M neurons, respectively. This means that the local dynamics function of the nodes mentioned in Section 2.2 can also be approximated using an MLP. This is beneficial for two reasons:

1. As mentioned in Section 2.2, there is a wide variety of neural-masses governing the local dynamics of the brain models. Each of these models describes the activity of the brain regions in a certain way with a certain set of differential equations. This makes creating a low-level hardware accelerator for each of these neural masses a difficult task, as implementing each of the possible functions that could be used for modeling is a time-consuming and tedious task. Having an MLP approximating the local dynamics of the nodes allows us to switch between different neural-mass models in the hardware by simply changing the parameters of the network. This makes the platform a general platform, where the local dynamics of the nodes are not limited by the hardware implementation.

2. Having an MLP allows us to go beyond the mathematical models and train the network using the data collected directly from the patient (for example using EEG). In this way, the patient-specific-trained MLP represents the local dynamics of the nodes exclusive to the patient. This makes all parts of the system, from connectivities to the local dynamics, very general making it suitable for different models and simulations. Although the use of an MLP has the drawback of decreased accuracy (because MLPs only *approximate* the functions) and also higher computational need, it makes all parts of the system specific to the patients, resulting in better outcomes for all patients.

As mentioned earlier, the MLPs approximate the local dynamics function of the nodes. The nodes are mathematically modeled as dynamical systems, and training an MLP to approximate these types of functions could become challenging and require special processes. Neural Ordinary Differential Equations (NeuralODE) [14] represent a new viewpoint in modeling continuous dynamics using neural networks. In NeuralODEs, a black-box ODE solver is used in the process of training (and more specifically, as part of the loss function) to produce the expected values of the system. Using a method called *adjoint sensitivity* [11], the gradient of the loss function with regards to the parameters of the neural network is calculated. Since the brain, and the models that try to simulate the brain, are dynamical systems, the NeuralODE can be used to better train the MLPs that are going to approximate them [9].

2.4. Problem Formulation

The problem at hand is the acceleration of large-scale **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 effects of the centers on each other are called the *couplings* and are modeled as input variables to the differential equation of the centers. A center's effect on another center takes time to arrive, as in the physical brain signals travel at limited speeds. As a result, our problem boils down to solving a series of *coupled delay 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 2.2 and 2.3.

$$\frac{d\vec{X_i}(t)}{dt} = F\left(\vec{X_i}(t),\, \vec{C_i}(t)\right) + u(t) + \eta(t) \tag{2.2}$$

$$\vec{C_i}(t) = K_{post}\left(\sum_{j=1}^{N} W_{ij} K_{pre}\left(\vec{X_j}(t - D_{ij}), \vec{X_i}(t)\right)\right) \tag{2.3}$$

- The local dynamics of the centers are represented with $F\left(\vec{X_i}(t),\, \vec{C_i}(t)\right) = \begin{bmatrix} f_1(X_{i0}(t),\, C_{i0}(t)) \\ f_2(X_{i1}(t),\, C_{i1}(t)) \\ \dots \end{bmatrix}$.

  As mentioned in Section 2.3, when using NeuralODEs, this vector of functions can be replaced by a Multilayer Perceptron (MLP) that can approximate F, meaning $MLP\left(\vec{X_i}(t),\!:\; \vec{C_i}(t)\right) \approx F\left(\vec{X_i}(t),\, \vec{C_i}(t)\right)$.
- 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 2.2. As shown in Equation 2.3, 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 $K_{pre} \in (\mathbb{R}^M, \mathbb{R}^M) \to \mathbb{R}$ and $K_{post} \in \mathbb{R} \to \mathbb{R}^M$ 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 [50].

- 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 in the case 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 ($\eta(t)$) 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[1]. 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 holds 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 2.4 and 2.5 show the Forward Euler integration for the NMM problem. In Equation 2.4, the term $\xi(t)$ represents the integration of the noise term mentioned in Equation 2.2 and it can be replaced with a normal distribution with a variance of h, meaning $\xi(t) \sim N(0,h)$ [39].

$$\vec{X}_i(t+1) = \vec{X}_i(t) + h \cdot \left(MLP(\vec{X}_i(t)) + \vec{C}_i(t) + u(t)\right) + \xi(t) \tag{2.4}$$

$$\vec{C}_i(t) = K_{post}\left(\sum_{j=1}^{N} W_{ij} K_{pre}\left(\vec{X}_j(t-d_{ij}), \vec{X}_i(t)\right)\right) \tag{2.5}$$

In order to make understanding of the problem, in addition to the design and implementation of the system, easier, some simplifying assumptions were made to Equations 2.4 and 2.5 [50]:

1. The effect of centers on each other happens through only one state variable. This means that $\vec{X}_i$ consists of one *global* variable that can be seen by all centers and multiple local variables that can only be seen by center i itself.
2. 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.
3. 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.

With the above assumptions in mind, we can rewrite Equations 2.4 and 2.5 as follows:

$$\vec{X}_i(t+1) = \vec{X}_i(t) + h \cdot \left(MLP(\vec{X}_i(t)) + C_i(t)\right) \tag{2.6}$$

$$C_i(t) = K_{post}\left(\sum_{j=1}^{N} W_{ij} K_{pre}\left(X_j(t-d_{ij}), X_i(t)\right)\right) \tag{2.7}$$

The $C_i(t)$ calculated in Equation 2.7 is only added to one of the state variables of center i. Additionally, the pre- and post-synapse functions are redefined as $K_{pre} \in (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ and $K_{post} \in \mathbb{R} \to \mathbb{R}$.

[1]Actually, due to the presence of a stochastic term it would be more accurate to refer to the integration method as Euler-Maruyama, but in this report, we refer to it simply as Forward Euler.

2.5. Computing Paradigms

The most commonly used computers today are based on a computing paradigm proposed by John von Neumann, a Hungarian computer scientist. The von Neumann computers, in general, separate the computing unit from the memory. This means that instructions and data are stored in the memory, and at each iteration, an instruction is loaded from the memory to the CPU. Based on the instruction, the needed data is also loaded from the CPU, and after the calculations are done, the results are written back to the CPU. Figure 2.7 shows the block diagram of this paradigm. The von Neumann computers are very good at being general purpose, meaning that they are capable of performing any computing workload. However, the cost of this generality is performance, where this constant loading and storing of data consumes time and energy. Efforts to improve the performance of von Neumann machines such as increasing the clock frequency, caching the data, adding multiple threads and cores, vector calculations, and using very long instruction words (VLIW) have been performed, but the bottleneck of accessing the memory for instructions and data still remains in these types of machines.

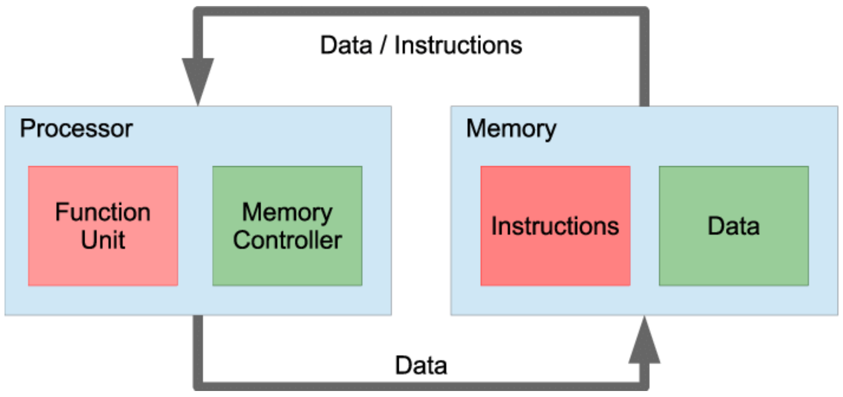


Figure 2.7: von Neumann Computing Paradigm [38]

Alternatively, the dataflow computing paradigm tries to eliminate this bottleneck by removing the notion of instruction. As shown in Figure 2.8, a dataflow machine stores the data in the memory similar to a von Neumann machine, with the difference that when a dataflow machine loads the data to the processing unit, it streams it through a series of operation engines. These engines perform a small operation on the inputs and produce a result for the next engine or the main memory. As opposed to von Neumann computers, the dataflow computers are not fit for general-purpose workloads, because the engines and the way they connect to each other differ from application to application. On the other hand, because the data is only read once from memory (because the data stays in the processing unit and is not written back until it is no longer needed), and there are no instructions to be fetched, the dataflow machines can perform significantly better both in terms of speed and power consumption. It should be noted that dataflow computers are application-specific, and the structure of the processing unit changes from application to application. This requirement for custom structure in processing is one of the main reasons that reconfigurable devices, such as FPGAs, are appropriate platforms to realize dataflow computers. Ideally, tools and compilers exist to map the high-level algorithms to the low-level engines and connections in the processor [53].

2.6. Versal Adaptive Compute Acceleration Platform

The Versal Adaptive Compute Acceleration Platform (ACAP) was first introduced by Xilinx in 2018[2]. The Versal ACAP is a heterogeneous compute platform, built on the TSMC 7nm FinFET technology [6], which combines the traditional FPGA fabric (referred to as *Adaptable Engines* or *Programmable Logic (PL)*) with intelligent engines [65]. Figure 2.9 shows the functional block diagram of the Versal ACAP.

The device used in our project is a *VC1902 AI Core* series, which is on a VCK190 Evaluation Kit.

[2]After acquiring Xilinx in 2022, AMD rebranded the ACAP product line to the Versal Adaptive SoC. Throughout this report, we use the Versal ACAP term for this product line.

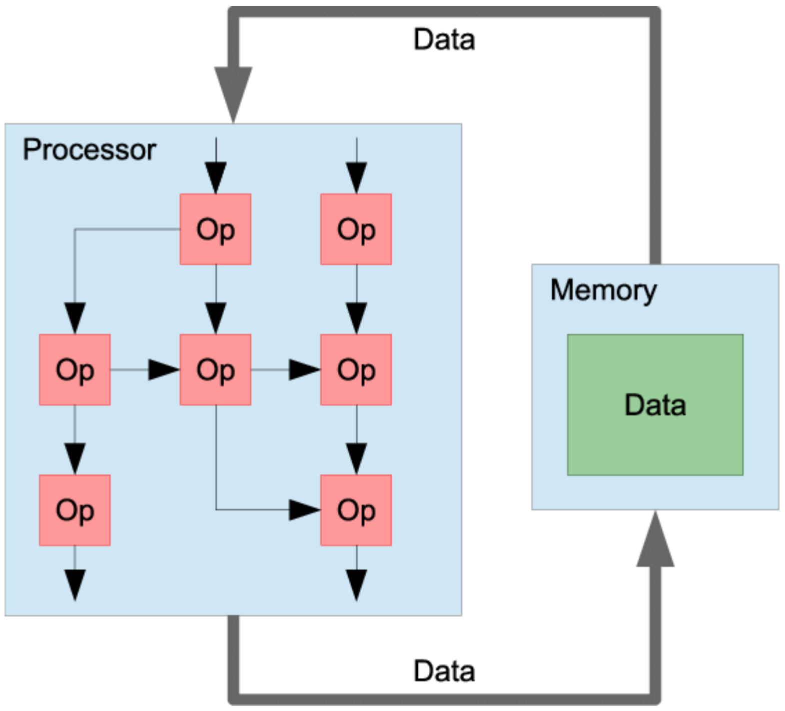


Figure 2.8: Dataflow Computing Paradigm [38]

The VC1902's Intelligent Engines contain the most amount of AI Engines, optimized for networking, digital signal processing, and cloud workloads. For the rest of this section, we dive into details of different parts of this device.

2.6.1. Processing System (PS)

The processing system (Scalar Engine in Figure 2.9) of the Versal ACAP is a dual-core ARM Cortex-A72 processor. Next to this processor, there is an ARM Cortex-R5F processor for real-time applications as well [7]. The PS has 256 KBytes of on-chip memory, in addition to 4 off-chip DDR memory controllers. The PS can run Linux, or operate bare-metal and acts as the controlling entity of the Versal ACAP, communicating with and configuring other parts of the device.

2.6.2. Programmable Logic (PL)

The programmable logic (Adaptable Engines in Figure 2.9) of the Versal ACAP is a traditional FPGA fabric with direct connections to the Scalar and Intelligent Engines present in the device. The programmable logic has around 2 million logic cells, in addition to around 900,000 LUTs and 2000 DSP engines [64]. The PL is fully compatible with purely FPGA-based applications, in addition to high-performance heterogeneous systems.

The PL has 4.25 MBytes of Block Ram (BRAM) and 16.25 MBytes of Ultra Ram (URAM), totaling 20.5 MBytes of PL SRAM memory [64]. This amount of memory makes the PL a very good candidate for custom application-specific memory hierarchies which according to Xilinx mitigates the high latency and uncertainty of the cache-based systems [65].

2.6.3. AI Engines (AIE)

The AI Engines of the Versal ACAP are the differentiating factor of this device compared to other FPGAs and accelerators. AMD includes the AIE in a variety of its products (including the Versal ACAP) as the XDNA Neural Processing Unit (NPU) architecture [2]. The AIE architecture (or the XDNA architecture) is a spatial dataflow architecture that provides more compute density with lower power consumption. In general, the AIE consists of 400 very long instruction word (VLIW) and single instruction, multiple data (SIMD) processing units that are organized in a 2-dimensional array structure [3]. As shown in

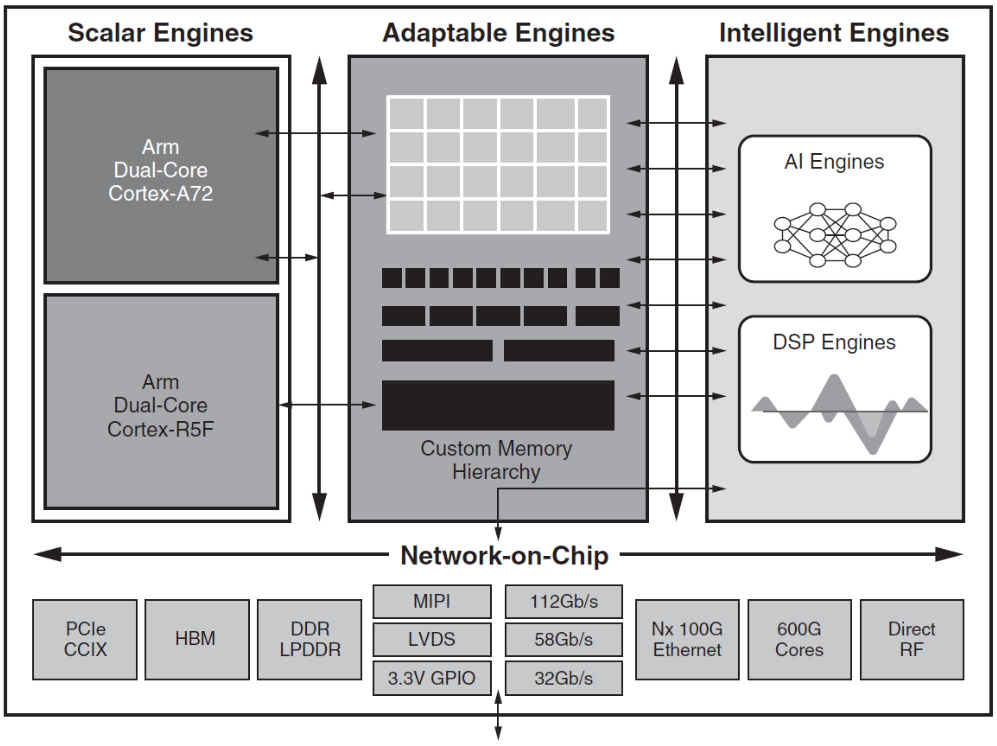


Figure 2.9: Versal ACAP Functional Block Diagram [65]

Figure 2.10, each AIE tile has an engine, a memory block, and an interconnect block.

The processing engine in the AIE tile consists of a scalar processor and a vector processor running at 1.25 GHz, 16 KBytes of program memory, and 3 address generators. The vector processor of the AIE tile is capable of performing 8 single-precision floating-point MAC operations per cycle (20 FLOPs per second). The memory module of each AIE tile has 32 KBytes of data memory divided into eight banks, a memory interface, and a Direct Memory Access (DMA) block. Each AIE tile can access four different memory modules, its own and three of its neighbors. This enables the tiles to move data between themselves by putting it in the memory of neighboring tiles.

In terms of communication, as shown in Figure 2.11, there are four main ways for the AIE tiles to move data between them [3].

1. **Using Shared Memory:** As mentioned earlier, each AIE tile has access to three of its neighbors' memory blocks. This means that two AIE tiles that are neighbors with another tile can share data by putting it in the memory of the tile that they both have access to. To avoid data-access conflicts, different banks of the shared memory module can be used as ping-pong buffers, where data is written to different banks alternatively.
2. **Using Memory and DMA:** Similar to the previous method, non-neighboring AIE tiles can access each other's memory using DMA and locks. Compared to the previous method, this way consumes more memory resources and has more latency.
3. **Using AXI4-Stream Interconnect:** The 8×50 array of AIE tiles are connected to each other via an AXI4-Stream protocol using each tile's interconnect module. In this way, each AIE tile can send a stream of data to any other tile or a group of tiles (Multicast) in the array. In the multicast streaming fashion, all of the receivers have to be ready for the transaction to happen, unless the transmitter stalls until all the receivers are ready. The AXI4-Stream interconnect of the AIE array is also capable of implementing *Explicit Packet Switching*, where packets of data can be sent from many destinations to a single source with addressing, or from many sources to a single destination.

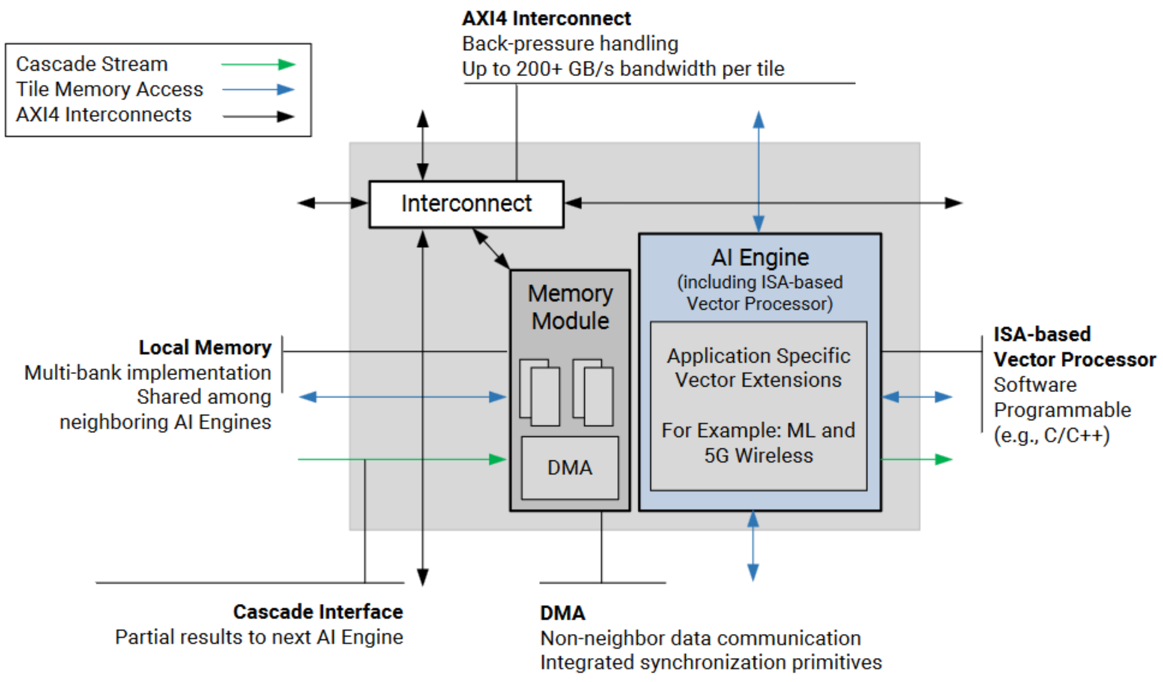

Cascade Stream
Tile Memory Access
AXI4 Interconnects
AXI4 Interconnect
Back-pressure handling
Up to 200+ GB/s bandwidth per tile
Interconnect
Memory Module
DMA
AI Engine
(including ISA-based Vector Processor)
Application Specific Vector Extensions
For Example: ML and 5G Wireless
Local Memory
Multi-bank implementation
Shared among neighboring AI Engines
ISA-based Vector Processor
Software Programmable (e.g., C/C++)
Cascade Interface
Partial results to next AI Engine
DMA
Non-neighbor data communication
Integrated synchronization primitives

Figure 2.10: AIE Tile Architecture [3]

4. **Using Cascade Stream** A chain starting from the bottom-left AIE tile and ending in the top-left tile directly connects all the tiles together. This chain (the Cascade Stream) is 384 bits wide, and each tile can read and write to this stream. This stream can be really helpful when multiple tiles need their values to be accumulated, meaning each tile can read the value from the stream, add its own calculated result to the read value, and write it back to the stream.

In addition to full internal connectivity, the AIE array also has connections to the rest of the device. There are 50 interface tiles at the bottom of the AIE array, which includes PL Interface, NoC Interface, and Configuration tiles. Figure 2.12 shows the details of each of these interface tiles. From the AIE perspective, the PL Interface tiles, all together, provide around 1 $\frac{TB}{s}$ and 1.3 $\frac{TB}{s}$ of bandwidth to and from the PL, respectively.

2.6.4. Network-on-Chip (NoC)

The Network-on-Chip (NoC) of the Versal ACAP is an AXI-Interconnect for information transfer between PL, PS, and the AIE [5]. The NoC is comprised of horizontal (HNoC) and vertical (VNoC) paths, transferring data in all four directions. Figure 2.13 shows the block diagram of the Versal ACAP NoC. he NoC converts the AXI3, AXI4, or AXI4-Stream connections to a 128-bit wide NoC packet protocol [4] and moves it through the device using VNoC and HNoC paths. The HNoC lies at the bottom and top of the Versal ACAP, closer to the I/O banks and the integrated blocks such as PCIe or memory controllers. The VNoC connects the top and the bottom of the device, and the number of it depends on the number of DDR memory controllers.

2.6.5. Tooling and Design Methods

Heterogeneous systems provide tremendous speedups and power efficiency over homogeneous systems because of more specialized, dedicated compute units. However, one of the main costs of these benefits is a more complicated development process [34]. The Versal ACAP is a heterogeneous system with 3 sub-systems within it, the PS, the PL, and the AIE. In order to facilitate the development process for this system, Xilinx proposes a platform-based design methodology [63]. In this methodology, the system is divided into different parts:

1. **The Platform** which contains the foundational blocks of the system. This part of the system, as the name suggests, is only an underlying substrate that does not include any functionalities of the system yet.

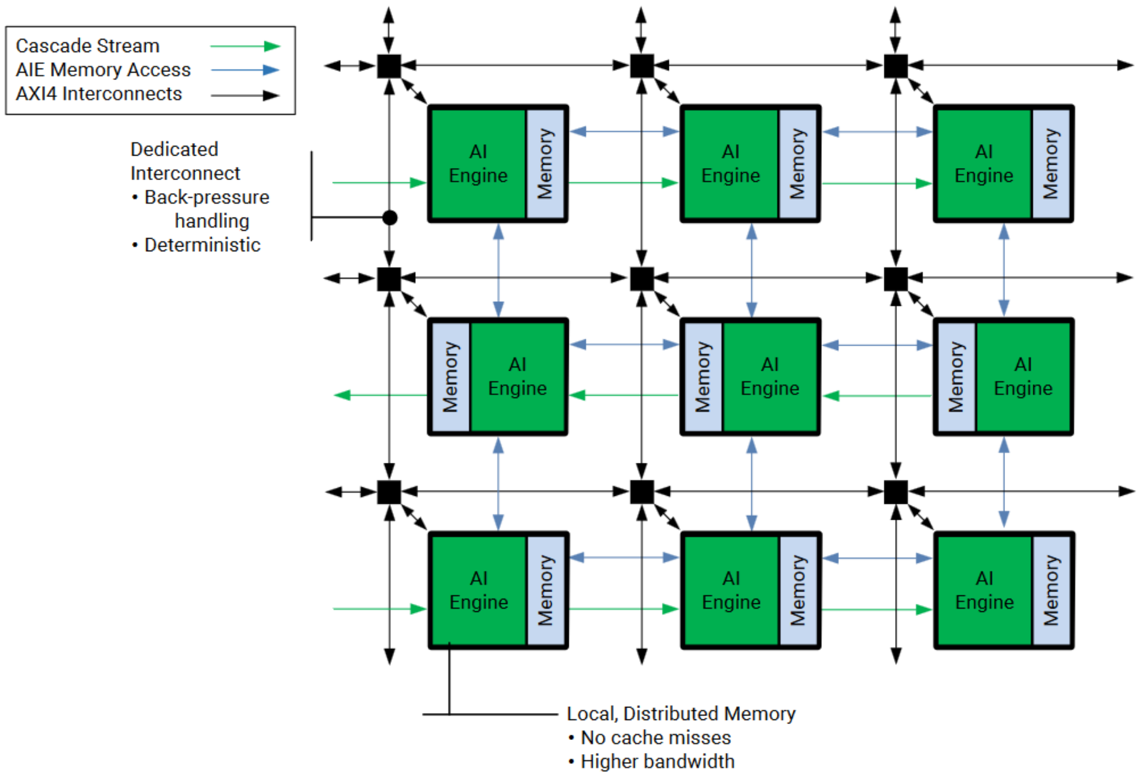


Figure 2.11: AIE Array Communication Methods [3]

2. **The Processing System** which includes the PS, PL, and AIE features that implement the main functionalities of the system on top of the platform.

These two parts of the development process can happen in parallel, making it faster and easier to develop and integrate the Versal ACAP system. Figure 2.14 shows the design flow used in Versal ACAP. The application development for the PS, the PL, and the AIE in addition to the creation of the system platform is suggested to be performed in parallel. In the next step, the PL and the AIE subsystems are integrated and verified. Finally, the custom platform, in addition to the PS application is integrated with the PL and the AIE to form the final system.

For the AIE application development, first, the kernels that are going to run on the AIE tiles are developed in C++. Subsequently, the connections between the kernels (referred to as *Graph*), in addition to the connections to the PL and the NoC are specified. After the kernels and the graph of the AIE application are ready, the `aiecompiler` is used to compile them. The `aiecompiler` first performs the placing process for the kernels, where it maps the kernels to the AIE tiles. Then, the connections between the kernels and the interconnect tiles are routed by the `aiecompiler`. Eventually, after the success of the place and route processes, the kernels inside of the AIE tiles are compiled using a C++ compiler.

For the PL application development, both RTL and HLS modules can be used. However, because in the next steps the tool has to integrate all parts of the system together, the RTL modules must be compiled into a kernel object (`.XO` files) recognizable by the tool. The PS is responsible for controlling the device and configuring and running the kernels inside the PL and the AIE. The application for the PS can be both bare-metal written in C++, or using Linux. After all the applications of different parts of the device are ready, meaning the binary for the AIE tiles, the `.XO` files of the PL, and the code for the PS, the tool links all the parts together, creating a final system. In the linking stage, in addition to all the outputs of different parts of the system, the tool also requires a configuration file where the top-level connections of the device (for example PL-AIE connections) are specified. With all the mentioned information provided to the linking tool, the tool outputs an image file to be booted to the device.

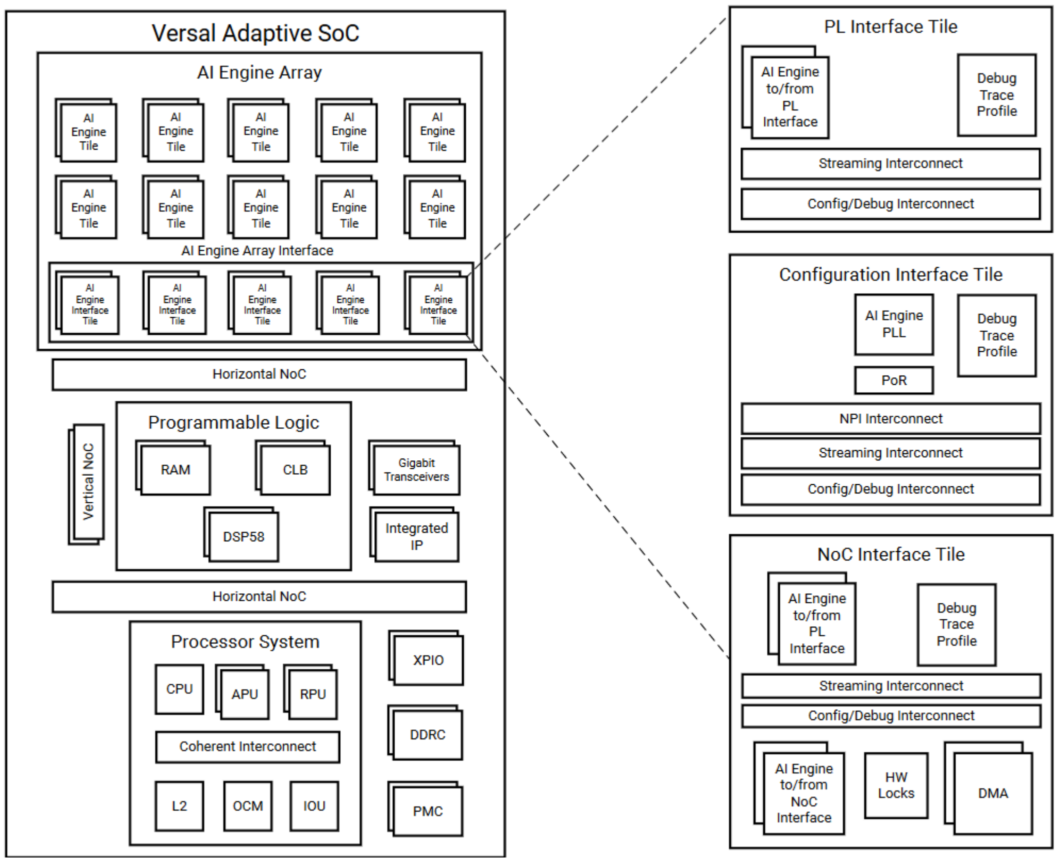


Figure 2.12: AIE Interface Tiles [3]

2.7. Sparse Matrix Calculation

As mentioned in Section 2.2, the connectivities between the centers in the model are specified by a weight matrix (W). This matrix in large-scale brain network models is most usually sparse. In linear algebra, a sparse matrix is a matrix with most of its elements being zero [15]. Because of their unique characteristic, there exist special methods for storing sparse matrices [15]. For any method of storing that is chosen, some sort of compressing algorithm is required to avoid storing the zero values. This compression should be fast and flexible enough to allow for performing different matrix calculations.

The simplest form of representation for the sparse matrices is the *zero-based triplet* form. In this method, the sparse matrix A is represented by three arrays i, j, and x. The array x contains all the non-zero elements of matrix $A \in \mathbb{R}^{M\times N}$ in arbitrary order, and matrices i and j contain the row and column of the array x elements matrix A. This method is easy to understand and construct, but it is difficult to use in many matrix operations [15]. Also, if we assume a sparsity of $0 < SP < 100\%$ for matrix A, in order for this method to be more efficient than storing the whole matrix in terms of memory, the sparsity of the matrix should be more than 66% as shown in Equation 2.8.

$$3 \times (100 - SP) \times (M \times N) < 100 \times (M \times N) \Rightarrow SP > 66.66\% \tag{2.8}$$

Another method to store the sparse matrices is the *Compressed Sparse Row (CSR)* method [13]. This method is the most widely used way to store sparse matrices. In the CSR method, as shown in Figure 2.15, the non-zero elements of the sparse matrix are put into an array row-by-row consecutively (array Data). In another array, a pointer to the first element of each row from the original matrix in the array Data is stored (array Row Pointers). The column indices of the elements in the array Data are also stored in another array (array Column Offsets). A similar method called CSC also exists that instead of keeping the row indexes, keeps the column indexes.

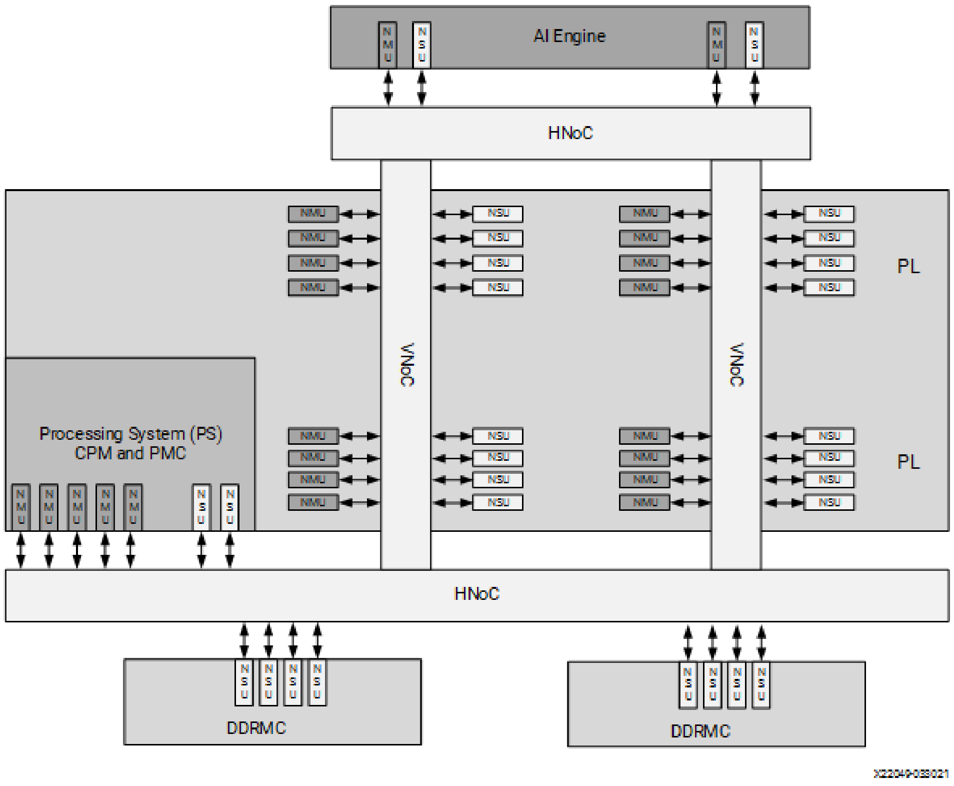


Figure 2.13: Versal ACAP Network-on-Chip Block Diagram [5]

In order for the CSR method to be more efficient in terms of memory usage, the sparsity of the matrix, for large enough matrices, should be more than 50%, as shown in Equation 2.9. This is a better compression compared to the zero-based triplet method mentioned above.

$$2 \times (100 - SP) \times (M \times N) + 100 \times M < 100 \times (M \times N) \Rightarrow SP > (50 + \frac{100}{2N})\% \qquad (2.9)$$

2.8. Related Work

As mentioned earlier in the chapter, in this project we try to implement the large-scale brain simulation using TVB-style models on the Versal ACAP. This means that the related work to our project is mainly the work done by the TVB team, which comes in different versions of the TVB tool. TVB is part of the Human Brain Project (HBP) [25] which was a European flagship project in the field of brain research. As mentioned earlier in the report, TVB is a prominent brain simulation toolkit that generated many methods and findings, especially regarding epilepsy [31, 46, 8, 33, 58].

2.8.1. Original TVB

The original TVB [44] is a Python-based tool for large-scale brain modeling available to neuroscientists. The numerical algorithms behind the original TVB are extracted from the main tool by the TVB team and are open to the public and are referred to as `tvb_algo` [61]. 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, single-core C++ code (`tvb_algo_c`) [40]. 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.

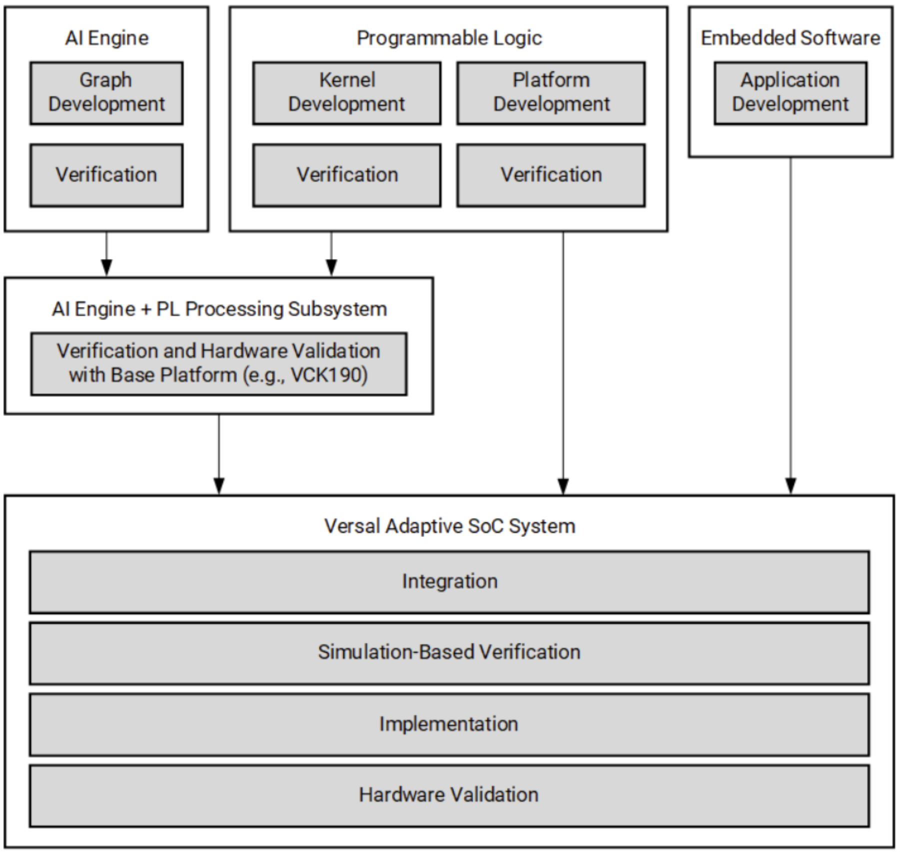


Figure 2.14: Versal ACAP Platform-based Design Flow [4]

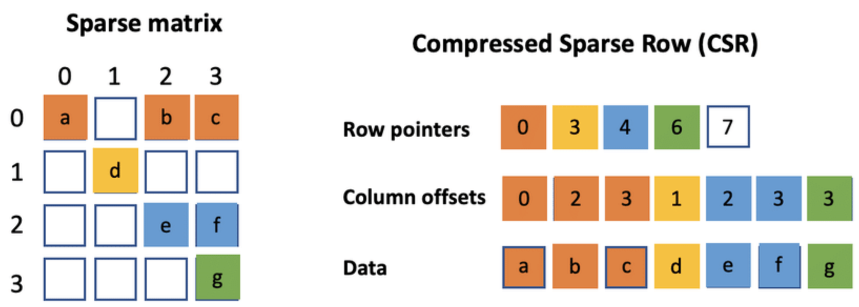


Figure 2.15: Compressed Sparse Row (CSR) Method for Sparse Matrix Representation [28]

2.8.2. Optimized TVB on CPU

The original TVB is developed with the idea of generality, maintainability, and ease of development which often comes at the cost of decreased performance. To tackle the performance bottleneck, other versions of TVB have been developed for faster and more efficient CPU execution.

The Fast TVB [51] is a specialized and optimized C implementation of TVB for a specific neural-mass model. The Fast TVB traded the generality of the original TVB for the higher performance of a single model. Using the CPU resources efficiently, the Fast TVB allows for simulating even very large models in a reasonable time.

TVB C++ [36] 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.

Both of the faster CPU versions of TVB mentioned in this section benefit from the multithreading and SIMD capabilities of the CPU. They both have a limited number of models available (only one for the Fast TVB and only a few for TVB C++), mostly aiming for specific research questions.

2.8.3. TVB on GPU

The original TVB is also implemented as a JAX-based [12] Python package for execution on the GPU [62]. 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 dynamics evaluation.

The main idea behind the acceleration performed in `vbjax` is batching many simulation instances and calculating them all at the same time. As mentioned in Section 2.2, a large number of simulations must be performed in order to fit the brain model into the data taken from the patient's brain. The GPU version of TVB can simulate all of the TVB-style brain models, however, all of the concurrent 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.

2.8.4. Conclusion

Table 2.1 summarizes the related work that we consider in this thesis.

Name	Platform	Language	Notes
Original TVB	CPU	C++	This is a single-core, single-thread port of the original Python-based TVB backend (`tvb_algo`) to C++ by us. *MLP evaluation* and *sparse calculations* are added on top of `tvb_algo` in this implementation.
Fast TVB	CPU	C	This implementation is developed by the TVB team and can only simulate one model.
TVB C++	CPU	C++	This is a multithreaded, SIMD C++ implementation of TVB developed by the TVB team. It can simulate all of the TVB models, but does not have MLP evaluation.
TVB on GPU	GPU	Python (JAX)	This is the GPU version of the TVB algorithm which is developed as a JAX-based package (`vbjax`).

Table 2.1: Summary of the Related Work on TVB

3 Design

In this chapter, the process taken to design the system on the Versal ACAP device is discussed. During the design phase of the project, first, the problem was analyzed and its requirements regarding performance, memory, and transmission bandwidth were evaluated. Then the requirements were examined against the resources available in the Versal platform. Using the analysis of the problem and the platform, three design candidates were studied, and one was chosen to be implemented.

The Neural-Mass Model problem is formulated as shown in Equations 2.6 and 2.7. Algorithm 1 shows the pseudocode for the NMM problem.

Algorithm 1 Large-Scale Neural-Mass Model Problem Pseudocode

```
$W \leftarrow Weights()$
$D \leftarrow Delays()$
$\theta \leftarrow MLP\_Parameters()$
$X \leftarrow Initial\_States()$
$H \leftarrow Update\_History(X)$
for $t \in [1, tf]$ do
    $dX \leftarrow MLP(X, \theta)$
    $C \leftarrow Coupling(X, W, D, t)$
    $X \leftarrow X + h(dX + C)$
    $H \leftarrow Update\_History(X)$
end for
```

The Neural-Mass Model problem is analyzed in Section 3.1, and the three design candidates are presented and discussed in Section 3.2. In Section 3.3, we discuss the positive and negative aspects of each design, and a final architecture is chosen for implementation.

3.1. Problem Analysis

The goal of this section is to model the performance and memory requirements of different parts of the application. This modeling helps us to determine the feasibility and effectiveness of a certain design with respect to the platform that we are using.

As mentioned in Section 2.4, the problem at hand is the acceleration of neural-mass models (NMMs) which consists of solving a series of *coupled delay differential equations* using the Forward Euler method. The differential equations, which are the local dynamics of the centers of the NMM, are calculated using MLP. In order to analyze the problem and derive its requirements, we divide the system into 2 different parts: 1) MLP evaluation and the FE Solver, and 2) Coupling calculations. For each part, the requirements regarding aspects like performance and memory capacity are studied. Additionally, the bandwidth requirements for the data transmissions across different parts of the system are also evaluated within the context of each of the three design candidates and are presented in their own sections. It is worth noting that throughout the analysis, we assume all the values are single-precision floating point numbers, resulting in 4 bytes of memory requirement per value. Additionally, worst-case scenarios were considered to get a better idea of the requirements of the application.

3.1.1. MLP and FE Solver Memory Capacity

In this section, we look into the memory capacity requirement of the MLP and FE Solver. The main memory needs contributors are as follows:

1. **Previous Timestep State Variables** These are needed by the FE Solver to calculate the next step state variables. This requirement can be formulated as follows:

$$\text{Mem}_{\text{PrevStates}} : 4 \times (NM) = 4NM \qquad \text{[Bytes]} \qquad (3.1)$$

2. **MLP Parameters** These are weights and biasses used by the MLP to calculate the local dynamics of the centers at each timestep. The amount of memory needed by the parameters of the MLP depends on the size of the neural network. The input and output layers of the MLP are the same size as the number of state variables per center (M). We can assume that the MLP has H hidden layers, and each hidden layer has L neurons. Additionally, there might be multiple MLPs running at the same time and each requires its own set of parameters separately stored. We assume that the value of the concurrent MLPs running at the same time is E. With the above assumptions, in addition to the presence of bias values and the absence of any optimization like pruning, we can formulate the memory need of the MLP parameters as follows:

$$\text{Mem}_{\text{MLPParams}} : 4 \times E \times ((M+1)L + (H-1)(L+1)L + (L+1)M) \qquad \text{[Bytes]} \qquad (3.2)$$

3. **Intermediate and Coupling Values** This corresponds to the memory needed by calculations of the MLP, the output values, and also to store the coupling values that are needed by the FE Solver. Assuming single-batch MLP evaluation, these memory requirements can be formulated as follows:

$$\text{Mem}_{\text{Inter}} : 4 \times (L + NM + N) = 4N(M+1) + 4L \qquad \text{[Bytes]} \qquad (3.3)$$

So the total amount of memory needed by the MLP and the FE Solver can be calculated by adding the values of Equations 3.1, 3.2, and 3.3:

$$\begin{aligned}\text{Mem}_{\text{MLP+FE}} &= \text{Mem}_{\text{PrevStates}} + \text{Mem}_{\text{MLPParams}} + \text{Mem}_{\text{Inter}} \\ &= 4E((M+1)L + (H-1)(L+1)L + (L+1)M) + 4N(2M+1) + 4L\end{aligned} \qquad \text{[Bytes]} \qquad (3.4)$$

3.1.2. MLP and FE Solver Computation Performance

Almost all of the required computation for this part of the application comes from the MLP evaluation. Assuming that the MLP has H hidden layers with L neurons each, in addition to M neurons for input and output layers, and no optimization (for example pruning), we can formulate the amount of calculations needed for each output evaluation as follows:

$$\text{Comp}_{\text{MLP}} : ML + (H-1)L^2 + ML = 2ML + (H-1)L^2 \qquad [\tfrac{\text{MAC}}{\text{Center}}] \qquad (3.5)$$

In addition to the MLP calculations, the FE Solver performs about $3M\ (\frac{MAC}{Center})$ to evaluate the next step state variable. Equation 3.6 formulates the total amount of computation needed for each center at each timestep of the simulation.

$$\text{Comp}_{\text{FE}} : 2ML + (H-1)L^2 + 3M \qquad [\tfrac{\text{MAC}}{\text{Center}}] \qquad (3.6)$$

As in Section 3.1.1, we assume that there are E different MLP and FE Solvers running at the same time, meaning each of them is tasked with the calculations of $\frac{N}{E}$ centers. Additionally, since we are aiming for real-time simulation, each of the E blocks needs to finish the calculations within a certain time limit. This time limit is less than the timestep h, and we refer to its value as h_{RT} (with a unit of seconds) where $0 < h_{RT} < h$. The lower the value of h_{RT}, the more computation-intensive the application. With all the needed variables defined, we can formulate the required computation performance by each MLP and FE Solver block as follows:

$$\text{Comp}_{\text{MLP+FE}} : \frac{1}{h_{RT}} \times \frac{N}{E} \times \big(2ML + (H-1)L^2 + 3M\big) \qquad [\tfrac{\text{MAC}}{\text{s}}] \quad (3.7)$$

3.1.3. Coupling Calculation Memory Capacity

Memory needs for coupling calculations consist of storing past state variable values for all centers and the connectivity data (including the weight and delay matrices). In order to better analyze the memory capacity requirements of the coupling calculation engines, we define a new matrix $T \in \mathbb{R}^{N\times N}$ which is the combination of connectivity (W) and timestep-delay (d) matrices:

$$\forall\, i,j \in [1,N] : \text{ if } W_{ij} \neq 0, \text{ then } T_{ij} = d_{ij} \text{ otherwise } T_{ij} = 0$$

Matrix T basically keeps the value of d at each position where there is a connectivity between 2 centers. In other words, we apply the sparsity pattern of the weight matrix to the delay matrix. By looking at the columns of T, the maximum value of each column j (max$\,(T_{:j})$) shows the number of timesteps that the state variable of center j ($\vec{X_j}$) is needed for coupling calculation of other centers. max$\,(T_{:j})$ specifies the memory needed for each center of the model to keep its history. The larger the value of max$\,(T_{:j})$, the more memory is needed for center j to keep a history of its state variable.

For each center j, we need to keep max$\,(T_{:j})$ elements. Assuming the worst-case scenario, the total memory needed to store the history of the state variable values is $N * T_{max}$, in which N is the number of centers and T_{max} is the maximum value of matrix T. The real memory capacity requirement is less than $N * T_{max}$ since we don't need to keep the history of all the centers for T_{max} timesteps.

In order to perform the coupling calculations, we need to store the connectivity data (W and d matrices). Assuming the worst-case, meaning no sparsity in the matrices, the memory required for each of the mentioned matrices is N^2 elements. Therefore, the total memory needed to store the connectivity data is $2N^2$ elements.

Adding the requirement for state variable history and the connectivity data, the total memory that the coupling calculations need is as follows:

$$\text{Mem}_{\text{CC}} : (NT_{max} + 2N^2) \times 4 \;=\; 4NT_{max} + 8N^2 \qquad [\text{Bytes}] \quad (3.8)$$

The value of T_{max} depends on the physical distances between the centers in the brain, the conductance speed, and the timestep of the simulation (h). Since these parameters of the simulation are mostly fixed, this means that the value of T_{max} is mostly fixed, and this means if we increase the number of centers in the model, the connectivity data will become the dominant memory-requiring part. This is evident in Figure 3.1 where in models with more than around 500 centers, the connectivity data will consume most of the memory. However, it is worth noting that this is only true when considering the worst-case scenario. In reality, as the number of centers in a model grows, the sparsity of the weight matrix also grows. This means if an implementation stores the connectivity data in sparse format, the memory requirement for them grows closer to a linear pattern rather than quadratically.

3.1.4. Coupling Calculation Computation Performance

The real computation load of coupling calculation heavily depends on the sparsity of the weight matrix. However, here we assume a worst-case scenario of an all-to-all connected NMM. In this case, for each center at each timestep N MAC operations must be performed. Assuming the real-time limitation of h_{RT} mentioned in Section 3.1.2, we can write the required computation performance for the coupling calculations at each timestep as follows:

$$\text{Comp}_{\text{CC}} : \frac{1}{h_{RT}} \times N^2 \qquad [\tfrac{\text{MAC}}{\text{s}}] \quad (3.9)$$

Equation 3.9 shows that the computation required for the coupling calculation grows cubically with regard to the number of centers. However, an all-to-all connected NMM is a worst-case but unrealistic

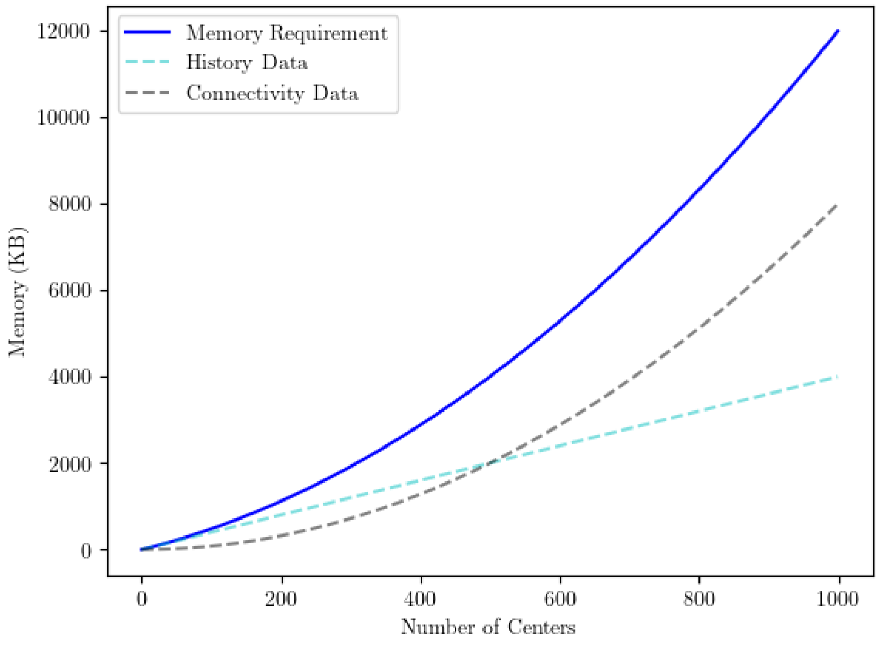


Figure 3.1: Coupling Calculation Memory Requirement ($T_{max} = 1000$)

model, and in reality (as mentioned in Section 3.1.3), the computation growth relative to the number of centers has a more quadratic relation rather than a cubic one.

3.2. Design Candidates

In this section, three different design architectures are presented and discussed. Each of the design candidates is introduced in detail, their performance and memory capacity requirements are compared against the capabilities of the device using formulas derived in Section 3.1, and any new requirements that are specific to the design are also studied.

3.2.1. Design 1: By-Center Parallelization

Introduction

In this design, parallelization happens at center level for each timestep. This means that for MLP evaluation, Forward Euler solving, and coupling calculation we divide the centers into groups and perform the computation of those groups in parallel. Because of the nature of the problem which is a transient simulation, this design only parallelizes the workload within a single timestep and no parallelization happens across the timesteps.

The details of the design are as follows:

1. In this design, there are *MLPFE* and *CC* engines. The MLPFE engines are responsible for the evaluation of the MLP and performing the Forward Euler to calculate the next timestep value. The CC engines are responsible for the calculation of the couplings for the centers.
2. The N centers are assigned to 1 MLPFE and 1 CC engine (we call the number of engines E). Each engine is responsible for calculations of $\frac{N}{E}$ centers. The engines run in parallel, and within a single engine, the calculations of the centers are time-multiplexed.
3. For each center at each timestep, the MLPFE and CC engines run at the same time. When the CC engine is done with its calculation, it transmits the calculated couplings to the MLPFE engine. The MLPFE engine evaluates the next timestep state variables of the centers.
4. Each CC engine has a control unit that controls the flow of data and instruction in the engine. Additionally, each CC engine has dedicated on-chip memory. The previous state variable values

($\vec{X}_i(t-d)$) of the centers that are assigned to an engine are stored in the memory block of that engine. The amount of history for each center depends on the max $(T_{:j})$ value of that center.

5. 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.
6. Unlike the MLP calculations, the coupling calculations for each center require variables from other centers. These dependencies can be of 2 types: Either the value that is needed for the calculation is also stored in the same engine (meaning that both centers that are connected to each other are assigned to the same engine), or the value needed for the calculation is stored in another engine. In the first case, the access to the required value is straightforward. In the second case, the access to the required value happens through the *Inter-Engine Data Exchange* block. This block ensures the fast and efficient transfer of data between different CC engines.
7. The Inter-Engine Data Exchange block is also the entity responsible for communication with the off-chip memory and the PS. Configurations are loaded from the PS to the engines through this block. Additionally, when a value is not needed anymore, it is transferred from the CC engine to the off-chip memory by this block.

As mentioned in Section 2.6, the AI Engines of the Versal ACAP are designed for dataflow-style processing. The MLPFE engine described in this design is a good candidate to reside in the AI Engines. On the other hand, because of the inter-engine dependencies of the CC engines, the AI Engines of the ACAP would not be a good platform to realize them. This is why this design puts the CC engines on the PL region of the device. Figure 3.2 shows the general block diagram of this design.

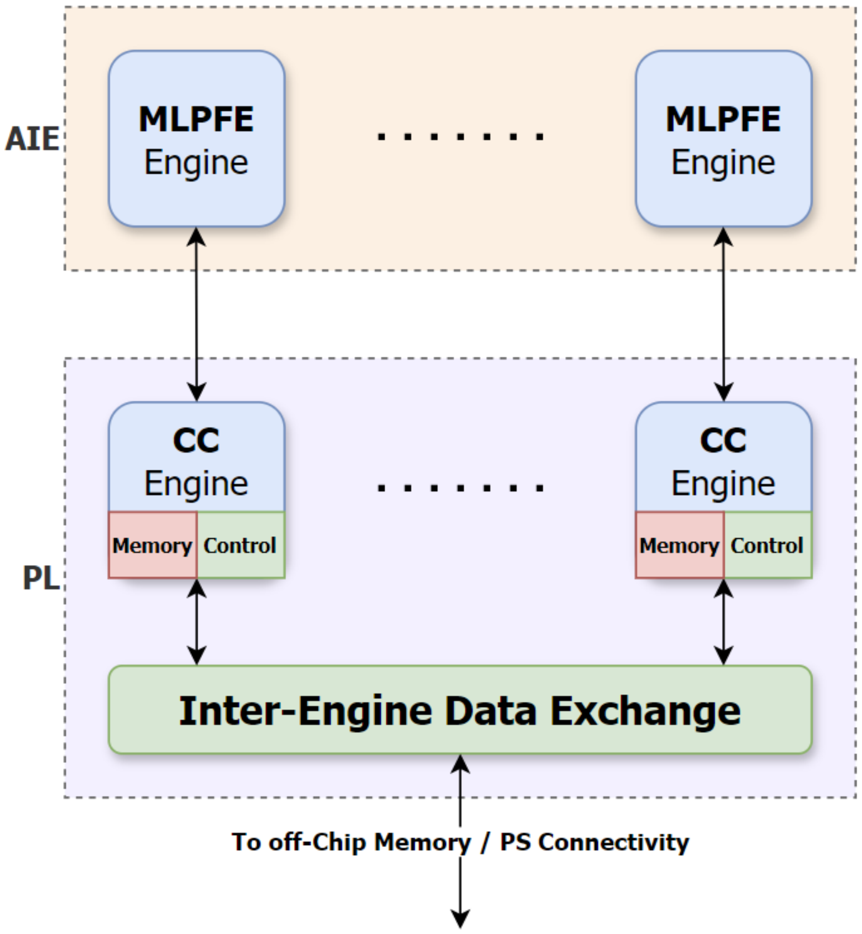


Figure 3.2: Design 1 (By-Center Parallelization) Block Diagram

This design can potentially use some more advanced features to increase its performance:

1. As the number of centers in an NMM increases, the memory needed to store all the state variables and their history will also increase. In order to tackle this challenge, we can store the state variables that are not immediately needed, meaning the centers with relatively large $T_{:j\ min}$ value, on the off-chip memory and bring them back to the device when they are needed. This will save the on-chip memory for variables that are immediately needed and reduce the memory need of the design.
2. One of the bottlenecks of this design is the data transfer between multiple CC engines. Since the dependencies and data access patterns for each center are known before the start of the simulation and it is fixed throughout the simulation, the Inter-Engine Data Exchange block can pre-fetch the data needed by each engine from other engines to save time.
3. To tackle the bottleneck of the data transfer between multiple CC engines, pre-processing can be done to divide the centers among the engines in a way that reduces the data dependency of engines on each other.
4. If the memory capacity limitations allow, to reduce the inter-engine data accesses, we can store a copy of all the needed values by an engine in the engine itself.

Requirement Check

With the assumption of partitioning the design according to Figure 3.2, the number of engines of 32 ($E = 32$), and that each MLPFE engine is implemented in a single AI Engine tile, we can check the performance and memory capacity requirements of this design. We use the performance and memory capacity models derived in Section 3.1, and information provided in Section 2.6 (such as the memory capacity in the PL and the AIE in addition to the computation performance in the AIE) to compare the requirements against the resources and capabilities of the device. Equations 3.10, 3.11, and 3.12 show the MLPFE memory capacity, MLPFE performance, and CC memory capacity respectively. The performance of the CC engine computation performance is heavily implementation-dependent and it is not considered here.

$$\frac{\text{Mem}_{\text{MLP+FE}}}{E} < 32,768 \qquad \text{[Bytes]} \quad (3.10)$$

$$\text{Comp}_{\text{MLP+FE}} < 10^{10} \qquad [\tfrac{\text{MAC}}{\text{s}}] \quad (3.11)$$

$$\text{Mem}_{\text{CC}} < 20,500,000 \qquad \text{[Bytes]} \quad (3.12)$$

Additionally, we can look into the bandwidth requirement for the MLPFE-CC engines link. In this design, the coupling input values of the centers are calculated in the CC engines (PL), then it is transferred to the MLPFE engines (AIE) in order for the next state variable values to be calculated, and in the end, the calculated state variables are transferred back to the PL. These data exchanges between the PL and the AIE happen through the direct connection between the two fabrics.

At each timestep, N coupling values are calculated in the CC engines and are transmitted to the MLPFE engines. Additionally, NM newly calculated state variables are transferred from the MLPFE engines back to the PL, to go to the CC engines and the main memory. As mentioned in Section 2.6, there are 39 different AXI4-Stream interfaces available between the AIE and the PL, each with a bandwidth of $24\ \frac{GB}{s}$ (from AIE to PL) and $32\ \frac{GB}{s}$ (from PL to AIE). The time it takes to transfer the required data from the PL to the AIE and the other way around can be calculated as follows:

$$\text{Time}_{\text{PL-AIE}} = \frac{4N}{39 \times 32 \times 10^9} \qquad \text{[s]} \quad (3.13)$$

$$\text{Time}_{\text{AIE-PL}} = \frac{4NM}{39 \times 24 \times 10^9} \qquad \text{[s]} \quad (3.14)$$

Furthermore, the Inter-Engine Data Exchange block should also be capable of accommodating high-bandwidth data transfer among the CC engines. We assume the worst-case an all-to-all connected NMM. In this case, each CC engine requests $N - \dfrac{N}{E} = (E-1)\dfrac{N}{E}$ values through the data exchange block. As a result, we can formulate the amount of data that is needed to be transferred through the Inter-Engine Data Exchange block at each timestep as follows:

$$4 \times E \times (E-1)\frac{N}{E} = 4N(E-1) \qquad \text{[Bytes]} \quad (3.15)$$

If we set a time limit where these data transfers have to happen within that period, we can calculate the bandwidth required by the inter-engine data exchange. We define this time-limit as h_{IEX} where $0 < h_{IEX} < h$ and write the required bandwidth as follows:

$$\text{BW}_{\text{IEX}} : \frac{4N(E-1)}{h_{IEX}} \qquad [\tfrac{\text{Bytes}}{\text{s}}] \quad (3.16)$$

In order to check the requirements against the capabilities of the device, we assume some ranges of values for different parameters of the problem and use the worst-case values to verify the feasibility of this design. These values are summarized in Table 3.1.

Parameter	**Estimated Range/Value**	**Reason**
# of Centers (N)	50 - 1000	Largest TVB dataset [61]
# of State Variables/Center (M)	2 - 10	Typical models [60, 10, 32, 1]
# of Engines (E)	32	-
# of MLP Hidden Layers (H)	1- 5	Typical MLP size for evaluating centers' local dynamics [9]
# of Neurons/Hidden Layer (L)	16 - 64	Typical MLP size for evaluating centers' local dynamics [9]
Maximum Delay (T_{max})	100 - 1000	Size of the brain, typical signal propagation speed, and average simulation timestep
Real-time Limit (h_{RT})	100 - 1000 (μs)	Average simulation timestep [44, 39]
Data Exchange Time Limit (h_{IEX})	10 - 1000 (μs)	A fraction of the average simulation timestep

Table 3.1: Range estimations of the parameters

If we assume the worst-case values for each of the parameters mentioned in Table 3.1, we can rewrite the Equations 3.10, 3.11, and 3.12 as follows:

$$\text{Mem}_{\text{MLP+FE}} \text{ Requirement (Bytes): } 71,976 + 2625 + 1024 < 32,768 \rightarrow 76,625 < 32,768 \times$$

$$\text{Comp}_{\text{MLP+FE}} \text{ Requirement } (\frac{\text{MAC}}{\text{s}}) \text{: } 10^4 \times 31.25 \times 17,694 < 10^{10} \rightarrow 0.55 \times 10^{10} < 10^{10} \checkmark$$

$$\text{Mem}_{\text{CC}} \text{ Requirement (Bytes): } 4,000,000 + 8,000,000 < 20,500,000 \rightarrow 12,000,000 < 20,500,000 \checkmark$$

We can see that a single AI Engine tiles memory is not enough to hold the largest MLP that will be used in the NMMs. Either the workload of a single MLP should be divided into multiple AI Engine tiles or smaller models with fewer hidden layers or neurons per layer be used to be able to fit all of their parameters in the memory of a single AIE tile.

For the AIE-PL interface bandwidth required by this design, we assume the same data exchange time limit we assumed for the Inter-Engine Data Exchange (h_{IEX}). Using this value, we can check the requirements for the PL-AIE link bandwidth mentioned in Equations 3.13 and 3.14 as follows:

$$\text{Time}_{\text{PL-AIE}} \text{ Requirement (µs): } = 0.0032 < 10 \checkmark$$

$$\text{Time}_{\text{AIE-PL}} \text{ Requirement (µs): } = 0.0427 < 10 \checkmark$$

It can be seen that the AIE-PL interfaces are more than capable of accommodating the bandwidth demands of this design.

In terms of the required performance by the Inter-Engine Data Exchange block, if we rewrite Equation 3.16 with the worst-case scenario values from Table 3.1, we can see that the Inter-Engine Data Exchange block should provide a performance of around $12.4\frac{GBytes}{s}$. If we assume a typical $250MHz$ clock frequency for this block, the Inter-Engine Data Exchange has to transfer around 50 Bytes of data per clock cycle. With these performance requirements, the Inter-Engine Data Exchange block can potentially become a major bottleneck of the design. This bottleneck can come from both its own performance and its communication bandwidth with the CC engines.

Furthermore, when the number of centers (N) is fixed, as the number of engines (E) increases the performance required by the Inter-Engine Data Exchange also increases. On the other hand, when the number of engines increases, the value of $\frac{N}{E}$ decreases, and this lowers the amount of computation needed by each MLPFE and CC engine. This means that at small values of E, the computation performance of the MLPFE or the CC engines would be the bottleneck of the application, and at larger values of E the bandwidth of the Inter-Engine Data Exchange is the main bottleneck. This means that having more and more engines doesn't necessarily increase the performance, and choosing the number of engines correctly (or in other words, setting the value of $\frac{N}{E}$ correctly) is very important in achieving the best performance this design can offer.

3.2.2. Design 2: By-Delay Parallelization

Introduction

In this design, the calculated state variables enter a flow of coupling calculations. The main difference between this design with the previous design discussed in Section 3.2.1 is the way couplings are calculated. With the general coupling calculation being challenging in terms of memory needs and access patterns, this design addresses this challenge by efficiently moving the data. The main idea of this design is to group 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 3.17 and 3.18. 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.

$$C_i(t) = K_{post}\left(\sum_{j=1}^{N} W_{ij}K_{pre}\left(X_j(t-d_{ij}), X_i(t)\right)\right) = K_{post}\left(\sum_{n=1}^{CCE_N} C_{in}(t)\right) \tag{3.17}$$

$$C_{in}(t) = \sum_{j=1}^{N} W_{ij}K_{pre}\left(X_j(t-d_{ij}), X_i(t)\right) \textbf{ For } d_{ij} \in [D_{n-1}, D_n) \tag{3.18}$$

The details of the design are as follows:

1. Just like design 1, this design also has MLPFE and CC engines responsible for MLP/Forward Euler evaluation and coupling calculation respectively. The MLPFE engines and the way they work are more or less the same as design 1. On the other hand, the process of calculating the couplings is different in this design.
2. 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 T. We take all the unique values in T, sort them, and divide the calculation points associated with those delay 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.
3. 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 contributes 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 simulation 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.

4. 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.
5. Within a CC engine, the calculation of the couplings is a combination of parallel and time-multiplexing. The exact structure of the engines depends on their implementation.
6. This design eliminates one of the possible bottlenecks of coupling calculation of design 1 which was the Inter-Engine Data accesses. In this design, the data flows through a series of engines and are used whenever they are necessary. However, instead of the Inter-Engine Data Exchange block, in this design, the values calculated in different CC engines have to be added and organized. Here, the coupling input of a certain center i is calculated in parts in multiple engines. These values should be gathered together, accumulated, organized, and then sent accordingly to the corresponding MLPFE engine.

Since unlike the control-heavy coupling calculation process in design 1, the coupling calculation in this design has a dataflow style, it is suited for implementation on both the PL and the AIE. Figure 3.3 shows the general block diagram of this design.

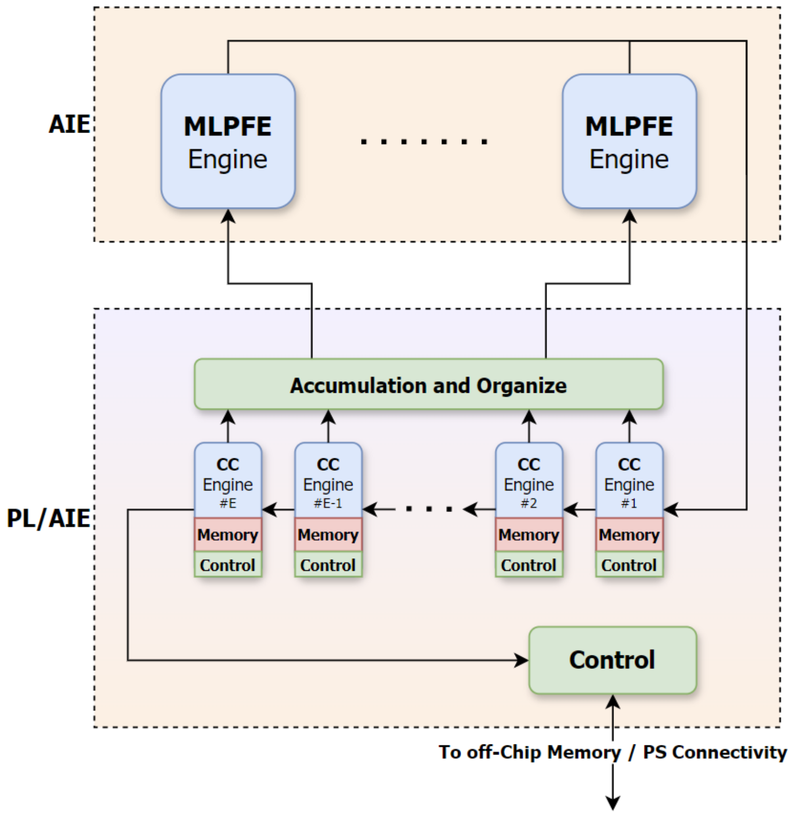


Figure 3.3: Design 2 (By-Delay Parallelization) Block Diagram

There is room for performance improvements in this design. Some ideas are as follows:

1. In order to save on memory use, bypass lanes can be implemented between non-adjacent engines, and from engines to the off-chip memory. These bypass lanes can prevent unneeded data in a certain engine from being stored in that engine, in addition to the data that are no longer needed in the simulation not being stored on the on-chip memory anymore.
2. Pre-processing can be performed to balance the load of each CC engine in terms of calculation and memory use.
3. Within the CC engines, multiple sub-engines can work in parallel to further divide the workload of the delays. Within the sub-engines, the different couplings are calculated by time-multiplexing.

Requirement Check

For this design, the requirements regarding the computation performance and memory capacity of the MLPFE engines, in addition to the AIE-PL link bandwidth are the same as design 1 discussed in Section 3.2.1. Since the coupling calculation engines can be implemented in either PL or the AIE, we will discuss the performance and memory requirement of the CC engines in both cases.

If the CC engines are implemented in the PL, the memory and computation requirements for them are similar to the ones discussed in Section 3.2.1. As mentioned before, the accumulator block is responsible for adding the calculation results from the CC engines together. At each timestep, we can formulate the amount of computation needed for the accumulation task as $(E-1)\times N$. This is because there is a calculated value corresponding to each center of the NMM in each of the CC engines and they are needed to be added together.

If we assume a time limit of $h_{IEX} = 10\mu s$ for this task to be done, and a clock frequency of $250MHz$ for the Accumulator block, we can write the required computation performance of this block using the worst-case values from the Table 3.1 as follows:

$$\frac{1}{h_{IEX}} \times \frac{(E-1)N}{Clock\ Frequency} = 12.4 \qquad [\tfrac{\text{Additions}}{\text{Clock Cycle}}] \tag{3.19}$$

Since the reduction operation that the Accumulator block is performing is highly parallelizable, achieving 12.4 addition operations per clock cycle is feasible.

If the CC engines are implemented in the AIE, we should look into the performance and memory capabilities of the AIE and compare it against the requirements of the coupling calculation process in this design. The memory capacity and the computation performance of coupling calculation in this design, in this case, are formulated and compared in Equations 3.20 and 3.21 respectively. We assume that the computation workload of the coupling calculation is evenly distributed among the CC engines, however, the distribution of the workload heavily depends on the simulation parameters.

$$\text{Mem}_{\text{CC}}\ \text{Requirement (Bytes)}: 4N \times \frac{T_{max}}{E} + 8 \times \frac{N^2}{E} < 32,768 \rightarrow 375,000 < 32,768\ \times \tag{3.20}$$

$$\text{Comp}_{\text{CC}}\ \text{Requirement } (\frac{\text{MAC}}{\text{s}}): \frac{1}{h_{RT}} \times \frac{N^2}{E} < 10^{10} \rightarrow 0.3125 \times 10^{10} < 10^{10}\ \checkmark \tag{3.21}$$

It can be seen the main restriction of CC engines being implemented on the AIE is the memory capacity. Although an all-to-all connected NMM is an unrealistic assumption, Equation 3.20 shows that the memory capacity has to be given special consideration if this design is implemented with CC engines residing in the AIE.

3.2.3. Design 3: Tile-Based Scheduling

,

This design is inspired by [21], and unlike the first two, the simulation of different timesteps can be evaluated in parallel. First, we discuss the principle behind this design. [21] argues that by using bandwidth-reducing algorithms, like Reversed Cuthill-McKee (RCM), we can re-order the rows and columns of the connectivity matrix (W) in a way that newly indexed centers depend on the history of the set of centers from a limited range of indices. With this re-ordering, only the elements close to the diagonal of the matrix would have a non-zero value. Additionally, the RCM algorithm keeps the short delays closer to the diagonal. We refer to the re-ordered connectivity matrix as W_b.

Now we look at our problem of solving the differential equation as a two-dimensional iteration space, where one dimension is the center number (which corresponds to a row in W_b), and the other dimension is the timestep. As can be seen in Figure 3.4, the value of state variables of a center at timestep t depends on its value at timestep $t-1$ and the value of other neighboring state variables at timesteps before $t-1$.

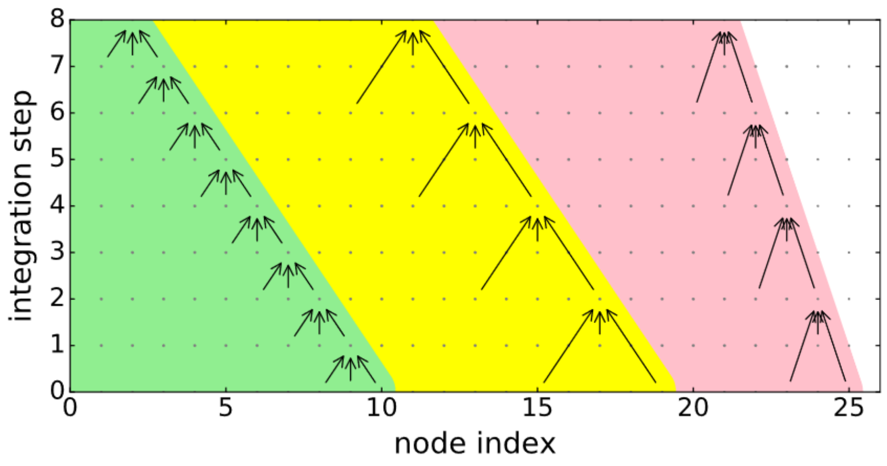


Figure 3.4: Dependency pattern in the iteration space [21]

The main idea for this design is to create tiles in the iteration space that can be executed in parallel. In order to do so, [21] suggests hexagonal-shaped tiles across the iteration space. Figure 3.5 shows the proposed tiling strategy given in [21]. The green tiles are done with their computation, the yellow tiles are in the middle of their computation, and the triangle shows the data dependency of a certain center at a certain timestep.

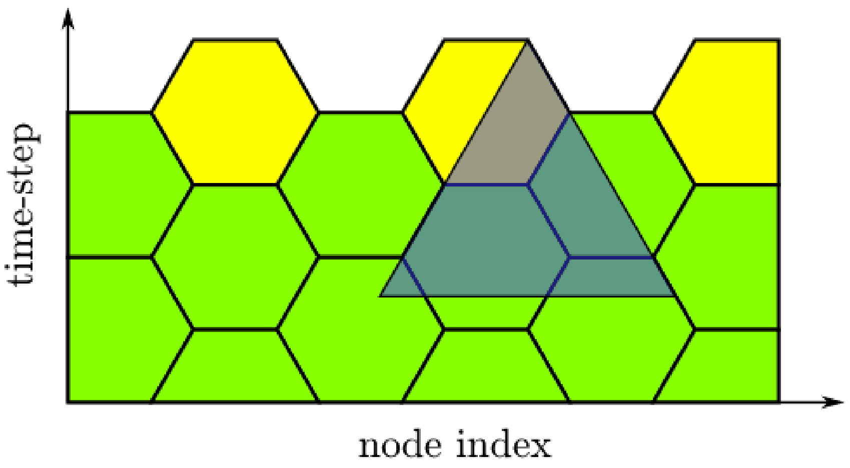


Figure 3.5: Hexagonal-shaped tiling [21]

Each tile only has data dependency on already computed tiles, and tiles in the same row can be executed at the same time. [21] discusses the algorithm for how to form these tiles and how to schedule them.

The details of the design based on the tiling scheduling idea are as follows:

1. Before the start of the simulation, pre-processing is done to create the tiling and scheduling of them. This scheduling would be based on the connectivity matrix and the resources available.
2. The design consists of E *Tile-Based Solvers*. Each Tile-Based Solver is responsible for solving the values associated with 1 tile at a time. Each Tile-Based Solver consists of an MLP, a coupling calculation, and a Forward Euler solving unit. Based on the implementation, these units can work in parallel to increase the performance of the design.
3. When an engine is assigned to calculate the values of a tile, a control unit loads all the data needed by the engine to its dedicated memory from the off-chip memory. After all the data are provided, the engine starts the calculation of the values of the assigned tile.
4. No data communication would happen between the tiles. All the tiles would have all the data needed by them for the calculation in their memory.
5. To reduce memory use, some data-sharing schemes between the engines can be implemented for data points that are needed by more than one engine. Similarly, caching mechanisms can also be included to reduce the off-chip memory accesses even further.

The Tile-Based Solvers can reside entirely on the PL, on the AIE, or be partially residing on both of them. Memory-heavy parts of the solver can reside in the PL, while control-heavy parts of the application can be assigned to the AIE to benefit from higher clock frequency. Figure 3.6 shows the general block diagram of this design.

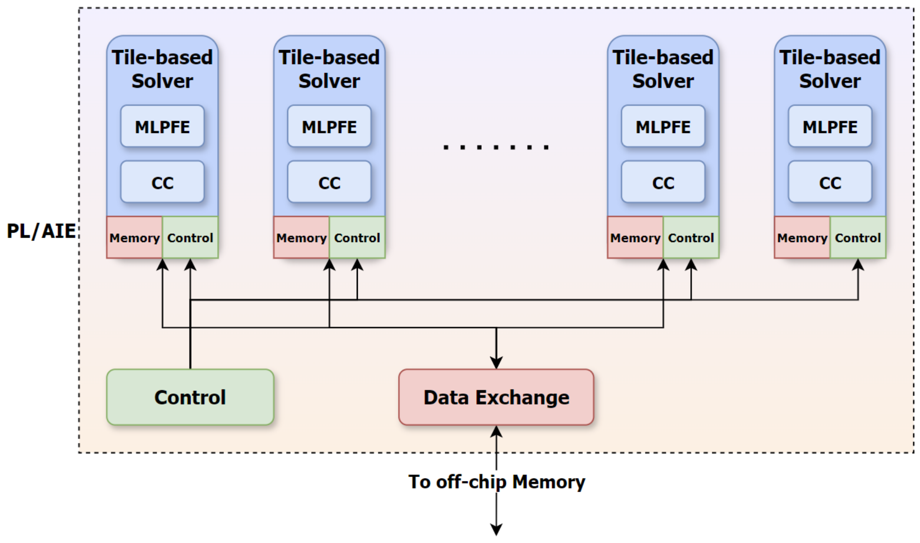


Figure 3.6: Design 3 (Tile-Based Scheduling) Block Diagram

Requirement Check

In terms of memory, the analysis for this design is similar to the ones presented in Sections 3.2.1 and 3.2.2. In terms of computation, the amount of calculations needed by each solver depends heavily on the system input. In reality, the tiling algorithm might produce smaller tiles, meaning the amount of calculations per Tile-Based Solver is not that large. However, this puts the bottleneck on the Data Exchange block which is required to transfer the data needed by the solvers at a higher rate. If the algorithm produces larger tiles, the pressure would be on the Tile-Based Solvers, as they are required to perform a large amount of computation for a single instance of input. An interesting question of how many solvers to put in this design rises, as more solvers mean faster calculation but also means more pressure on the Data Exchange to feed these solvers.

Due to the strong dependency of the performance of this design on the input, a more detailed analysis of the computation performance would not be studied here.

3.3. Final Architecture

In this section, we compare the pros and cons of each design and decide on a final design to implement. The designs are compared with different metrics in mind, ranging from their potential performance to how well they map to the device. Table 3.2 summarizes the calculations performed earlier in Sections 3.2.1 and 3.2.2 for the first and second designs. To see the effect of different simulation parameters on the analyzed metrics, the big-O notation of the memory capacity, computation performance, and connection bandwidth are included in this table. As mentioned earlier in this chapter, and as seen in Table 3.2, both designs 1 and 2 might encounter limitations with MLP and FE Solver memory capacity in the AIE. Additionally, if coupling calculations in design 2 are implemented in the AIE, the memory capacity would be a limiting factor according to the performed analysis.

Metric	Big-O Notation	Design 1: By-Center Parallelization	Design 2: By-Delay Parallelization	Requirement
Mem$_{\text{MLP+FE}}$ (KBytes)	$\mathcal{O}(N+M+E +L^2+H)$	$E \times 76.6$	$E \times 76.6$	< $E \times 32.8$
Comp$_{\text{MLP+FE}}$ ($\frac{\text{MAC}}{\text{s}}$)	$\mathcal{O}(N+M+\frac{1}{E} +L^2+H)$	0.55×10^{10}	0.5×10^{10}	< 10^{10}
Mem$_{\text{CC}}$ (KBytes)	$\mathcal{O}(N^2+T_{max})$	- (AIE) $12,000.0$ (PL)	$E \times 375.0$ (AIE) $12,000.0$ (PL)	< $E \times 32.8$ (AIE) < $20,500.0$ (PL)
Comp$_{\text{CC}}$ ($\frac{\text{MAC}}{\text{s}}$)	$\mathcal{O}(N^2)$	- (AIE) - (PL)	0.3×10^{10} (AIE) - (PL)	< 10^{10} (AIE) - (PL)
Time$_{\text{PL-AIE}}$ (s)	$\mathcal{O}(N)$	3.2×10^{-9}	3.2×10^{-9}	< 10^{-5}
Time$_{\text{AIE-PL}}$ (s)	$\mathcal{O}(N+M)$	4.27×10^{-8}	4.27×10^{-8}	< 10^{-5}
Scatter/Gather	$\mathcal{O}(N+E+\frac{1}{h_{IEX}})$	$12.4 \ \frac{\text{GBytes}}{\text{s}}$	$12.4 \ \frac{\text{Additions}}{\text{Clock Cycle}}$	-

Table 3.2: Memory Capacity, Computation Performance, and Bandwidth Requirements of Designs 1 and 2 Considering the Worst-case Scenario

Design 1 implements probably the most straightforward parallelization for the problem. The main negative point of this design is the control-heavy nature of its coupling calculation process. In the worst-case scenario, the data exchange between the CC engines required around $12.4\frac{GBytes}{s}$ of throughput. Additionally, the scalability of this design might be challenging due to the fact that increasing the number of engines also causes the required throughput of the data exchange process to increase. On the other hand, this design's partitioning of the problem is good enough to meet most of the performance and memory capacity requirements.

Design 2 changes the control-heavy nature of coupling calculations in Design 1 to a dataflow-style architecture. This helps mitigate the bottleneck of data exchange between the engines that exist in Design 1. However, if the CC engines in this design were implemented in the AIE to make use of the higher frequency and shorter development time, the memory capacity would become a major challenge.

Design 3 proposes a different approach to solving the problem, parallelizing also across the timesteps of the simulation. This approach, although interesting, makes the performance of the design heavily dependent on the simulation inputs. If the connectivity matrix is not structured in a way that can be changed to a band matrix, then the tiling algorithm produces very large tiles resulting in a close-to-sequential performance. In other words, this design faces the challenge of how general it is.

With all the analysis and the comparisons done, it was decided to move forward with **Design 2: By-Delay Parallelization** for implementation. This design could map to the Versal ACAP very well, and could potentially deliver a significant performance and power efficiency.

4 Implementation

In this chapter, we dive into the implementation details and gradual improvements of the system. As mentioned in Section 3.3, the design that parallelizes the coupling calculation by delay was chosen to be implemented. The chosen design is shown in Figure 3.3, having the option to implement the CC engines in the AIE or the PL. We decided to implement both versions of this design: 1) A system in which all of its components are implemented in the AIE (AIE-Only), and 2) A system in which the coupling calculation part of it is implemented in the PL (Heterogeneous). This is done for the following reasons:

1. To familiarize myself with the newly developed AI Engines and the tools associated with them, especially for someone new to the Versal ACAP device.
2. To gain experience working with a truly heterogeneous system, and to see what goes into the design, realization, and tooling of such systems and also the application that would be implemented on it.
3. To benefit from the fast development cycle of the AI Engines.
4. To asses how the AIE performs against the PL in the coupling calculation workloads.
5. To observe how this platform addresses the bottleneck of data movement between different parts of a heterogeneous system.
6. The AIE-only version of the system can be mapped to the AMD XDNA™ architecture [2], which is available on other AMD devices including some Ryzen CPUs.

For the remainder of this chapter, Section 4.1 dives into the details of the AIE-only system implementation and what improvements were performed. Section 4.2 discussed the implementation of the Heterogeneous system that uses both the AIE and PL, and Section 4.3 presents the concluding remarks regarding the implementation of the system.

4.1. AIE-Only System

As mentioned in Section 3.2.2, the implemented design uses dataflow-style coupling calculations. This makes this process to be appropriate to be mapped to the AIE of the Versal ACAP. This version of the system where both the MLPFE and CC engines are implemented in the AIE is referred to as the **AIE-Only System**. In this section, we go through the implementation of the AIE-only system, and the evolution stages that were taken to arrive at the final implementation.

4.1.1. Basic Implementation

Description

In the basic AIE-only implementation, a total number of 16 CC engines, 32 MLP engines, and 1 Forward Euler Solver (FES) engine are used. Each of these engines is hosted on a single AIE tile. The MLP calculation workload of the centers is divided equally among the 32 MLP engines, and also the coupling calculation workload is divided among the 16 CC engines. All of these engines are running in parallel to each other, and when they are done with their calculation, they send their results to the FES engine to calculate the next step state variables.

Because there are limitations on the number of input streams for each AIE tile, not all MLP engines can send their results directly to the FES. Additionally, the CC engines can't send their calculations directly to the FES engine to be added together either. That is why reduction trees were used to combine all the streams coming from the MLP and CC engines together. Figure 4.1 shows the block diagram of this implementation.

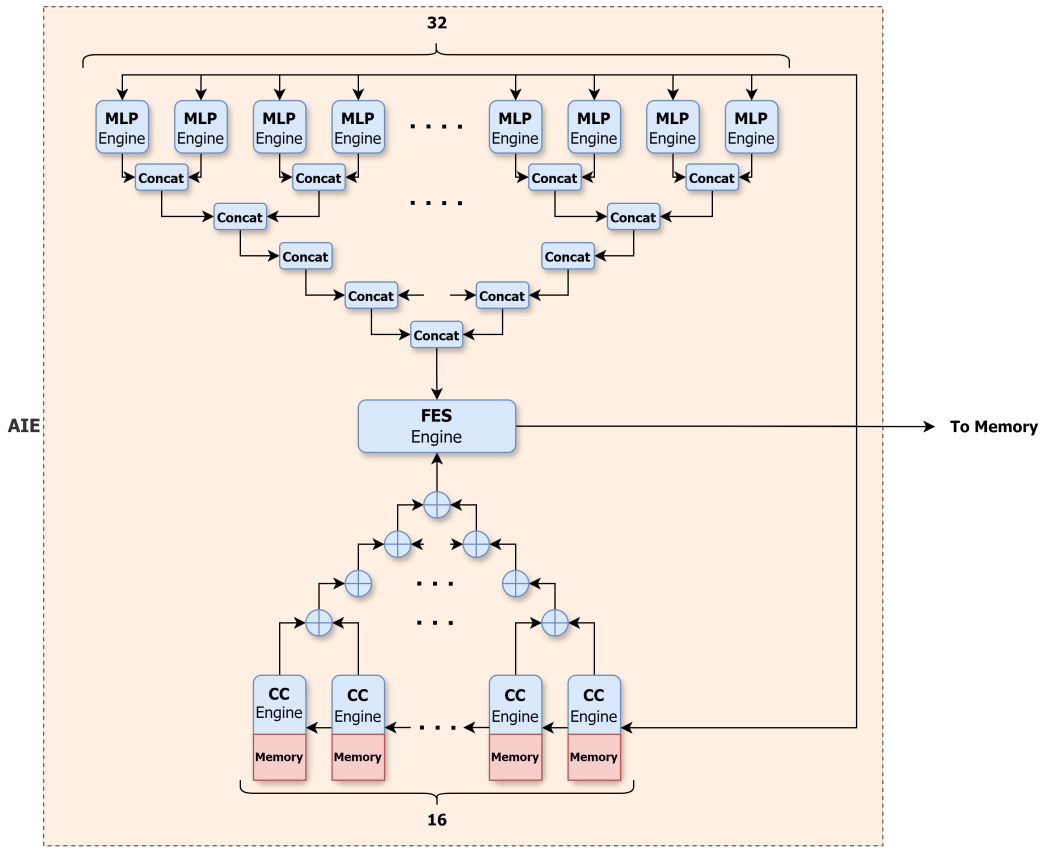


Figure 4.1: Block Diagram of the Basic Implementation of the AIE-Only System

As can be seen in Figure 4.1, a concatenation tree is implemented in the basic implementation to put the outputs of all MLP engines together and feed them to the FES engine. The same tree structure is implemented to add the values calculated by the CC engines.

Bottlenecks and Possible Improvements

The main bottlenecks of this implementation are the trees used to reduce the outputs of the MLP and CC engines. Both the concatenation and reduction trees used for MLP and CC engines respectively introduce a large amount of latency. The latency of these trees is so high that in NMMs with a smaller amount of centers, the majority of time is spent on the execution of these trees. Additionally, this bottleneck prevents us from scaling up this implementation with more MLP and CC engines, because the more engines we have, the deeper these trees get, and more latency is introduced to the system. Addressing this bottleneck should be the priority of the improved version of the system.

Furthermore, when the number of centers increases, the workload of the MLP engines also increases linearly. In this implementation, all of the calculations within the engines are scalar and no vectorization is performed. Since MLP engines are nothing but matrix multiplications, vectorizing these engines is very efficient and can help reduce their calculation time significantly.

4.1.2. First Improved Implementation

Description

As mentioned in Section 4.1.1, the main bottleneck of the basic implementation is the concatenation and reduction trees. In the first improved version, these trees are replaced with *Packet Switching* constructs. The AIE provides Packet Switching as a solution to the limitation on the number of input/output streams to each tile. In Packet Switching, two concepts of *Merging* and *Splitting* are used. In merging, multiple streams of data can share a single physical stream to send AXI4-Stream packets. In splitting, a single physical stream can be divided into multiple streams of AXI4-Stream packets. The difference between packet split and broadcasting is that in packet split the packets are routed to the desired destination and are not sent to all the connected destinations.

In the first improved version of the AIE-only implementation, we replaced the reduction and concatenation trees with packet merge constructs. Additionally, the separate FES engine was removed and its functionality was added to the MLP engines forming the MLPFE engines. Each MLPFE engine is responsible for calculating the local dynamics of a group of centers (MLP evaluation), receiving the couplings from the CC engines, and producing the new state variables (Forward Euler solver). The functionality of the CC engines remains the same as in the previous implementation. After experimenting with different configurations of packet merge, the system shown in Figure 4.2 with 32 MLPFE engines and 16 CC engines performed the best.

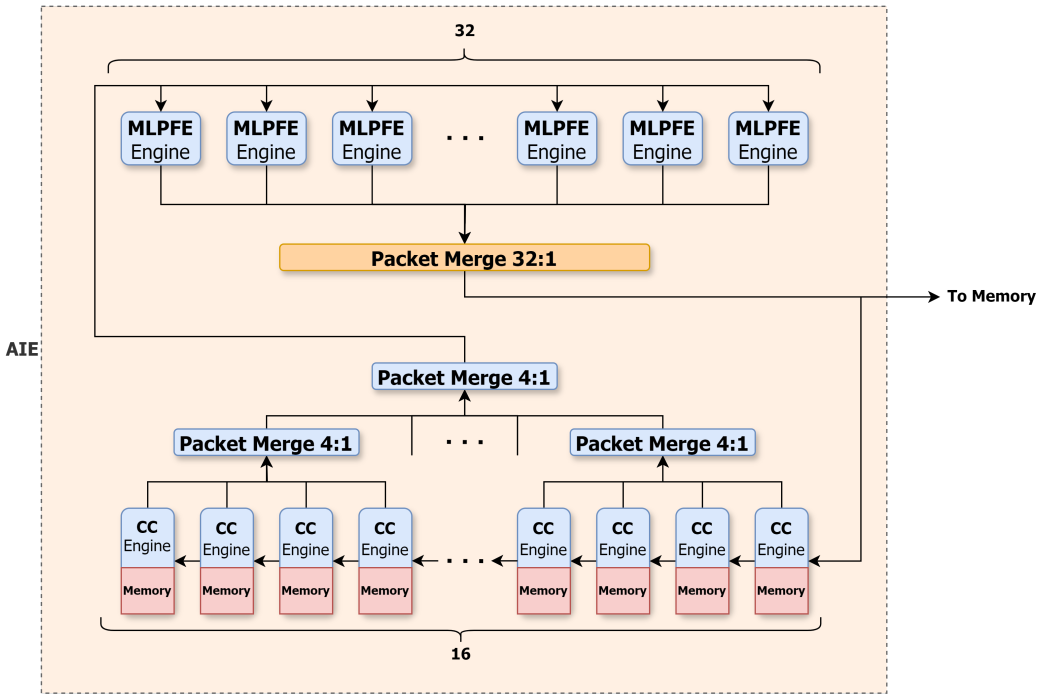


Figure 4.2: Block Diagram of the First Improved Implementation of the AIE-Only System

There is a difference between the packet merging structure for the MLPFE engines and for the CC engines. This is due to the fact that the output streams of the MLPFE engines are low bandwidth streams that are needed to form a higher bandwidth stream, but the output streams from the CC engines have a higher bandwidth and form a stream with the same bandwidth. This difference leads us to break up the 32:1 packet merging for the CC engines and implement it as a tree of 4:1 packet merges.

Bottlenecks and Possible Improvements

The first improved implementation performs 30% better compared to the basic AIE-only implementation with the same benchmark. This shows the packet-switching constructs work better compared to reduction trees. However, after profiling this implementation, we can see the bottleneck of this design is the reduction process associated with the CC engines, and this is the area that requires improvements.

Each 4:1 packet merge block has to receive 4 packets of N values and add them together to produce a single packet of N values at the output. Due to an imbalance in the CC engines' amount of calculations, meaning some have to do more calculations than others, the input packets to the first level of packet merge blocks arrive at different times, introducing delays within the reduction process. Using packet merge instead of a tree (like the basic implementation) is more scalable, however, if we increase the number of CC engines, increasing the number of packet merge blocks would be resource-heavy which can lead to time-consuming or even impossible place and routing processes.

4.1.3. Second Improved Implementation

Description

As mentioned in Section 4.1.2, the main bottleneck of the first improved implementation is the reduction process of the calculated couplings. In the second improved AIE-only implementation, the packet merge blocks that are connected to the CC engines are replaced with *Cascade Stream* connections that exist within the AIE tiles. As mentioned in Section 2.6, in addition to AXI4-Stream and adjacent memory connections, AIE tiles can transfer data to other tiles using cascade streams. The cascade stream is a 384-bit wide connection chaining AIE tiles together unidirectionally. The green arrow in Figure 4.3 shows how the cascade stream connects AIE tiles together. This connection is 384 bits wide and is capable of transferring the whole width in a single clock cycle.

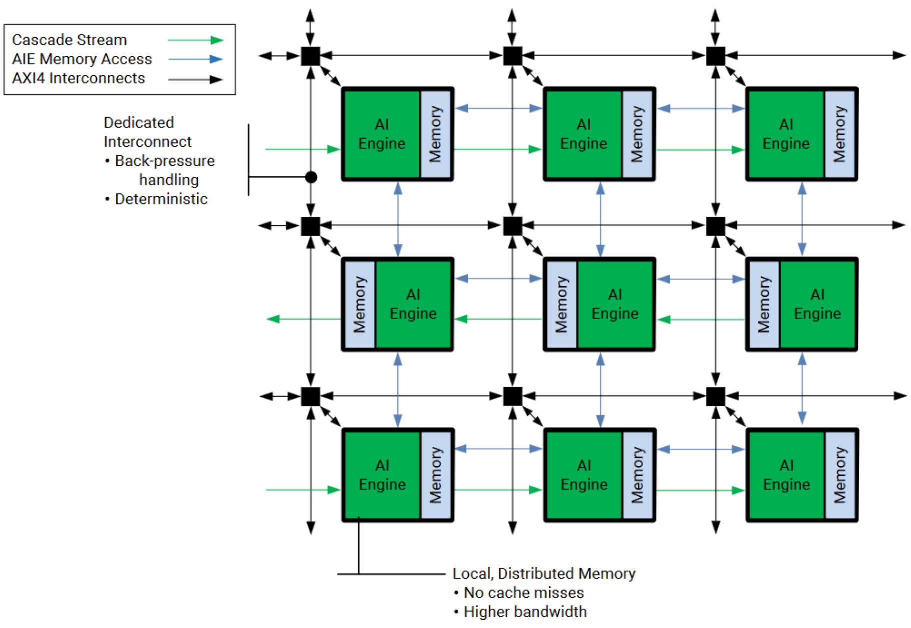


Figure 4.3: Cascade Stream Connections Between AIE tiles (Shown in green) [3]

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 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. However, the first engine has to wait for the input from the MLPFE engines to be able to calculate the couplings.

Using the cascade stream and the above-mentioned optimization, the second improved implementation shown in Figure 4.4 was implemented.

Some details regarding this implementation are as follows:

1. The calculation part of the MLPFE and the CC engines happens more or less in parallel which helps improve the performance of the system.
2. A cascade stream chain (the green arrows in Figure 4.4) 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.

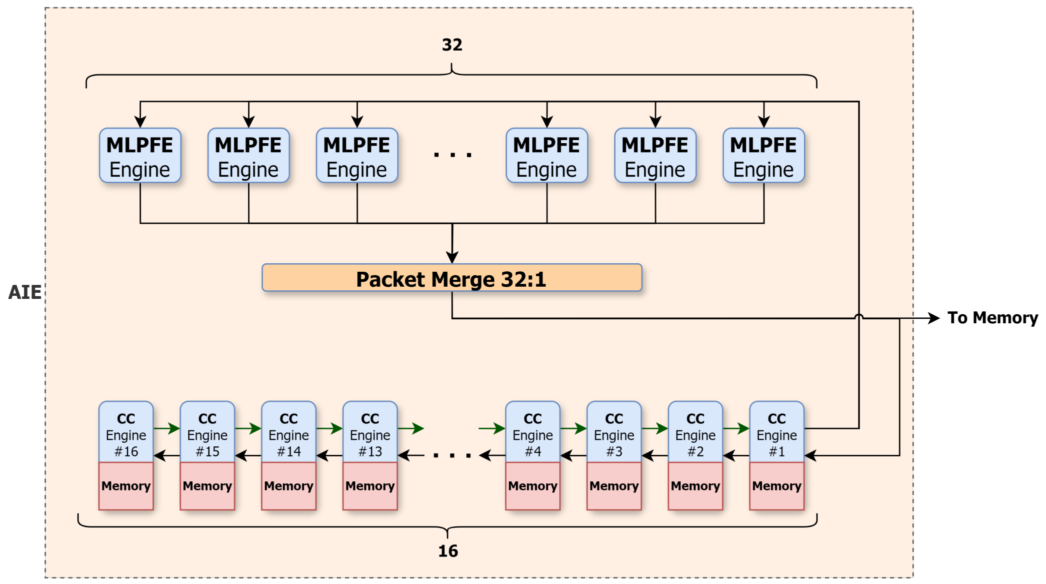


Figure 4.4: Block Diagram of the Second Improved Implementation of the AIE-Only System

3. With this design, the coupling calculation processes of the CC engines, the coupling data transfer, and the accumulation of the coupling data overlap to a good extent.
4. If we decide to increase the number of CC engines, this implementation would have very little overhead compared to the first 2 implementations. In other words, the second improved implementation is much more scalable than the basic and the first improved implementations.

Bottlenecks and Possible Improvements

After benchmarking the second improved implementation with 16 CC engines, we see a 20% improvement in the runtime of the simulation. This shows that the cascade stream and other optimizations performed on the system helped reduce the overhead of the accumulation process of the CC engines. As mentioned earlier in this section, the MLPFE and CC engines run more or less in parallel. This means that the bottleneck of the system would be either the calculations of the MLPFE engines or the calculations of the CC engines. After profiling, we can see the bottleneck of this implementation still resides in the coupling calculations. In order to better understand the nature of the bottleneck, we can take a look at what happens inside the MLPFE and CC engines.

In the MLPFE engines, after receiving the initial states, each engine starts calculating the MLP result for the centers that it is responsible for. For each center, the engine performs $2(L \times M) + (H-1)L^2$ floating-point MAC operations. For each MAC operation, the engine loads three values and writes back one value. After the MLP calculations are done, the engines wait for the coupling inputs from the CC engines to be received, and then the Forward Euler solver calculations are performed. After the solver calculations are done, the results are sent to the first CC engine and also to the off-chip memory and the MLPFE engines start calculating the MLP results for the next timestep.

In the CC engines, after receiving the connectivity data associated with each engine for the calculations, the engine enters a loop. In that loop, it first exchanges the state variable data with the neighboring engines, then it starts the calculation for the couplings. For each calculation, each CC engine performs three floating-point operations and on average, reads about eight values and writes back one value. After all the calculations are done, the engine reads the values from the neighboring engine on the cascade stream, adds its own results to the read values, and outputs the result on the cascade stream to the next engine on the chain. In comparison with the MLPFE engine, although the CC engines might need to perform fewer calculations, they need to access more values from the memory for their calculation. Additionally, the MLPFE engines perform more structured memory accesses and calculations, making it a perfect candidate for compiler optimization (for example software pipelining).

To profile the bottleneck of the second improved implementation, we can look at the number of centers (N), the size of the MLP ($MLPS$), and the sparsity of the connectivity matrix (SP). The total amount of calculation needed for the coupling input is proportional to $(1 - SP) \times N^2$ and for the MLP evaluation is proportional to $MLPS \times N$. While the MLP workload is divided equally between the MLPFE engines, the amount of workload assigned to each CC engine depends on the distribution of the values in the delay matrix. As mentioned in Section 3.2.2, we divide the range of $[0, T_{max}]$ equally between E CC engines. In this case, an engine may be assigned with half of the total coupling calculation workload (because there are a lot of connections with delays in the range assigned to this engine) while another engine is assigned with no workload. Some processing can be done to assign non-equal delay ranges to the CC engines to reduce the amount of imbalance in the workload, however, memory capacity limitation for each CC engine prevents us from achieving a suitable distribution of the coupling calculation workload among the CC engines. In other words, if we assign a non-equal delay range assignment to the CC engines, the engine with less amount of workload assigned needs to hold more history values and the $32KBytes$ memory limitation of each AIE tile does not allow for this flexibility. Additionally, this uneven workload distribution challenges the scalability of this system, meaning that it is possible by adding more CC engines the performance would remain the same. More discussions regarding these issues are presented in Section 5.2.1

4.1.4. Final Implementation

Description

In the final implementation of the AIE-only system, the bottleneck of uneven distribution of workload among the CC engines mentioned in Section 4.1.3 is addressed. The mentioned bottleneck is mitigated in this implementation by adding more CC engines. However, as mentioned in Section 4.1.3, adding more engines to the current chain of CC engines might not help. This is why in the final implementation, the current CC engine chain is multiplied, and this implementation uses multiple CC engine chains. Figure 4.5 shows the block diagram of the multi-chain final implementation of the AIE-only system with 4 CC engine chains.

Some details about the final implementation of the AIE-only system are as follows:

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 outputs 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 an 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.
3. In addition to the division of workload between multiple CC engine chains and reducing the maximum workload of the engines, the delay of sending the coupling calculation results from the first CC engines to the corresponding MLPFE engines decreases because there are fewer destinations to send to.

Bottlenecks and Possible Improvements

Incorporating multiple chains in the design in addition to performing load-balancing on the chains led the final implementation to perform around $2.6\times$ better compared to the second improved implementation with the same number of CC engines per chain. This amount of speedup shows that the uneven distribution of the workload among the CC engines in a chain would cause a huge bottleneck, and adding more chains would better this situation to some extent.

The final implementation can still be improved in many ways, including but not limited to:

- **Engine Optimization** It goes without saying that improving the performance of the MLPFE and CC engines would improve the performance of the design. For instance, The MLPFE engines are perfect candidates for vectorization (this is done in the Heterogeneous system described in

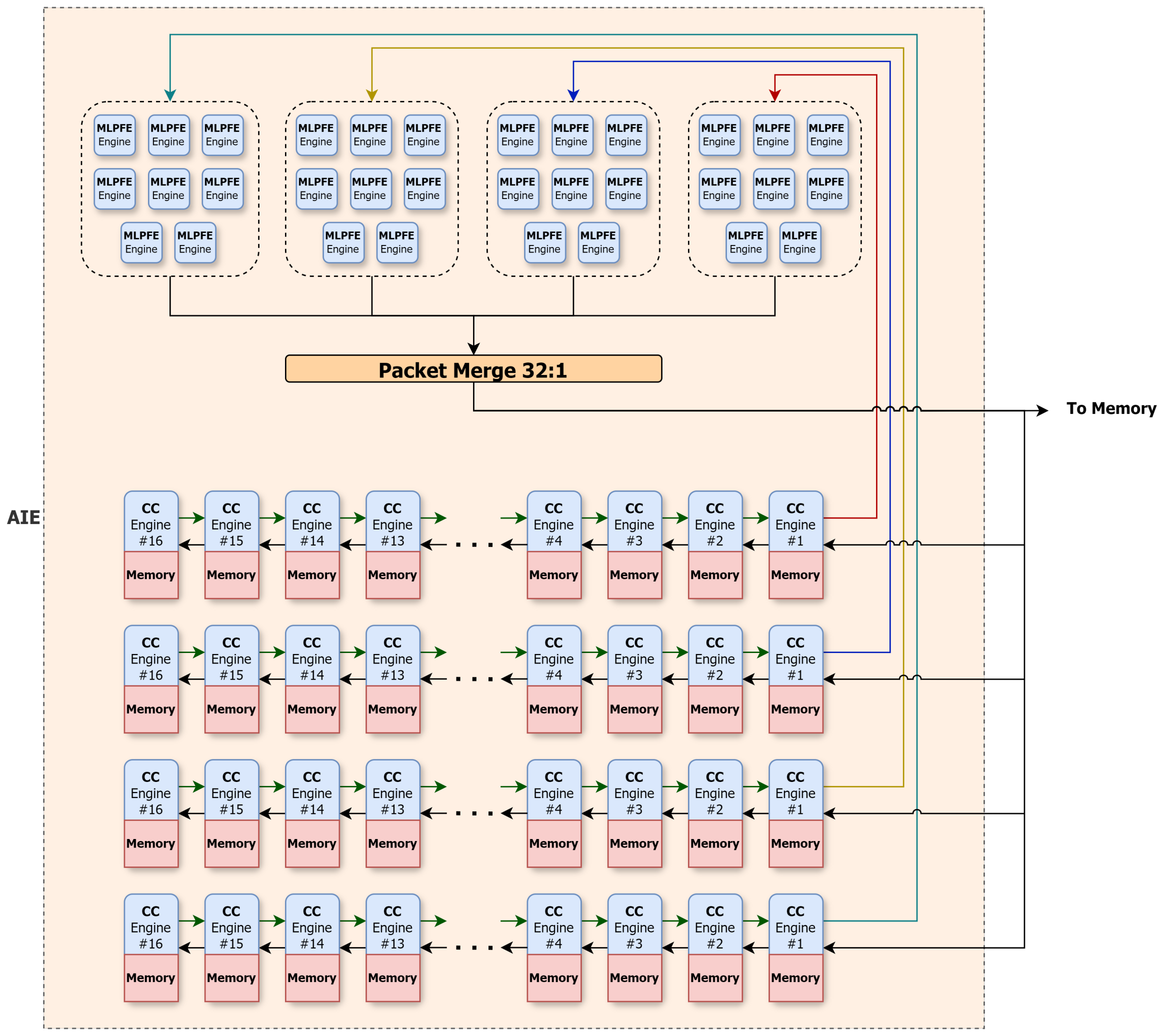


Figure 4.5: Block Diagram of the Final Implementation of the AIE-Only System

Section 4.2), and the number of memory accesses and the flow of operations in the CC engines could potentially be optimized.

- **Using Fixed-Point Arithmetic** Using fixed-point operations can help improve the speed of the calculations in addition to potentially helping with reducing the memory requirement for the engines.

4.1.5. Conclusion - AIE-Only System

During the four stages of improvement of the AIE-only system, we can see more than $3\times$ performance improvement from the basic implementation to the final implementation. The main bottleneck of this system remains to be the coupling calculations. As with any chain in the world, the chain of CC engines is also as fast as its slowest engine. The ideal case is to have all the coupling calculation workload divided perfectly equally among the CC engines. As mentioned in Section 4.1.3, the memory capacity limitation of the AIE tiles prevents us from equally distributing all the workload among the CC engines. Although having multiple CC engine chains helped, but not completely solves this issue. Moving the coupling calculation process to the PL, as will be discussed in Section 4.2, can help improve the performance in different ways:

1. Almost 20 MBytes of memory resources of the PL can be assigned in a custom way to different parts of the design, as opposed to the $12.5MBytes$ of memory in the AIE which is strictly divided among 400 tiles of $32KBytes$ each.

2. In the PL, the CC engines themselves can be implemented as dataflow engines as opposed to the von Neumann architecture of the AIE tiles.

Having mentioned these possible benefits of the Heterogeneous system, it is still unknown whether the Heterogeneous system would perform better in all cases against the AIE-only system or not. This is discussed in Chapter 5, where more in-depth discussions regarding the results from the AIE-only system are also presented.

4.2. Heterogeneous System

In this section, we discuss the implementation details of the Heterogeneous system. As mentioned in Section 3.2.2, 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.

Figure 4.6 shows the general block diagram of the Heterogeneous system.

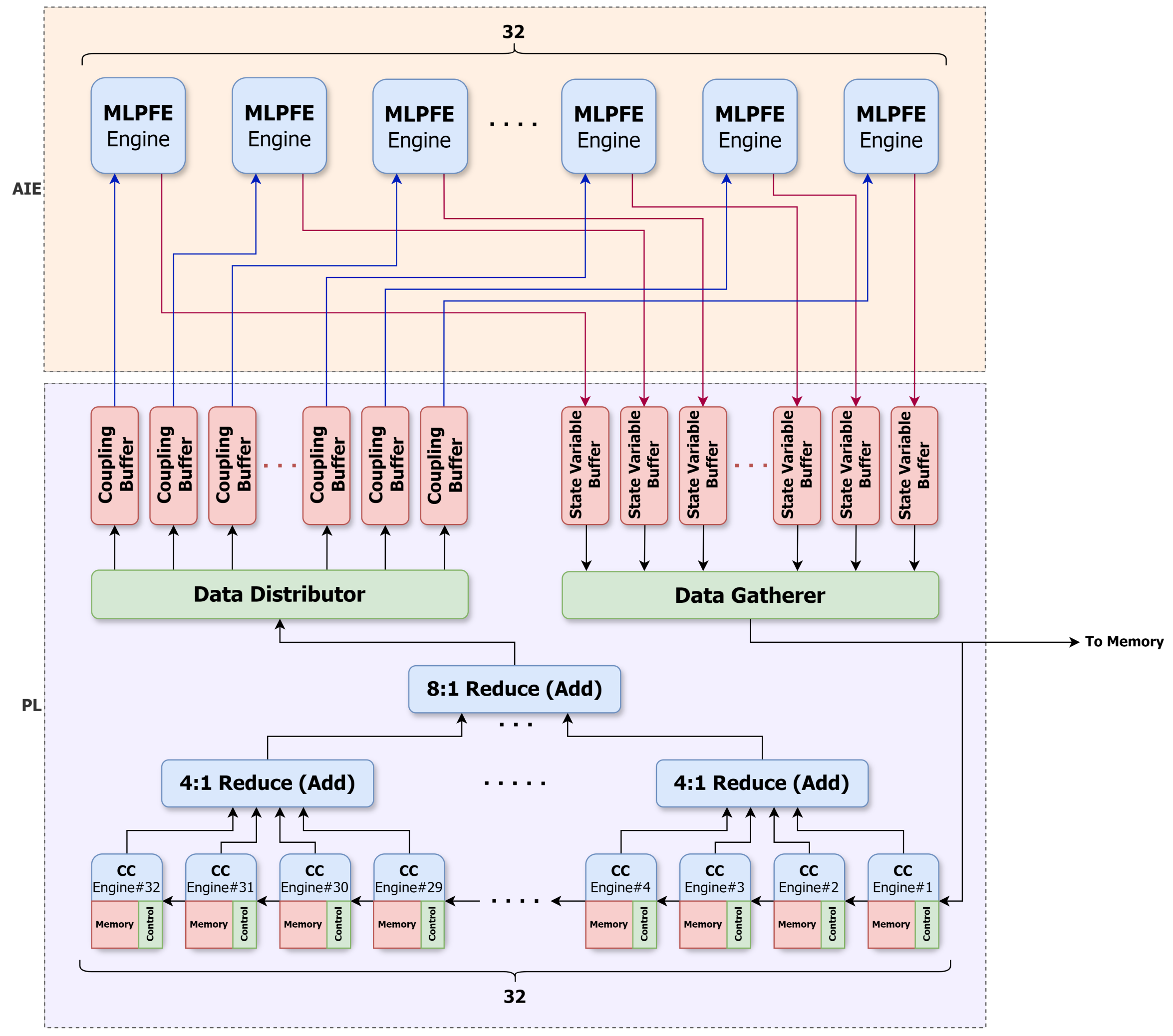


Figure 4.6: Block Diagram of the Implementation of the Heterogeneous System

As can be seen in Figure 4.6, the CC engines in the PL calculate the couplings for each center from a certain delay range, and 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 engine 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 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-vectorized implementation of the engines.

As mentioned earlier, the CC engines reside in the PL. Each engine is assigned a URAM chunk to store the state variables' history data and the connectivity data. For a system with 32 CC engines, each engine would be assigned around 310 KBytes of URAM. Additionally, the BRAM resources in the PL are used for buffering between different parts of the PL, preventing communication stalls.

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 4.7. 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 center 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 from happening on 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 them 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.

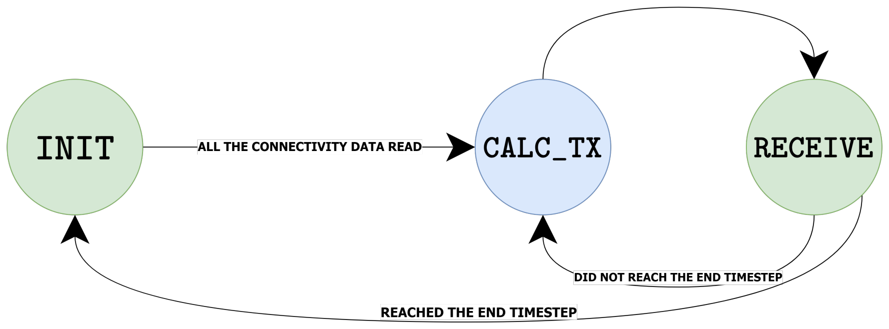

INIT
ALL THE CONNECTIVITY DATA READ
CALC_TX
RECEIVE
DID NOT REACH THE END TIMESTEP
REACHED THE END TIMESTEP

Figure 4.7: CC Engines FSM

All of the states of the CC engine FSM are implemented as a dataflow pipeline. 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 4.8, is more complicated. The operations seen in Figure 4.8 are 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.

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 reading 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).

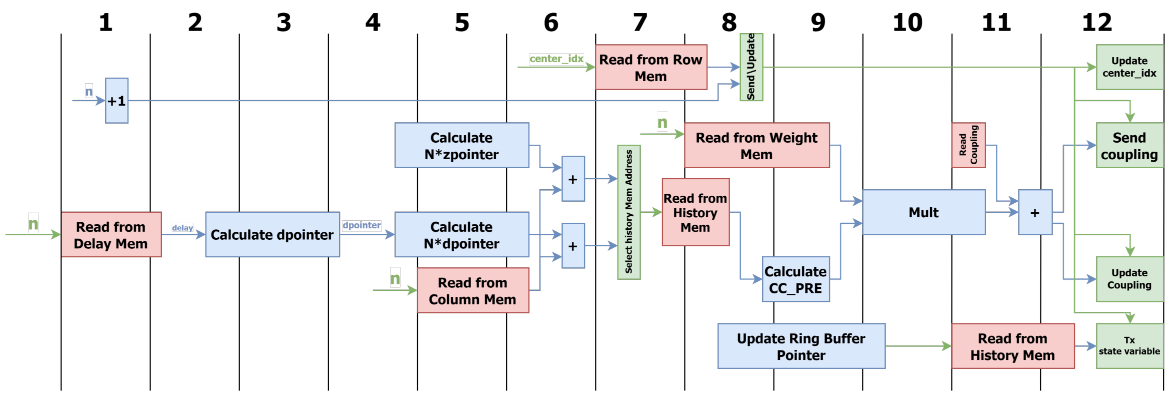


Figure 4.8: CC Engines `CALC_TX` State Pipeline

3. The `dpointer` calculation continues in this stage.
4. With `dpointer` calculated, the calculation of the possible History memory addresses begins. These 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 continue 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 the 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 2.4. 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 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 is fully pipelined, and the described pipeline runs with minimum intervals in between. This means if a center has to calculate 100 calculation 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 4.6 are also implemented as dataflow pipelines. This is also true for Data Distributor and Gatherer blocks.

Comparing the Heterogeneous implementation with the final AIE-only implementation might raise the question of why not use the multi-chain design in the Heterogeneous system. To answer this question, we should look into why we incorporated the multi-chain implementation in the first place. The main reason for using a multi-chain design was to flatten the workload amount in different CC engines of a single chain as much as possible, as memory limitation would not allow us to assign different delay ranges to the CC engines. Another reason was that the more AIE tiles we used in the AIE-only system, the more memory resources we would have. Although the AIE of the Versal ACAP has around 12.25 MBytes of memory, this amount of memory is distributed among 400 tiles. This way, if a tile is not used in an implementation, its memory would not be used either. This would reduce the effective amount of memory available to the system. In the Heterogeneous system, however, all the PL memory resources are available regardless of the number of CC engine chains used in the implementation. In fact, if multiple chains exist in the Heterogeneous system, the same data (the history data) would be copied multiple times in the same amount of memory, which would result in much less effective memory resources available to the system. In the Heterogeneous system, because of the relatively large amount of memory available to each CC engine, the division of the coupling calculation workload in a more balanced way is possible.

When benchmarked, the Heterogeneous system performed around $2.5\times$ faster compared to the final implementation of the AIE-only system. Discussions surrounding the reasons behind this speedup are presented in Section 5.3, where more detailed results of the Heterogeneous system and its performance are discussed in Section 5.2.2.

4.3. Conclusion

In this chapter, we dived into the implementation details of the AIE-only and Heterogeneous systems. At each improvement stage, the implementation was benchmarked with standard, its bottlenecks were identified using the benchmarks and simulation tools, and the identified bottlenecks were addressed in the next version of the implementation. From the basic AIE-only implementation to the Heterogeneous system, we observed around $5\times$ performance improvement. In Chapter 5, the detailed results of the systems discussed in this chapter, in addition to the results from the systems introduced in Section 2.8, are presented. Furthermore, the results and performance of the systems are discussed and compared in more depth.

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 results of the systems are discussed and compared, and the reasons behind the differences in the results are analyzed. Table 5.1 shows all of the systems we look into in this chapter. The Original TVB, Fast TVB, TVB C++, and TVB on GPU are discussed in Section 5.1, and the AIE-Only and Heterogeneous systems are analyzed in Section 5.2. The performance results of the systems listed in Table 5.1 are compared in Section 5.3.

Name	Platform	Language	Notes
Original TVB	CPU	C++	This is a single-core, single-thread port of the original Python-based TVB backend (`tvb_algo`) to C++ by us. *MLP evaluation* and *sparse calculations* are added on top of `tvb_algo` in this implementation.
Fast TVB	CPU	C	This implementation is developed by the TVB team and can only simulate one model.
TVB C++	CPU	C++	This is a multithreaded, SIMD C++ implementation of TVB developed by the TVB team. It can simulate all of the TVB models, but does not have MLP evaluation.
TVB on GPU	GPU	Python (JAX)	This is the GPU version of the TVB algorithm which is developed as a JAX-based package (`vbjax`).
AIE-Only	Versal ACAP	C++ (ADF)	This is the implementation of the TVB on the Versal ACAP that only uses the AI Engines.
Heterogeneous	Versal ACAP	C++ (ADF) Vitis HLS	This is the implementation of the TVB on the Versal ACAP that uses both the AI Engines and the Programmable Logic.

Table 5.1: Summary of all the TVB Systems

5.1. Experimental Setup

5.1.1. Benchmarks

The Neural-Mass Model used for benchmarking the systems developed in this thesis uses the connectivity data from The Virtual Brain (TVB) [44]. The TVB dataset contains three different models with different numbers of centers and connectivity data shown in Table 5.2. Table 5.2 also includes the Calculation Points of each dataset which is the number of non-zero elements in the weight matrix (W) of the datasets. As described in Section 2.2, the NMM divides the brain into a number of centers that are connected to each other in a certain way, and each has a number of state variables and local dynamics. The TVB dataset defines the number of centers and how they are connected to each other. For a large-scale 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 an 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 [30], and the single hidden layer was sufficient to accurately calculate the local dynamics for this model [9]. 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 neuroscientific validity and 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 ms step size, unless otherwise 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 5.2: TVB Dataset

5.1.2. Original TVB

Table 5.3 shows the performance results of the original TVB (sequential) system. As mentioned in Section 2.8.1, the sequential system is a single-core 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 30 MBytes of cache running at a maximum of 5.4 GHz. This processor consumes 125 Watts of power during normal operation, with the maximum power dissipation being around 253 Watts [29].

Benchmark	Original TVB System (`tvb_algo_c`)
TVB76	101.4 ms
TVB192	259.1 ms
TVB600	848.7 ms
TVB998	1,510.9 ms

Table 5.3: Original TVB (`tvb_algo_c`) Execution Time (3000 timesteps)

Due to the large cache capacity of the machine, the sequential system performs relatively well. The coupling calculations are memory-heavy operations and the $30MB$ of cache available to the system benefits this part of the application significantly. As mentioned in Section 2.6, the total available on-chip memory in the Versal ACAP (AIE and PL together) is around 33 MBytes, which puts it very close to the cache capacity of the sequential system. This would make the comparison interesting, as how a traditional system with a much higher clock frequency and a large cache compares to a dataflow machine with a lower clock frequency and around the same amount of on-chip memory. Additionally, the sparse representation used in coupling calculations improved the performance of the `tvb_algo_c` by $2\times$ over the version using dense representation.

5.1.3. Optimized TVB on CPU

The performance numbers for the Fast TVB and TVB C++ were extracted from [36], which because of the limitation of the Fast TVB tool, only a certain model with 379 centers was tested. Table 5.4 shows the execution time of the Fast TVB and TVB C++ running for 100,000 timesteps and a step size of 0.1 ms.

Benchamrk	Fast TVB	TVB C++
Reduced Wong Wang with 379 centers	5,100.0 ms	6,400.0 ms

Table 5.4: Optimized TVB on CPU Execution Time (100,000 timesteps)

As expected, the Fast TVB performs better compared to TVB C++. This is because the Fast TVB system is built around the Reduced Wong Wang model, and it trades off generalization for performance for this specific model.

5.1.4. TVB on GPU

Table 5.5 shows the performance results of the GPU version of TVB (`vbjax`) when running a model for 3000 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 current GPU version of TVB since the `vbjax` does not perform sparse matrix operations at this time.

# Centers (Dense)	Batch Size							
	1	16	32	64	128	256	512	1024
76	105.1 ms	144.1 ms	144.1 ms	144.1 ms	147.2 ms	201.2 ms	549.6 ms	1,084.1 ms
192	114.1 ms	147.2 ms	153.2 ms	159.2 ms	315.3 ms	702.7 ms	1,360.4 ms	2,690.7 ms
600	111.1 ms	186.2 ms	285.3 ms	612.6 ms	1,162.2 ms	2,258.3 ms	4,234.2 ms	8,228.2 ms
998	135.1 ms	279.3 ms	540.5 ms	1,006.0 ms	1,894.9 ms	3,633.6 ms	6,966.0 ms	13,663.7 ms

Table 5.5: TVB on GPU (`vbjax`) Execution Time (3000 timesteps). The numbers in this table are not divided by the batch size.

The batch size in Table 5.5 refers to the number of simulation instances the GPU runs at the same time. The number of simulations GPU can perform at the same time depends on the GPU itself and the size of the model. As the size of the model grows, the GPU saturates faster, and it either performs worse or is unable to fit all the needed data in its memory and crashes. The cells marked with green in Table 5.5 are the best performing (when looking at the time per one simulation) batch sizes for a certain model size. Sometimes, the performance of the system worsens when increasing the batch size, because of the GPU saturation in terms of compute or memory. If the runtime of the simulations increases, there is the chance that the outputs of the concurrent simulation instances do not fit in the memory of the GPU, in which case the chosen batch size must be decreased.

The Quadro RTX6000 used by us for benchmarking the GPU version of TVB operates at a maximum TDP power of 260 Watts. Table 5.6 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 the `nvidia-smi` command-line tool. Based on the reported numbers in Table 5.6, it is fair to assume an average power consumption of around **162 and 236 Watts** for single and batched simulation cases, respectively.

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 5.6: TVB on GPU Power Consumption

5.2. Experimental Results

5.2.1. AIE-Only System

Tables 5.7 and 5.8 show different stages and the performance results of all of the implementations of the AIE-Only system, respectively.

The basic (reduction-tree-based) and first improved (packet-switched-based) implementations were only tested with the TVB76 model and only 16 CC engines during the development phase. The second improved (cascade-stream) and final (multi-chain) implementations are benchmarked with more datasets, however, the TVB998 connectivity data would not fit in their CC engines' memory. As can be

Name	Description
Basic	Uses reduction trees for both the MLP and CC engines.
First Improved	Uses packet switching constructs for both MLPFE and CC engines.
Second Improved	Uses cascade stream for the CC engines.
Final	Incorporates multiple CC-engine chains.

Table 5.7: Summary of all the AIE-Only System Implementations

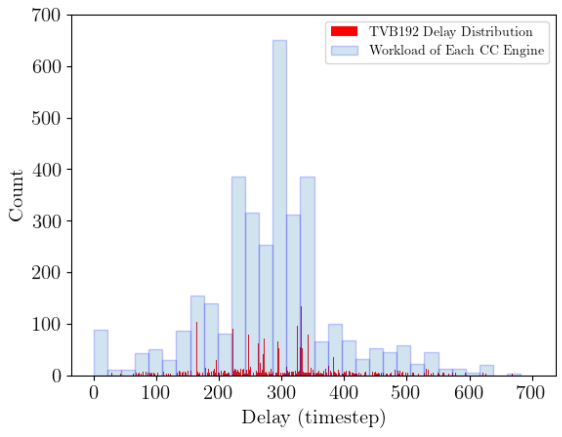
AIE-Only	Basic	First Improved	Second Improved				Final		
# CC Engines	**16**	**16**	**16**	**32**	**64**	**96**	**64 (4×16)**	**128 (4×32)**	**160 (4×40)**
TVB76	66.0 ms	51.0 ms	41.8 ms	30.5 ms	30.1 ms	30.2 ms	15.8 ms	15.4 ms	15.2 ms
TVB192	×	×	165.6 ms	118.3 ms	76.1 ms	74.1 ms	55.0 ms	40.0 ms	39.7 ms
TVB600	-	-	-	-	-	319.8 ms	-	-	205.1 ms
TVB998	-	-	-	-	-	-	-	-	-

Table 5.8: AIE-Only Systems Execution Time (3000 timesteps). The cells marked with ”×” are not benchmarked. The cells marked with ”-” are not runnable on the specified system due to memory limitations.

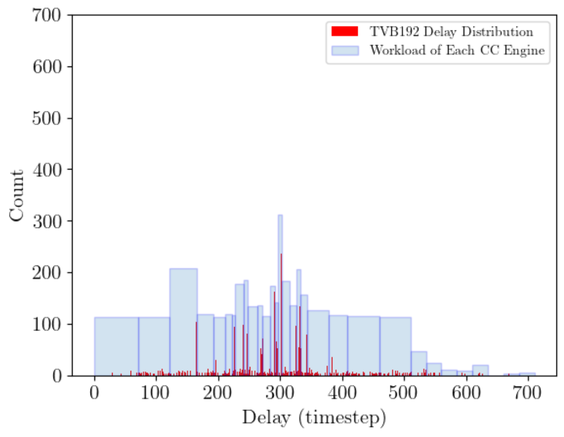
seen in Table 5.8, the stages of the improvement of the AIE-Only system are apparent, with the final implementation with 160 CC engines performing around $4.3\times$ better compared to the basic design. For the rest of this section, we dive into the performance of the CC engines chain, which is present in the second improved and the final implementations of the AIE-Only system.

We mentioned the uneven coupling calculation workload distribution among the CC engines of a chain in Section 4.1.3. We also mentioned that this uneven workload distribution affects the performance of the system negatively, and the time it takes to finish the coupling calculation is dictated by the engine with the largest workload. This bottleneck worsens if the CC engine with the maximum workload assigned is the first one. In this case, not only does the first engine have to wait for all the MLPFE engines to send the newly calculated state variables to it, but it has to do the most calculations among all the CC engines, and also output the coupling results to all the MLPFE engines. The effect of this uneven workload distribution can be seen in Table 5.8, wherein the second improved and final implementations when we increase the number of CC engines, in some cases the runtime would not decrease. This could be for two reasons: either the bottleneck of the system has shifted from the CC engines to the MLPFE engines (which is the case for the final implementation running the TVB76 dataset), or by increasing the number of CC engines the workload of the engine that has to do most calculations is not changed, or changed negligibly (which is the case for the rest).

Figure 5.1a shows the delay distribution of the calculation points in the TVB192 dataset and the workloads assigned to each CC engine if we divide the set of delays equally between all the engines. We can see the unbalanced distribution of the workload among the 32 available engines. On the other hand, Figure 5.1b shows the workloads assigned to each CC engine if we try to balance the workloads between the engines. It can be seen that in the delay regions where the concentration of calculation points is higher, the delay period assigned to each CC engine is very small, and if the concentration of delay points is lower, a larger delay period is assigned to a single engine. Looking at the two strategies of dividing the workloads among the CC engines, we can see the workload of the engine with the maximum amount of calculation points assigned to it in the adaptive distribution is around half the amount in the equal distribution strategy. This results in a better performance of around $2\times$ with the adaptive-distribution strategy.

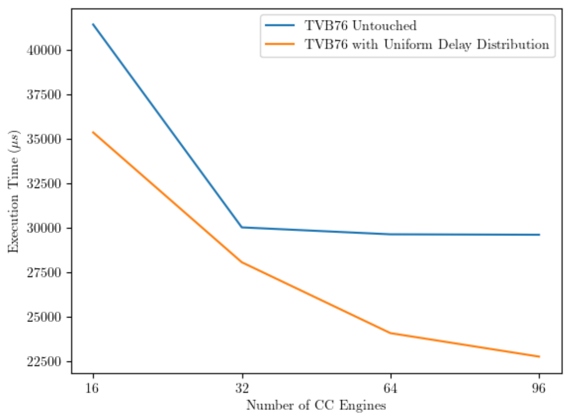


(a) Equal Delay Ranges are Assigned

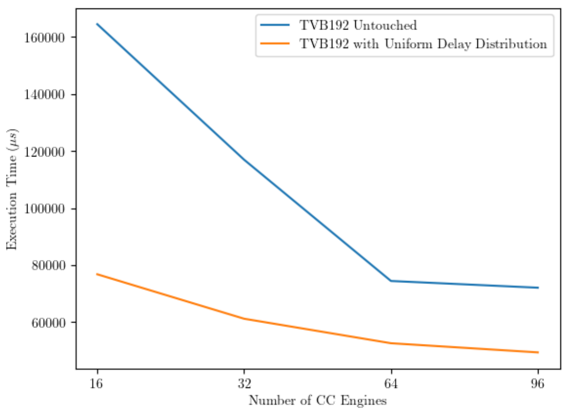


(b) Adapted Delay Ranges are Assigned

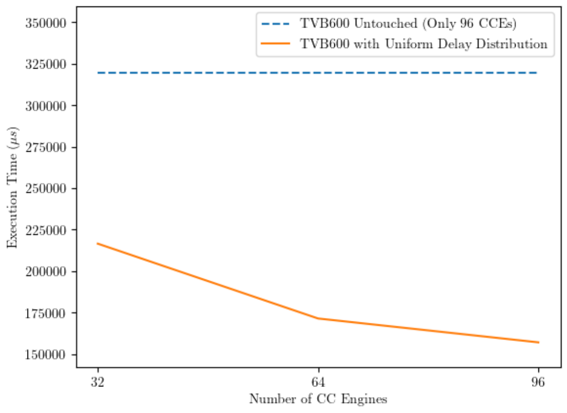
Figure 5.1: TVB192 Delay Values Distribution and CC Engines Assigned Workload



(a) TVB76 Dataset



(b) TVB192 Dataset



(c) TVB600 Dataset

Figure 5.2: Second Improved AIE-Only System's Performance with Uniform Delay Distribution

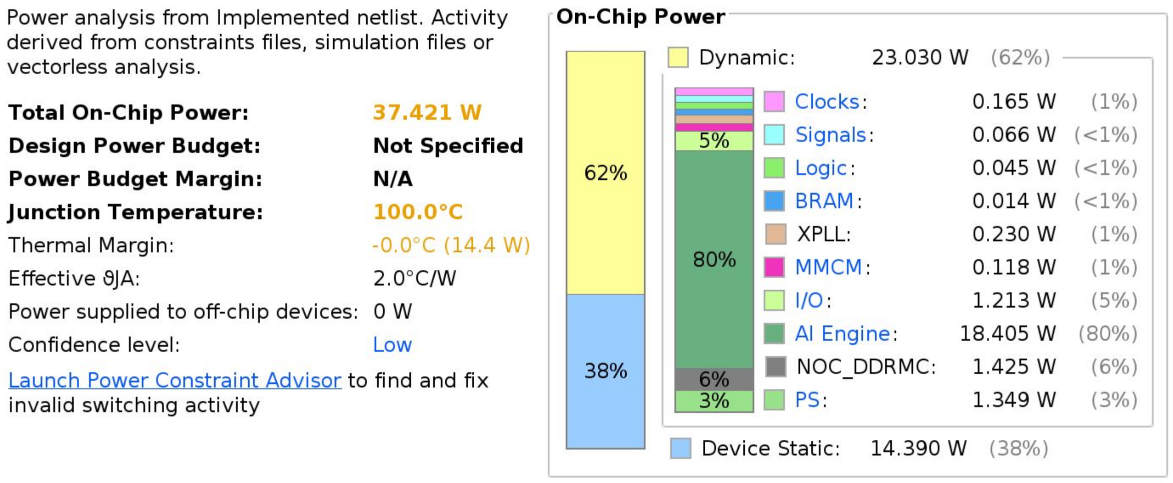
However, as mentioned in Section 4.1.3, the adaptive workload distribution is not possible (at least in an effective way) in the AIE-Only system due to memory capacity limitations. Let's take a look at the example of the TVB192 dataset. In the equal-distribution strategy, each CC engine is assigned a delay range of 22. This means that each engine requires $22 \times 192 \times 4 \approx 17$ KBytes of memory to store the history of the state variables. This is possible as each CC engine has access to 32 KBytes of data memory. On the other hand, in the adaptive workload distribution, the first engine is assigned a delay window of size 71. This means that this engine requires $71 \times 192 \times 4 \approx 54.5$ KBytes of memory for the history data, which is not possible.

In order to see the effect of this uneven workload distribution among the CC engines, we tweaked the TVB datasets in a way that they have the same sparsity but the delay values are distributed uniformly. This means that the workloads assigned to the CC engines are equal. Figure 5.2 shows the performance of the second improved implementation when simulating the tweaked datasets compared to the untouched datasets.

As can be seen in Figure 5.2, when the delay values in a dataset are uniformly distributed, and the workloads of the CC engines are balanced, even with the same sparsity, the performance increases significantly. This shows the effect of uneven distribution of the workload among the CC engines on the performance. As mentioned earlier, the performance does not improve when increasing the number of CC engines, it is either due to the fact that the bottleneck has shifted to the MLPFE engines (since MLPFE and CC engines are running more or less in parallel) or it is because the value of the largest workload assigned to a CC engine has not changed. A good pre-processing algorithm should minimize the maximum workload it assigns to the CC engines while meeting the memory capacity requirements.

In the final implementation of the AIE-Only system, another dimension (the number of CC-engine chains) is added to the design. With this new dimension added, tuning the number of chains and the CC engines per chain would become an interesting challenge. As mentioned earlier, the performance of the coupling calculation chain is limited by the largest amount of workload assigned to any of the CC engines ($MaxWL$). There are a limited number of AIE tiles available in the device, and not all of them can be assigned to CC engines. With this limited amount of AIE tiles available, finding the best pair of CC-engine chains and CC engines per chain is important for potentially maximizing performance.

The power consumption of the Versal ACAP heavily depends on the utilization of the device. AMD provides a power estimation tool as part of Vivado. As shown in Figure 5.3, the power consumption value of the AIE-Only system lies around **32.42 Watts**. Since the entirety of the system is used for any of the simulations regardless of their size, the power consumption value is independent of the input of the system.



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**37.421 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**100.0°C**
Thermal Margin:	-0.0°C (14.4 W)
Effective ϑJA:	2.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨 Dynamic:	23.030 W	(62%)	
🟪 Clocks:	0.165 W	(1%)	
🟦 Signals:	0.066 W	(<1%)	
🟩 Logic:	0.045 W	(<1%)	
🟦 BRAM:	0.014 W	(<1%)	
🟫 XPLL:	0.230 W	(1%)	
🟪 MMCM:	0.118 W	(1%)	
🟨 I/O:	1.213 W	(5%)	
🟩 AI Engine:	18.405 W	(80%)	
⬛ NOC_DDRMC:	1.425 W	(6%)	
🟩 PS:	1.349 W	(3%)	
🟦 Device Static:	14.390 W	(38%)	

Figure 5.3: Power Consumption and Breakdown of The AIE-Only System

5.2.2. Heterogeneous System

Table 5.9 shows the performance results of the Heterogeneous system with 32 CC and 32 MLPFE engines.

Benchmark	Heterogeneous System (32 CC / 32 MLPFE)
TVB76	6.3 ms
TVB192	14.7 ms
TVB600	54.5 ms
TVB998	95.3 ms

Table 5.9: Heterogeneous System Execution Time (3000 timesteps)

The performance of the Heterogeneous system in the case of the TVB76 and TVB192 benchmarks is limited by the MLP evaluation process, and in the case of TVB600 and TVB998 is limited by the coupling calculations. Although there are no chains in this design, the coupling calculation performance is still limited by the CC engine with the largest workload assigned to it. Unlike the AIE-Only system where more equal workload distribution among the CC engines was not possible due to the memory capacity limitations, there are more memory resources available to the CC engines in the Heterogeneous system. By readjusting the pre-processing algorithm to try to distribute the coupling calculation workload equally among the CC engines, in addition to increasing the number of MLPFE engines, we can potentially achieve higher performance. Table 5.10 shows the performance of the Heterogeneous system with 32 CC engines, 64 MLPFE engines, and a more even coupling calculation workload distribution. Additionally, with the number of MLPFE engines increasing, the Data Gather block shown in Figure 4.6 is divided into 2 levels of gathering blocks, increasing the parallel execution of their operation.

Benchmark	Heterogeneous System (32 CC / 64 MLPFE)
TVB76	4.1 ms
TVB192	7.5 ms
TVB600	21.1 ms
TVB998	33.8 ms

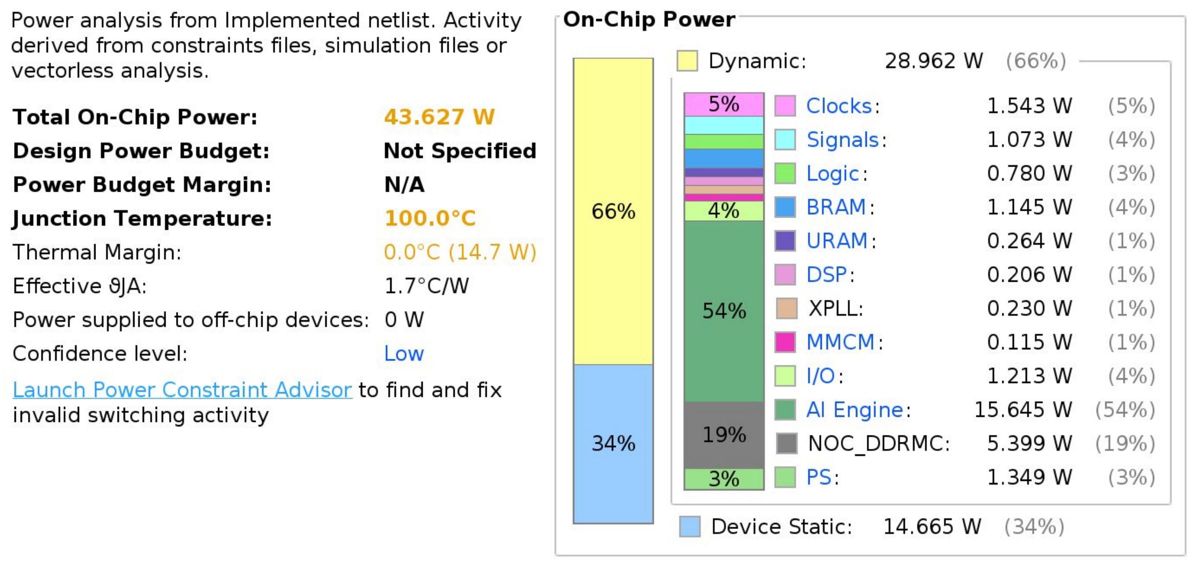
Table 5.10: Final Heterogeneous System Execution Time (3000 timesteps)

As can be seen in Table 5.10, the performance of the systems improves by almost $3\times$ in the largest model. This performance boost shows the effect of the coupling calculation workload distribution, the increase of the MLPFE engines, and also the optimization in the data-gathering process. Although the number of CC engines is the same, with smarter distribution of tasks among them we can reduce the runtime of the coupling calculation process. Furthermore, with the runtime of MLPFE and CC engines decreasing, the data movement processes (such as data reduction or gathering) become a considerable portion of the runtime. Optimizing the Data Gather block, by making it multi-level and extra pipelining, helped the Heterogeneous system to achieve better performance.

Using Vivado, the power consumption of the Heterogeneous system is extracted as shown in Figure 5.4. Less power is consumed by the AIE part of the system compared to the AIE-Only system, and more power is dissipated by the PL as the coupling calculation process is moved from the AIE to the PL. As shown in Figure 5.4, the power consumption value of the Heterogeneous system lies around **43.63 Watts**. Since the entirety of the system is used for any of the simulations regardless of their size, the power consumption value is independent of the input of the system.

5.3. Performance Comparison

In this section, we discuss and compare the performance results from the different systems mentioned earlier. First, the results of the AIE-Only and Heterogeneous systems are compared to each other and to the CPU versions of the TVB. Next, an extensive comparison between the final Versal ACAP system (the Heterogeneous system) and the GPU version of the TVB is presented.


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**43.627 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**100.0°C**
Thermal Margin:	0.0°C (14.7 W)
Effective ϑJA:	1.7°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	28.962 W	(66%)
🟪	Clocks:	1.543 W	(5%)
🟦	Signals:	1.073 W	(4%)
🟩	Logic:	0.780 W	(3%)
🟦	BRAM:	1.145 W	(4%)
🟪	URAM:	0.264 W	(1%)
🟪	DSP:	0.206 W	(1%)
🟫	XPLL:	0.230 W	(1%)
🟥	MMCM:	0.115 W	(1%)
🟩	I/O:	1.213 W	(4%)
🟩	AI Engine:	15.645 W	(54%)
⬛	NOC_DDRMC:	5.399 W	(19%)
🟩	PS:	1.349 W	(3%)
🟦	Device Static:	14.665 W	(34%)

Figure 5.4: Power Consumption and Breakdown of The Heterogeneous System

5.3.1. AIE-Only, Heterogeneous, and TVB on CPU

As can be seen from the numbers presented earlier in this chapter in Tables 5.3, 5.4, 5.8, and 5.10, both the Final 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 ACAP, the custom, application-specific dataflow design of the AIE-Only and Heterogeneous systems made them perform better in terms of speed. Additionally, by looking at the power consumption of the AIE-Only and Heterogeneous systems as shown in Figures 5.3 and 5.4, comparing it to the base power consumption of the benchmarked CPU (125 Watts), we can see the Versal ACAP systems perform around $3\times$ better in terms of power efficiency. This makes the AIE-Only and Heterogeneous systems much more energy efficient compared to the CPU versions of TVB.

The Heterogeneous system performs around $10\times$ better compared to the Final AIE-Only implementation when running the TVB600 model. This significantly better performance of the Heterogeneous system is due to the pipelining in coupling calculation, no stalling in inter-engine communication, and parallelism in data gathering and distribution.

- **Pipelining in Coupling Calculations** As shown in Figure 4.8, the CC engine has a 12-stage pipeline in the Heterogeneous system. This pipelining is possible because of the custom memory hierarchy of the engine, in addition to the dataflow style implementation of the CC engines. Although the AIE array is designed for adaptive dataflow graphs, the AIE tiles themselves are still using von Neumann architecture. This means that the processor has to access the memory for each operation, and this limits the performance of the calculations, even though they are running on a clock frequency 5 times the one in the PL.
- **No Stalling in Inter-Engine Communication** In the Heterogeneous system, due to the larger amount of memory resources, there are buffers in between all the engine-to-engine communications. In a communication, if the receiver is not ready to receive data, with sufficiently large buffers in between the communication, the transmitter does not have to wait for the receiver. However, in the AIE-Only system, the AXI4-Stream communications between the AIE tiles have very small buffers in between. Additional buffers can be added for the communications, but it uses the memory resources of the AIE tiles. These memory resources are limited in size and location, meaning it is not possible to put sufficiently large buffers wherever it is needed. This causes stalls for the MLPFE and CC engines when they try to send data to other engines in the AIE-Only system. Due to the presence of larger and more flexible (in terms of location and accessibility) memory in the PL, the MLPFE and CC engines in the Heterogeneous system experience almost no stalls when they try to transmit data.

- **Parallelism in Data Gathering and Distribution** As mentioned in Section 4.1, the process of concatenation and summation of the outputs of the MLPFE and CC engines, respectively, were an important bottleneck. Due to the lack of sufficient buffering and the sequential nature of the AIE tiles, the tree-style reduction implementation was not successful in the AIE-Only system as discussed in Section 4.1.1. However, because of enough buffering in the communications, pipelining, and also the parallel nature of the hardware, the Heterogeneous system can perform the data gathering and distribution much more efficiently.

With the better performance of the Heterogeneous system over the Final AIE-Only system, and the fact that this performance boost was achieved by moving the coupling calculation process to the PL, a question that arises is how good the AIE of the Versal ACAP is. In other words, would we achieve even better performance if we also moved the MLP evaluation process to the PL? In order to answer this question, let's take a look at the AIE performance in the Heterogeneous system.

In the Heterogeneous system with 64 MLPFE engines, each engine performs the MAC calculations in a Single Instruction, Multiple Data (SIMD) fashion with 16 lines. The AIEs are capable of performing one SIMD operation per clock cycle, which means that the AIE in the Heterogeneous system achieves a computation rate of around $1024\ \frac{MAC}{Clock}$ or $2 \times 1.25 \times 1024 = 1,280\ \frac{GFLOPS}{s}$, considering the clock frequency of 1.25 GHz of the AIE. It is worth noting that only around 16% of the AIE tiles are being used in the Heterogeneous system for the MLPFE engines, and this number can increase with more engines. The MLP evaluation process mostly includes MAC operations, which makes it reasonable to assume the same rate of computation for it.

The PL runs at a clock frequency of around 250 MHz, which is 20% of the frequency of the AIE. This means that in order for the PL to match the performance of the AIE in the MLP calculation, it has to operate at around $5120\ \frac{MAC}{Clock}$. In order for the PL to reach this level of performance, it requires 5120 single-precision floating-point MAC units, which is more than the resources available in the PL. The Versal ACAP's PL has 1968 DSP slices [64], around 10% of them being used by the CC engines. The resource limitation also applies to the memory, where most of the BRAM and URAM resources of the PL are used by the coupling calculation process. Furthermore, if we put aside the matter of resources, by looking at the performance of the DNN inference systems implemented in pure FPGA [23, 52], we can see that the peak performance of these systems is in the range of 10s to 100s of $\frac{GFLOPS}{s}$, less than the computational performance of the AIE.

By comparing the performance results of the AIE-Only and Heterogeneous systems, and also considering the discussion presented earlier, it is fair to say that the AIE of the Versal ACAP is most suitable for dense, structured, and data-intensive computation workloads (such as MLP, or in general, DNN evaluation). On the other hand, computation workloads such as highly sparse calculation or scatter-gather operations would not be a good fit for the AIE and are better implemented in the PL. It is reasonable to see the AIE as a small-scale GPU with a high-bandwidth connection to the PL, capable of executing highly dense and data-intensive workloads very well at a much higher clock frequency, while the rest of the application's workload (in our case the coupling calculations, or in other cases control and networking) can lie in the PL.

5.3.2. Heterogeneous System and TVB on GPU

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

Throughput

When we talk about the throughput of the system, we refer to how many simulation timesteps (or iterations) the system can perform in a second ($\frac{iters}{s}$). Figures 5.5 and 5.6 show the throughput results of the Heterogeneous system and the GPU version of TVB, respectively. The dashed lines in Figure 5.6 are the throughput results of the Heterogeneous system and are used for comparison.

Firstly, as can be seen from both Figures 5.5 and 5.6, the throughputs 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

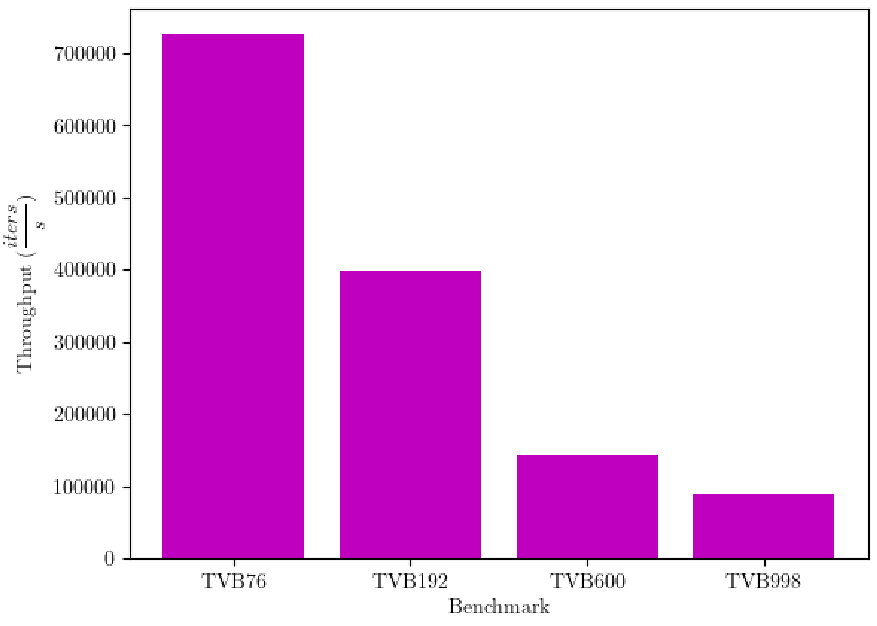


Figure 5.5: Throughput of the Heterogeneous System

for all of the benchmarks remains more or less the same. This shows that the GPU can be saturated with larger batch sizes or larger models due to memory limitations, as mentioned earlier in Section 5.1.4.

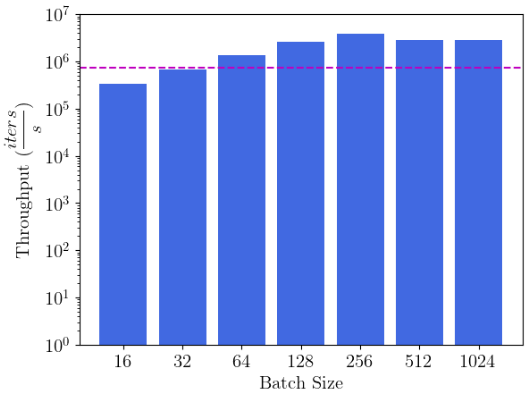
Additionally, when looking at Figure 5.6, we can see that the GPU version of TVB, in its best case, performs around $2.5 - 7\times$ faster than the Versal ACAP Heterogeneous system in terms of throughput. This performance gap is mainly due to the batching of the simulations performed by the GPU version of TVB. Furthermore, the batching of simulation instances helps with hiding the overhead that comes with the host-GPU communication. This high throughput of TVB on GPU makes it a great fit for the model-fitting workloads required to build personalized large-scale brain network models.

Currently, the Heterogeneous system does not support running multiple simulations at the same time. One of the main factors that limit the system from doing simulation batching is the on-chip memory capacity. If we want better throughput for the Heterogeneous system by incorporating simulation batching, the following measures can be taken:

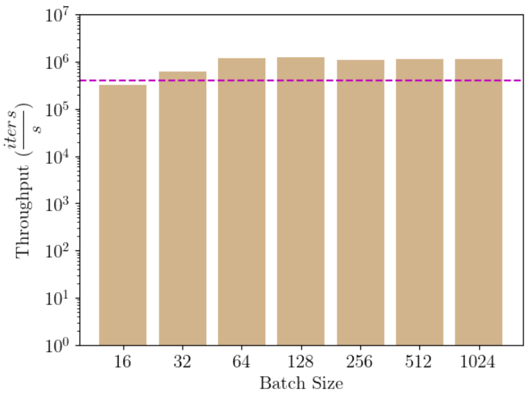
1. The on-chip memory assigned to each of the simulation instances is reduced, allowing for more concurrent, independent, coupling calculation processes. The model fitting process performs many simulations with the same connectivity structure, meaning that the sparsity pattern and delay values of the connectivity matrices are the same, and only the values of the weight matrix are 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 limitations on the largest model that the system can handle, off-chip memory can be used to compensate for this memory shortage. Incorporating the off-chip memory can cause a performance flop, but as we scale up by running multiple simulations at the same time, throughput ought to increase. 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

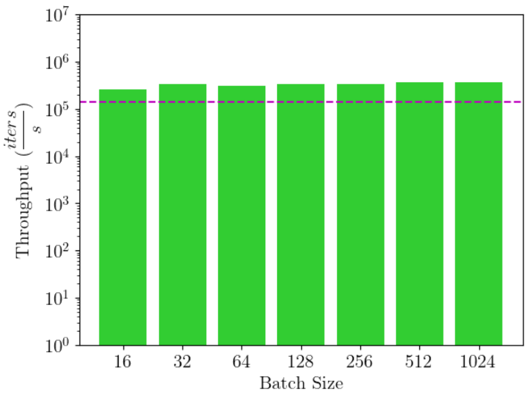
The first step in building a patient-specific, large-scale brain model is to perform the model fitting. In this step, as mentioned earlier, we need to perform a large number of simulations to train the parameters of the model. However, when the system is done with its training, it can be used in many



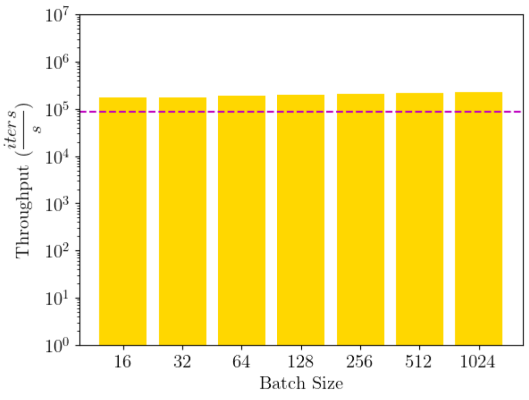
(a) *TVB76* Benchmark



(b) *TVB192* Benchmark



(c) *TVB600* Benchmark



(d) *TVB998* Benchmark

Figure 5.6: Throughput of the TVB on GPU

settings including real-time applications (such as the ones in [45, 59]). In this case, what matters is how fast the system performs a single simulation that is its latency. Figure 5.7 shows the latency of the Heterogeneous system and the GPU version of TVB when running different models for 3000 timesteps.

As can be seen in Figure 5.7, the Heterogeneous system on the Versal ACAP performs on average around $13\times$ 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 ACAP system makes it more suitable for potential real-time and streaming applications that might utilize large-scale neural-mass modeling. One example of such applications could be future multi-site brain-interface systems [48], which target multiple networks of the brain for stimulation. These systems can potentially benefit from low-latency large-scale brain network simulation to design their multi-site stimulation strategy. Another example is model-based approaches in clinical decision-making, such as virtual epilepsy surgery [58, 33] where possible scenarios regarding the surgical procedure can be tested and verified by large-scale brain simulation systems.

Power, Energy, and Area

Although the Versal ACAP consumes less power, to compare the power efficiency of the Versal ACAP 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. For the Heterogeneous system, the power consumption results shown in Figure 5.4 are used. For the GPU version of TVB, power consumption results reported in Table 5.6 are used. Figures 5.8 and 5.9 show the energy and performance per Watt results respectively.

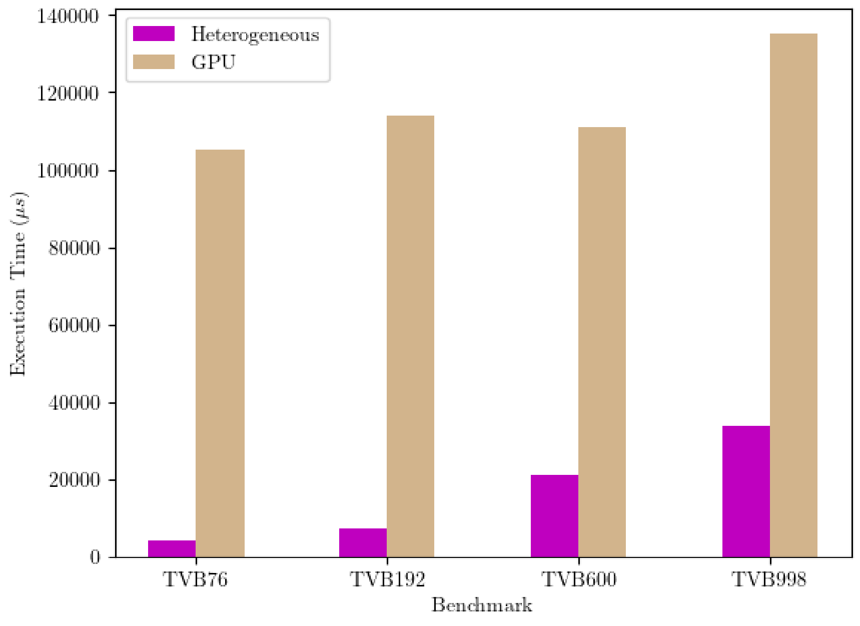


Figure 5.7: Latency of the Heterogeneous and TVB on GPU Systems. The latency results are extracted by running a single simulation for 3000 timesteps.

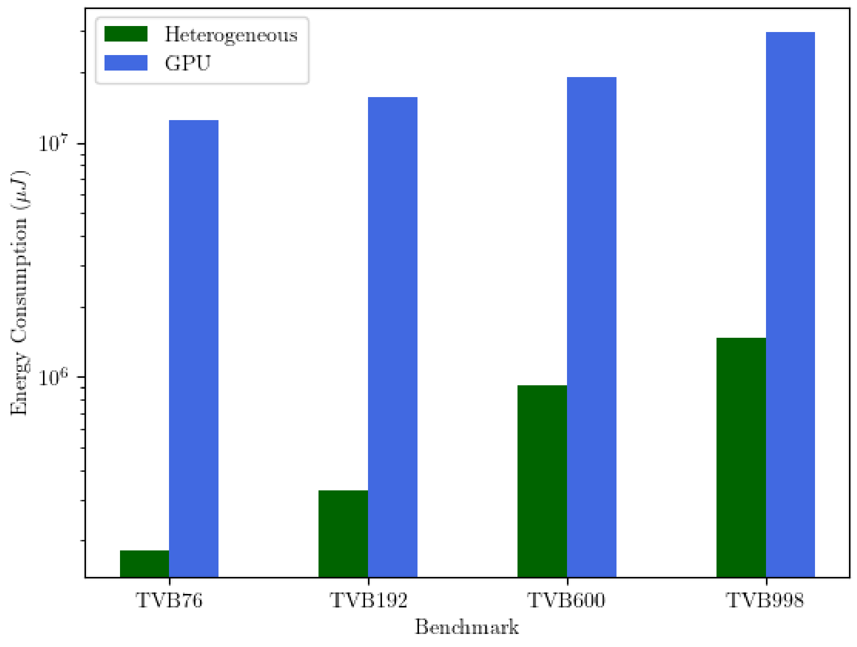


Figure 5.8: Energy Consumption for Heterogeneous and GPU Systems. The energy consumption results are for a single simulation running for 3000 timesteps.

As expected, the Heterogeneous system consumes less energy compared to the GPU version of TVB, as shown in Figure 5.8. When running a single simulation, the energy consumption difference is around two orders of magnitude, making the Versal ACAP system much more energy efficient compared to the GPU system. In terms of performance per Watt, as can be seen in Figure 5.9, 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 system makes it more power efficient.

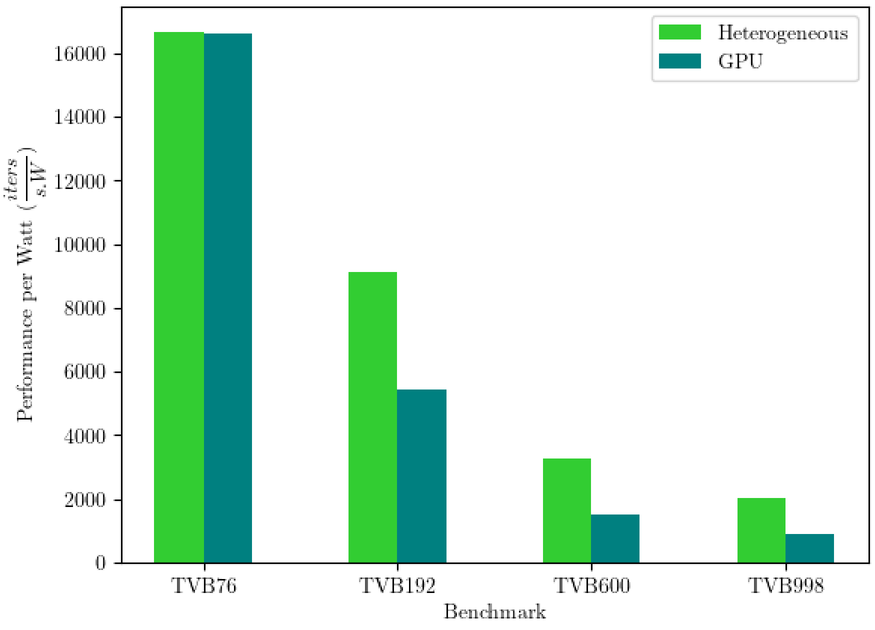


Figure 5.9: Performance per Watt for Heterogeneous and GPU Systems. The performance per Watt results are the throughput of the systems for a unit of power they consume.

In terms of area and process technology, the Quadro RTX6000 GPU is built on 12nm TSMC technology and fits 18.6 billion transistors in a chip size of 754 mm^2 [55]. The Versal ACAP AI Core series is built on TSMC 7nm FinFET technology with a packaging size of 2,025 mm^2 [64]. No public information regarding the die size or transistor count of the Versal ACAP exists, which makes the comparison between Versal and GPU difficult. However, considering Versal ACAP's smaller process technology and also relatively big packaging size, we can estimate that it packs more transistors compared to the RTX6000 GPU. This means that in terms of *Performance per Area*, the GPU version of TVB performs better compared to the Heterogeneous system.

5.4. Conclusion

In this chapter, the performance results from different versions of TVB on CPU, GPU, and Versal ACAP were presented and compared. Two different versions of Versal ACAP implementations (AIE-Only and Heterogeneous) were benchmarked in Sections 5.2.1 and 5.2.2 respectively, and compared with each other and to the CPU versions of TVB in Section 5.3.1. The final Versal ACAP system (the Heterogeneous system) was also compared to the high-performance GPU version of TVB in Section 5.1.4. We showed that although the Heterogeneous system is inferior in terms of throughput compared to the GPU version of TVB, it performs better in terms of latency, energy consumption, and performance per Watt; which makes it a great candidate for real-time and low-power applications that require large-scale brain network simulations.

6 Conclusion

In this thesis, we dived into the details of the large-scale brain network modeling problem, in addition to the design and implementation of an adaptive system to accelerate it. In Chapter 2, we discussed different abstraction levels regarding brain modeling and looked more in-depth into the large-scale neural-mass modeling of the brain and how we formulate this problem. We presented the brain network simulation's applications in different clinical settings, in addition to how these models are built. Furthermore, the details of the Versal ACAP device and its components were presented in this chapter. On top of that, algorithms and techniques that are used in this thesis, such as MLPs and sparse matrix operations, were discussed. At the end of the chapter, different CPU and GPU versions of The Virtual Brain (TVB) tool were briefly introduced.

In Chapter 3, the problem at hand was analyzed in terms of memory and computational requirements. Additionally, three different design candidates were presented and discussed in detail, in addition to comparing their device-specific requirements to the resources available in Versal ACAP. In the end, the design candidates were compared to each other and one was chosen for implementation in Versal ACAP.

In Chapter 4, we dived into the implementation details of the AIE-Only and the Heterogeneous systems. At each improvement stage, the implementation was benchmarked with a standard dataset, its bottlenecks were identified using the benchmarks and simulation tools, and the identified bottlenecks were addressed in the next version of the implementation. From the basic AIE-Only implementation to the Heterogeneous system, we observed around $5\times$ performance improvement.

In Chapter 5, the performance results from different implementations of the TVB algorithm on CPU, GPU, and Versal ACAP were presented and compared. Two different versions of Versal ACAP implementations (AIE-Only and Heterogeneous) were benchmarked and compared with each other and to the CPU versions of TVB. The final Versal ACAP system (the Heterogeneous system) was also compared to the high-performance GPU version of TVB. We showed that although the Heterogeneous system is inferior in terms of throughput compared to the GPU version of TVB, it performs better in terms of latency, energy consumption, and performance per Watt; which makes it a great candidate for real-time and low-power applications that require large-scale brain network simulations.

6.1. Main Contributions

The main contributions of this work are as follows:

- **Analysis of Large-Scale Neural-Mass Model Simulation Problem** The problem of large-scale NMM acceleration on the Versal ACAP was formulated based on the studied literature and analyzed extensively in terms of memory and computation requirements.

- **Design of Application-specific Dataflow Acceleration** From the three proposed designs to accelerate the problem, one was chosen based on the performed analysis for implementation in the Versal ACAP. The designs had to consider the main challenges of the problem which were high-performance requirements and generality of the system. At first glance, the problem seems to be a control-intensive and difficult-to-accelerate algorithm. However, in the chosen system, we turned the NMM acceleration problem into a dataflow-style computing system that can effectively map into the target device.
- **Implementation of AIE-Only System Usable on AMD XDNA™ Architecture** The chosen design was implemented purely in the AIE component of the Versal ACAP. The AIE-Only system performed on average $2 - 5\times$ better compared to the CPU versions of TVB. Additionally, the AIE-Only implementation is capable of being used on any AMD device with XDNA™ architecture, such as some Ryzen CPUs. This can potentially bring the performance boost of the AIE-Only system on the Versal ACAP to other AMD devices as well.
- **Implementation of a Low-Latency, Low-Energy, and Efficient Heterogeneous System** The chosen design was also implemented as a heterogeneous system in the Versal ACAP, incorporating both AIE and PL of the device. The Heterogeneous implementation performed considerably better in terms of latency, energy, and power efficiency compared to the GPU version of TVB.

6.2. Future Work

The possible future directions for this work and some suggestions are presented in this section.

- **In terms of future work:**

  1. As the benefit of using a dataflow-style architecture in this application is proven in the Versal ACAP system, one could port the same application onto hardware designed specifically for dataflow computing.
  2. The current implementation of the system focuses on accelerating the execution of a single simulation, or lower latency. In future work, one could tune the system to make it fit better for high-throughput simulation acceleration.
  3. One could move away from single-precision floating point numbers used in the current system and use either fixed-point or lower-precision floating point number representation.
  4. A Graphical User Interface (GUI) can be developed to make the interaction and using the system more straightforward.
  5. A smarter pre-processing algorithm that can divide the coupling calculation workload better, or optimally map the simulation instance to the best implementation could be beneficial in achieving better performance for different inputs to the system.

- **In terms of suggestions:**

  1. The AI Engines of the Versal ACAP provide great speedup while maintaining their straightforward development cycle. The memory structure and capacity of the AI Engines were limiting for our application, especially when implementing the CC engines on them. More flexible and larger memory banks could be beneficial in future versions of the device.
  2. The current compiler of the AI Engines does not allow for an AIE kernel to have more than 32 KBytes of program memory. Although each tile has access to 128 KBytes of memory, this limitation of the tool causes unnecessary challenges when working with the AIE.

References

[1] Shunichi Amari. “Dynamics of pattern formation in lateral-inhibition type neural fields”. In: *Biological Cybernetics* 27 (1977), pp. 77–87. DOI: `https://doi.org/10.1007/BF00337259`.

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

[3] AMD. *Versal Adaptive SoC AI Engine Architecture Manual (AM009)*. 2023. URL: `https://docs.xilinx.com/r/en-US/am009-versal-ai-engine` (visited on 01/31/2024).

[4] AMD. *Versal Adaptive SoC Design Guide (UG1273)*. 2024. URL: `https://docs.amd.com/r/en-US/ug1273-versal-acap-design` (visited on 06/14/2024).

[5] AMD. *Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller 1.0 LogiCORE IP Product Guide (PG313)*. 2024. URL: `https://docs.amd.com/r/en-US/pg313-network-on-chip/IP-Facts` (visited on 05/30/2023).

[6] AMD. *Versal Adaptive SoC Technical Reference Manual (AM011)*. 2024. URL: `https://docs.amd.com/r/en-US/am011-versal-acap-trm` (visited on 10/05/2023).

[7] AMD. *Versal AI Core Series*. 2024. URL: `https://www.xilinx.com/products/silicon-devices/acap/versal-ai-core.html` (visited on 06/12/2024).

[8] Sora An et al. “Optimization of surgical intervention outside the epileptogenic zone in the Virtual Epileptic Patient (VEP)”. In: *PLOS Computational Biology* 15.6 (June 2019), pp. 1–25. DOI: `https://doi.org/10.1371/journal.pcbi.1007051`.

[9] Nina Baldy et al. “Efficient Inference on a Network of Spiking Neurons using Deep Learning”. In: *bioRxiv* (2024). DOI: `https://doi.org/10.1101/2024.01.26.577077`.

[10] 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. DOI: `https://doi.org/10.1098/rstb.1956.0012`.

[11] Edward K. Blum et al. “The Mathematical Theory of Optimal Processes.” In: *American Mathematical Monthly* 70 (1963), p. 1114. DOI: `https://doi.org/10.2307/2312867`.

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

[13] Aydin Buluc et al. “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks”. In: *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 233–244. DOI: `10.1145/1583991.1584053`.

[14] Tian Qi Chen et al. “Neural Ordinary Differential Equations”. In: *Neural Information Processing Systems*. 2018. DOI: `https://doi.org/10.48550/arXiv.1806.07366`.

[15] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. USA: Society for Industrial and Applied Mathematics, 2006. ISBN: 0898716136.

[16] Gustavo Deco and Morten L. Kringelbach. “Great Expectations: Using Whole-Brain Computational Connectomics for Understanding Neuropsychiatric Disorders”. In: *Neuron* 84 (2014), pp. 892–905. DOI: `https://doi.org/10.1016/j.neuron.2014.08.034`.

[17] Gustavo Deco et al. “Key role of coupling, delay, and noise in resting brain fluctuations”. In: *Proceedings of the National Academy of Sciences* 106 (2009), pp. 10302–10307. DOI: `https://doi.org/10.1073/pnas.0901831106`.

[18] Gustavo Deco et al. “The Dynamic Brain: From Spiking Neurons to Neural Masses and Cortical Fields”. In: *PLoS Computational Biology* 4 (2008). DOI: `https://doi.org/10.1371/journal.pcbi.1000092`.

[19] National Academy of Engineering. *Grand Challenges*. 2008. URL: `https://www.engineeringchallenges.org/challenges.aspx` (visited on 06/21/2024).

[20] Richard FitzHugh. “Impulses and Physiological States in Theoretical Models of Nerve Membrane.” In: *Biophysical journal* 1 6 (1961), pp. 445–66. DOI: `https://doi.org/10.1016/S0006-3495(61)86902-6`.

[21] J. Fousek. “Efficient sparse matrix-delayed vector multiplication for discretized neural field model”. In: *J Supercomput* 74 (2018), pp. 1863–1884.

[22] Smaragdos G et al. “BrainFrame: A node-level heterogeneous accelerator platform for neuron simulations”. In: *IOP Journal of Neural Engineering* (2017). URL: `http://iopscience.iop.org/article/10.1088/1741-2552/aa7fc5/meta`.

[23] Kaiyuan Guo et al. “A Survey of FPGA-based Neural Network Inference Accelerators”. In: 12.1 (Mar. 2019). DOI: `10.1145/3289185`.

[24] Hermann Haken. *Brain Dynamics: Synchronization and Activity Patterns in Pulse-Coupled Neural Nets with Delays and Noise*. Springer Science & Business Media, 2006. ISBN: 978-3-540-46284-2.

[25] HBP. *Human Brain Project (HBP)*. 2024. URL: `https://www.humanbrainproject.eu/en/` (visited on 07/05/2024).

[26] 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`.

[27] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert L. White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2 (1989), pp. 359–366. DOI: `https://doi.org/10.1016/0893-6080%2889%2990020-8`.

[28] Kevin Hunter, Lawrence Spracklen, and Subutai Ahmad. “Two sparsities are better than one: unlocking the performance benefits of sparse–sparse networks”. In: *Neuromorphic Computing and Engineering* 2.3 (2022), p. 034004. DOI: `10.1088/2634-4386/ac7c8a`.

[29] Intel. *Intel® Core™ i7-13700KF Processor*. 2024. URL: `https://www.intel.com/content/www/us/en/products/sku/230489/intel-core-i713700kf-processor-30m-cache-up-to-5-40-ghz/specifications.html` (visited on 07/02/2024).

[30] 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`. (Visited on 11/23/2023).

[31] V. K. Jirsa et al. “The Virtual Epileptic Patient: Individualized whole-brain models of epilepsy spread”. In: *NeuroImage*. Individual Subject Prediction 145 (Jan. 15, 2017), pp. 377–388. DOI: `10.1016/j.neuroimage.2016.04.049`. (Visited on 06/14/2024).

[32] Viktor Jirsa and Hermann Haken. “Field Theory of Electromagnetic Brain Activity.” In: *Physical review letters* 77 5 (1996), pp. 960–963. DOI: `https://doi.org/10.1103/physrevlett.77.960`.

[33] Viktor Jirsa et al. “Personalised virtual brain models in epilepsy”. In: *The Lancet Neurology* 22.5 (2023), pp. 443–454. DOI: `10.1016/S1474-4422(23)00008-X`.

[34] A.A. Khokhar et al. “Heterogeneous computing: challenges and opportunities”. In: *Computer* 26.6 (1993), pp. 18–27. DOI: `10.1109/2.214439`.

[35] Andrea I. Luppi et al. “Paths to Oblivion: Common Neural Mechanisms of Anaesthesia and Disorders of Consciousness”. In: *bioRxiv* (2021). DOI: `https://doi.org/10.1101/2021.02.14.431140`.

[36] Ignacio Martin et al. “TVB C++: A Fast and Flexible Back-End for The Virtual Brain”. In: 2024. DOI: `https://doi.org/10.48550/arXiv.2405.18788`.

[37] Michael Mascagni and Arthur Sherman. “Numerical Methods for Neuronal Modeling”. In: 1989.

[38] Jurij Mihelic and Uros Cibej. “Experimental Comparison Of Matrix Algorithms For Dataflow Computer Architecture”. In: *Acta Electrotechnica et Informatica* (2018). DOI: `https://doi.org/10.15546/AEEI-2018-0025`.

[39] Anisleidy González Mitjans et al. “Accurate and Efficient Simulation of Very High-Dimensional Neural Mass Models with Distributed-Delay Connectome Tensors”. In: *NeuroImage* 274 (July 1, 2023), p. 120137. ISSN: 1053-8119. DOI: 10.1016/j.neuroimage.2023.120137. (Visited on 11/22/2023).

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

[41] Wieslaw Lucjan Nowinski. “Evolution of Human Brain Atlases in Terms of Content, Applications, Functionality, and Availability”. In: *Neuroinformatics* 19 (2020), pp. 1–22. DOI: https://doi.org/10.1007/s12021-020-09481-9.

[42] Neri Van Otten. *Multilayer Perceptron Explained And How To Train and Optimise MLPs*. 2024. URL: https://spotintelligence.com/2024/02/20/multilayer-perceptron-mlp/ (visited on 02/20/2024).

[43] 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. DOI: 10.3389/fncom.2022.866517. (Visited on 06/14/2024).

[44] Leon Paula et al. “The Virtual Brain: a simulator of primate brain network dynamics”. In: *Frontiers in Neuroinformatics* 7.10 (2013). DOI: https://doi.org/10.3389/fninf.2013.00010.

[45] 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). DOI: https://doi.org/10.3390/app13031279.

[46] Timothée Proix et al. “Individual brain structure and modelling predict seizure propagation”. In: *Brain* 140.3 (2017), pp. 641–654. DOI: https://doi.org/10.1093/brain/awx004.

[47] 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).

[48] Dirk Ridder et al. “NeuroDots: From Single-Target to Brain-Network Modulation: Why and What Is Needed?” In: *Neuromodulation: Technology at the Neural Interface* 27 (Apr. 2024). DOI: http://dx.doi.org/10.1016/j.neurom.2024.01.003.

[49] Petra Ritter et al. “The Virtual Brain Integrates Computational Modeling and Multimodal Neuroimaging”. In: *Brain connectivity* 3 2 (2013), pp. 121–145. DOI: https://doi.org/10.1089/brain.2012.0120.

[50] Paula Sanz-Leon et al. “Mathematical framework for large-scale brain network modeling in The Virtual Brain”. In: *NeuroImage* 111 (2015), pp. 385–430. DOI: https://doi.org/10.1016/j.neuroimage.2015.01.002.

[51] Michael Schirner et al. “Brain simulation as a cloud service: The Virtual Brain on EBRAINS”. In: *NeuroImage* 251 (2022). DOI: https://doi.org/10.1016/j.neuroimage.2022.118973.

[52] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. “FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review”. In: *IEEE Access* 7 (2019), pp. 7823–7859. DOI: 10.1109/ACCESS.2018.2890150.

[53] Maxeler Technologies. *Maxeler Technologies*. 2024. URL: https://www.maxeler.com/ (visited on 07/17/2024).

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

[55] Tech Power UP. *Quadro RTX6000*. 2024. URL: https://www.techpowerup.com/gpu-specs/quadro-rtx-6000.c3307 (visited on 06/24/2024).

[56] Haifeng Wang and Teng Wu. “Knowledge-Enhanced Deep Learning for Wind-Induced Nonlinear Structural Dynamic Analysis”. In: *Journal of Structural Engineering* 146 (2020). DOI: 10.1061/(ASCE)ST.1943-541X.0002802.

[57] Huifang E Wang et al. “Virtual brain twins: from basic neuroscience to clinical use”. In: *National Science Review* 11.5 (2024), nwae079. DOI: https://doi.org/10.1093/nsr/nwae079.

[58] Huifang E. Wang et al. “Delineating epileptogenic networks using brain imaging data and personalized modeling in drug-resistant epilepsy”. In: *Science Translational Medicine* 15.680 (2023), eabp8982. DOI: `10.1126/scitranslmed.abp8982`.

[59] Junsong Wang et al. “Suppressing epileptic activity in a neural mass model using a closed-loop proportional-integral controller”. In: *Scientific Reports* 6 (2016). DOI: `https://doi.org/10.1038/srep27344`.

[60] 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. DOI: `https://doi-org.tudelft.idm.oclc.org/10.1016/S0006-3495(72)86068-5`.

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

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

[63] Xilinx. *Versal ACAP System and Solution Planning Methodology Guide (UG1504)*. 2022. URL: `https://docs.xilinx.com/r/2021.2-English/ug1504-acap-system-solution-planning-methodology` (visited on 02/02/2024).

[64] Xilinx. *Versal AI Core Series Product Selection Guide*. 2024. URL: `https://docs.xilinx.com/v/u/en-US/versal-ai-core-product-selection-guidel` (visited on 06/12/2024).

[65] Xilinx. *Versal: The First Adaptive Compute Acceleration Platform (ACAP)*. 2020. URL: `https://docs.xilinx.com/v/u/en-US/wp505-versal-acap` (visited on 06/12/2024).