UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Master's Degree in Physics of Data

Final Dissertation

Beam Control with Fast Reinforcement Learning

Inference at the Edge

Thesis supervisor

Prof. Andrea Triossi

Thesis co-supervisor

Dott. Luca Scomparin

Candidate

Michail Sapkas

Academic Year 2023/2024

Contents

1 Introduction **1**

2 The Beam **3**
2.1 Synchrotron Accelerators . . . 3
2.2 Beam Physics . . . 4
2.3 Synchrotron Radiation . . . 4
2.4 RF Cavity and Bunching . . . 5
2.5 Microbunching Instabilities . . . 6
2.6 The KAPTURE-2 Data Acquisition System . . . 7

3 Reinforcement Learning **9**
3.1 Motivation . . . 9
3.2 Main Concepts . . . 9
3.3 Formal Definitions . . . 10
3.4 Actor Critic Method . . . 12
3.4.1 Actor . . . 12
3.4.2 Critic . . . 12
3.4.3 Actor-Critic Algorithm . . . 13

4 Principles of Computing and the VCK190 Evaluation Kit **15**
4.1 CPUs . . . 15
4.2 GPUs . . . 16
4.3 FPGAs . . . 16
4.4 DMAs . . . 17
4.5 AXI4-Stream and the TLAST . . . 18
4.6 The Versal VCK190 Evaluation Kit . . . 19
4.7 Vivado HDL Design, Applications and the Vitis IDE . . . 20
4.7.1 Vivado HDL Design . . . 20
4.7.2 Petalinux . . . 21
4.7.3 Creating Applications and Compiling in the Vitis IDE . . . 21
4.7.4 Development Flow Overview . . . 22
4.8 The Promise . . . 22

5 AI Engine **23**
5.1 The AI Engine Tile . . . 23
5.2 C++ Intrinsics . . . 25
5.3 Graphs and Kernel Programming . . . 25
5.4 Connectivity . . . 26
5.4.1 AXI4-Stream Connections . . . 26
5.4.2 Cascade Stream Connections . . . 26
5.4.3 Windows . . . 26
5.4.4 Kernel Connectivity . . . 27
5.5 Memory Banks . . . 27

5.6 Pipelining . . . 27
5.7 Parallelism . . . 28

6 The Existing Implementation 29
6.1 The Experiment . . . 29
6.2 Timing . . . 29
6.3 Existing Implementation on the AI Engine . . . 30
6.3.1 Storing Inputs and Outputs to Train . . . 31
6.3.2 Performing Offline Training . . . 31
6.4 Feedback Modification . . . 31

7 Benchmarking the AI Engine Hardware 33
7.1 Motivation . . . 33
7.2 The Simulation 'Arsenal' . . . 33
7.2.1 x86 Simulation . . . 33
7.2.2 AI Engine Emulation . . . 34
7.2.3 The Vitis Analyzer . . . 34
7.3 Custom Hardware Benchmark . . . 35
7.3.1 Reserving Memory for Tests . . . 35
7.3.2 Custom PL Modules . . . 36
7.3.3 Top Level Test Manager in Python . . . 37
7.3.4 Analyzing the Results . . . 39
7.4 Dot Product Test . . . 39

8 The GRU 43
8.1 Motivation . . . 43
8.2 Introduction to the GRU . . . 44
8.2.1 The Reset and Update Gates . . . 44
8.2.2 The Candidate Hidden State Gate . . . 45
8.2.3 The New Hidden State Gate . . . 46
8.3 Implementation on the AI Engine . . . 46
8.3.1 Memory . . . 47
8.3.2 The graph and Connections Between Kernels . . . 48
8.3.3 A Special Connection: The Hidden State Feedback . . . 49
8.3.4 Pipelining the Matrix-Vector Multiplication . . . 50
8.3.5 Activation Functions . . . 52
8.3.6 Passing Real Time Parameters to the Subgraph . . . 59
8.4 The Custom GRU Cell in PyTorch . . . 59
8.4.1 The Custom Activation Functions . . . 60
8.5 Benchmarking the GRU in the existing implementation . . . 62

9 Conclusions 64
9.1 Overview and Final Remarks . . . 64
9.2 Future Work . . . 65
9.3 Beyond this Implementation . . . 66

Chapter 1

Introduction

In the last years, the world has witnessed the rise of Artificial Intelligence technologies with the latest being the Large Language models. The success of these models is attributed to the plentiful availability of data and the computational power to train them. However, it is becoming increasingly clear that scaling up the models will be the next big challenge of the field, since both data, computational power and electricity resources are not infinite. Furthermore, AI on-the edge is gaining momentum as it is very energy efficient and ensures data privacy by localised operation. Enter the next player in the game, that promises data acquisition and processing on site, less power consumption and decoupling cloud resources: custom hardware accelerators. Undoubtedly, the next era of AI will have to be 'Hardware Aware'. That being said, scientific fields have to adapt and leverage the new tools available and this thesis is a step towards that direction. A multidisciplinary approach is nowadays absolutely necessary, as the field of AI is not only about algorithms and data, but also about hardware and software.

This thesis follows my internship at the Institute for Data Processing and Electronics (IPE) at campus North of the Karlsruhe Institute of Technology (KIT) were I had the opportunity to develop deep learning models on the AMD Xilinx Versal AI Engine. During my stay at IPE, I was able to delve into the work done by Michele Caselle's group on trying to induce control of fast beam instabilities at the Karlsruhe Research Accelerator (KARA) synchrotron light source and accelerator test facility. I expanded my skillset and experience by working closely with Luca Scomparin and had the privilege to explore their implementation of the project.

The project required understanding the the Versal platform and the AI Engine architecture. I was needed to develop code in C++, Python, and Verilog. Also, I had to usage of Vitis and Vivado which are the official development and simulation tools of AMD Xilinx and target their platforms. Moreover, understanding the contex of the project such as the physics of the beam and the control system of the KARA light source. Most importantly, the ability to understand and develop deep learning models in the reinforcement learning framework that they are deployed in.

The goal of this thesis is to develop a Gated Recurrent Unit (GRU) model on the AI Engine. But in order to reach that point, I will have to discuss: First, the context in which I will be using the GRU model, which is the KARA synchrotron light source. Second, I will provide a brief overview of the modern choices of hardware accelerators and how the Versal AI Engine fits in the picture, specifically the VCK190 evaluation platform. Third, I will discuss the existing implementation of the AI Engine and the modifications that were made during an actual deployment on KARA. Fourth, I will expand on what is the AI Engine, how it is programmed and how it is controlled. Fifth, I will present the custom testing tool that was developed in order to test the hardware and software. Finally, I will introduce the GRU model and how it was implemented on the AI Engine.

Due to the complexity and multidisciplinary nature of the project, there will be a lot of details left out of this thesis. The physics of the beam, the working principle of the data acquisition system and the Programable Logic side of the implementation are beyond the scope of this work, but will be

introduced to the extent necessary for the comprehension of this thesis.

This is a field in its infancy. The Versal is a very young platform and the AI Engine is a very new architecture and there is a clear lack of tools to design and develop machine learning applications and the ones used are old and trying to cover a wide field of applications is not very effective and they are not user friendly. Nevertheless, the promise of the AI Engine is great and the potential is huge. Beam control could be key for many scientific applications such as Nuclear Fusion and control in general as a Reinforcement Learning problem, has great potential to revolutionize machine calibration and control in noisy environments. Such an environment could be, for instance, the precise control potentials for driving qubit gates in Quantum Computing.

This thesis is a small step towards that direction and I can't wait to witness the leap that will be made in the next years.

Aknowledgements

I would like to thank professor Andrea Triossi, who somehow managed to find an internship with the exact focus that I was interested in: building AI applications, from the bottom up in hardware. More than that he was always available to answer my questions and provide guidance and was supportive of my work. He really believed in me and I am grateful for that.

Special thanks go to Luca Scomparin, who was my mentor during the internship and who was always available to answer my questions and provide guidance. I am amazed of the work that he has done and I am grateful for the opportunity to work with him. A learned so much and I am sure that he will have a great career ahead of him.

Many thanks to Michele Caselle for making this happen, his support and constant appreciation and encouragement.

I would also like to thank the people at IPE, who were supportive, friendly and always willing to help or have a chat: Nour Sharif, Ziyu Xie, Andreas Kopmann, Timo Dritschler, Simon Kraft, Fabian Hummer, Olena Manzhura, Zewei Lu, Lars Eisenblätter and Michael Zapf.

Finally, the AI4Accelerators group at the Institute for Beam Physics and Technologies that held great talks and were always available to help: Andrea Santamaria Garcia and Chenran Xu.

Chapter 2

Electron Beam Dynamics

2.1 Synchrotron Accelerators

Synchrotron accelerators were first developed in the 1940s and they were a fundamental instrument for discoveries in a wide variety of fields, like high energy physics, medicine, chemistry, and material science. The name 'synchrotron' comes from the fact that the magnetic field is synchronized with the energy of the particles. The accelerated charged particles are stored by making them followed a closed, approximately circular, path. To stay on this path, they experience centripetal acceleration that leads to emission of radiation, called synchrotron radiation. In electron accelerators, the light emitted by the beam can achieve a wide range of frequencies with high intensity and is used for various experiments in physics, chemistry and biology. Figure 2.1 shows a graphical representation of the storage ring and how experimental areas are placed around the accelerator to use the emitted light which is called synchrotron radiation. The creation of this chapter was based on the work done by Tobias Boltz [1]. Another excellent source of material for accelerators is the book 'Accelerator Physics' [2].

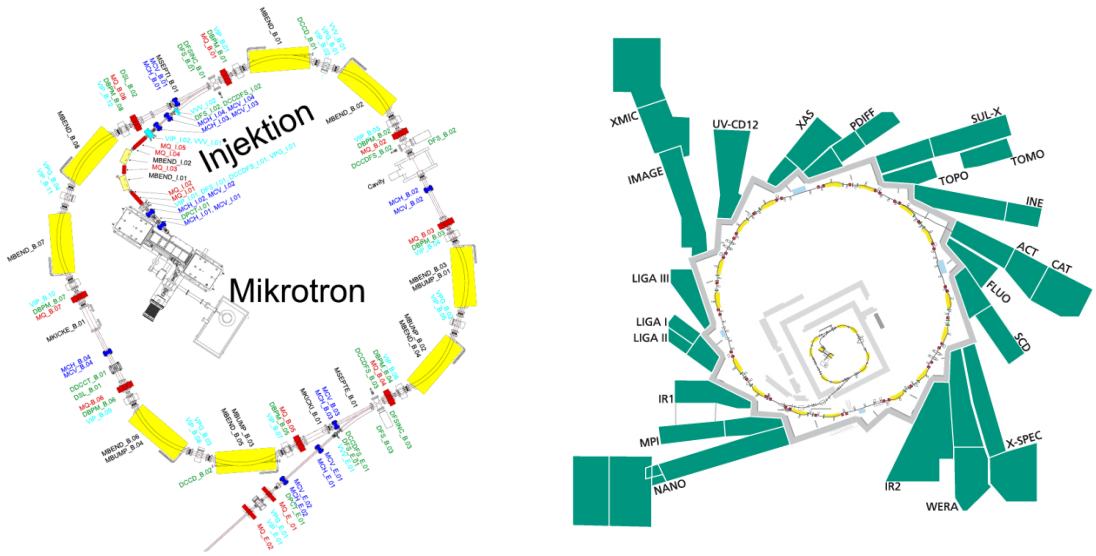


Figure 2.1: On the left, a schematic [3] of the main ring of the KARA synchrotron accelerator at KIT. Inside the main storage ring, the booster ring is used to capture emitting electrons and start their acceleration. The beam is then injected into the storage ring where it gets accelerated to its final energy and stored in a stable orbit. On the right, the figure [4] of how experiments are placed around the storage ring of the KARA synchrotron accelerator. Beamlines are used to direct the emitted radiation to the experiments. One of these locations is the KAPTURE-2 system [5], which is used to capture the emitted radiation and study the microbunching instability.

2.2 Fundamentals of Beam Dynamics at Synchrotrons

Particles are accelerated and stored through the ring by using two main types of components: electric and magnetic fields. The electric fields are used to provide energy to the beam electrons, while the magnetic fields are used to steer and focus the beam. The relevant force acting on a charged particle in an electromagnetic field is the Lorentz force:

$$F_L = q(E + v \times B) \tag{2.1}$$

When a charged particle is placed in a magnetic field, the Lorentz force acts as the centripetal force on the particle. This is used to store the particles in a circular closed path. In the case of an ultrarelativistic particle on a sufficiently large accelerator, in first approximation the electric field contribution can be discarded, leading to:

$$F_c = F_L \Rightarrow \frac{mv^2}{R} = qvB \tag{2.2}$$

where m denotes the particle's mass and R is the radius of the trajectory.

During its motion from position r_1 to r_2, the particle gains the energy and as dr and v are parallel, the term $(v \times B)dr$ is zero:

$$dW = \int_{r_1}^{r_2} F_L \cdot dr = q \int_{r_1}^{r_2} (E + v \times B) \cdot dr = q \int_{r_1}^{r_2} E \cdot dr = q\Delta V \tag{2.3}$$

where ΔV is the potential difference of the electric field. The energy gained by the particle is equal to the work done by the electric field.

2.3 Synchrotron Radiation

It is know by Maxwell's equations that every time a charged particle changes its acceleration, and therefore its velocity vector, it emits electromagnetic radiation as discussed in section 2.2. In the particular case of a ultrarelativistic particle moving in a circular path, the emitted radiation is focused in a narrow cone tangential to the particle's trajectory. This is because of the contraction and dilation of the electric field in the direction of motion of the particle. Basically the emitted radiation behind the particle will seem 'squashed' and in front of the particle will seem 'stretched' in a cone.

For relativistic energies, the cone's half opening angle is approximately given by

$$\theta \approx \frac{1}{\gamma} \tag{2.4}$$

with the relativistic Lorentz factor $\gamma = \frac{1}{\sqrt{1-\beta^2}}$ and the velocity of the electron $\beta = \frac{v}{c}$.

This 'directionality' of the emitted synchrotron radiation is also an advantage of the synchrotron light sources. The emitted radiation can be focused and directed to the experiments. The instantaneous power emitted by a single electron passing through a bending magnet is given by:

$$P = \frac{e^2 c}{6\pi\epsilon_0} \frac{1}{(m_e c^2)^4} \frac{E^4}{R^2} \tag{2.5}$$

with the elementary charge e, the electron rest mass m_e, and the dielectric constant of vacuum ϵ_0.

And thus we arrive to the crucial point that makes synchrotron machines work. The emitted radiation is a loss of energy for the electron beam:

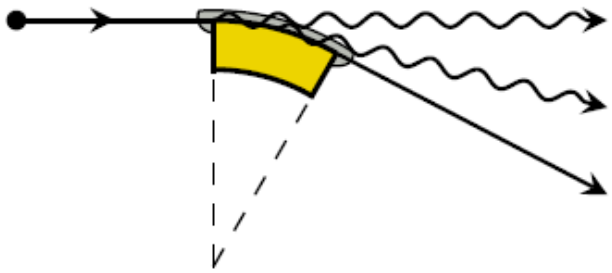


Figure 2.2: Synchrotron radiation emitted by a charged particle moving in a circular path. The emitted radiation is focused in a narrow cone tangential to the particle's trajectory. Image credited to Tobias Boltz [1]

$$\Delta E = \oint P dt = \frac{2\pi R}{c} P \tag{2.6}$$

which has to be compensated by the RF system of the storage ring. In the next section we'll see that by tuning the RF cavity, we can achieve precise control over the beam and compensate for the energy lost due to the emitted radiation.

2.4 RF Cavity and Bunching

By bending the beam the electrons emit synchrotron radiation and lose energy. This energy loss has to be compensated by the RF system, but contrary to what one might think, the RF system does not operate with a steady voltage difference. What actually happens is that the RF cavity is driven in a sinusoidal manner, this is necessary because if the field was constant, the particles would not be accelerated due to the conservativity of the field. The voltage inside an RF cavity is given by:

$$V_{RF}(t) = V_0 \sin(\omega_{RF} t + \phi_0) \tag{2.7}$$

where V_0 is the peak voltage, ω_{RF} is the angular frequency, and ϕ is the phase of the RF cavity. A synchrotron operates the phase of the RF cavity so that a particle that arrives at the RF cavity in the correct timing will gain as much energy as it had lost due to the emitted radiation. The energy gained by the particle when it passes through the RF cavity modulation is given by:

$$\Delta E = q V_0 \sin(\phi_e) \tag{2.8}$$

By using equation 2.2 but for relativistic particles, the radius of the particle beam can be calculated as:

$$R \approx \frac{pc}{qcB} \approx \frac{E}{qcB} \tag{2.9}$$

Now, we would like to keep the beam stable in the same radius R. This corresponds to finding an exact RF frequency that will allow the accelerating voltage to oscillate such that particles will arrive at a constant phase, balancing the energy lost due to the emitted radiation. This frequency f_{RF} is an integer multiple of the revolution frequency of the beam f_{rev} and is given by:

$$f_{RF} = h f_{rev} \tag{2.10}$$

where h is the harmonic number of the accelerator.

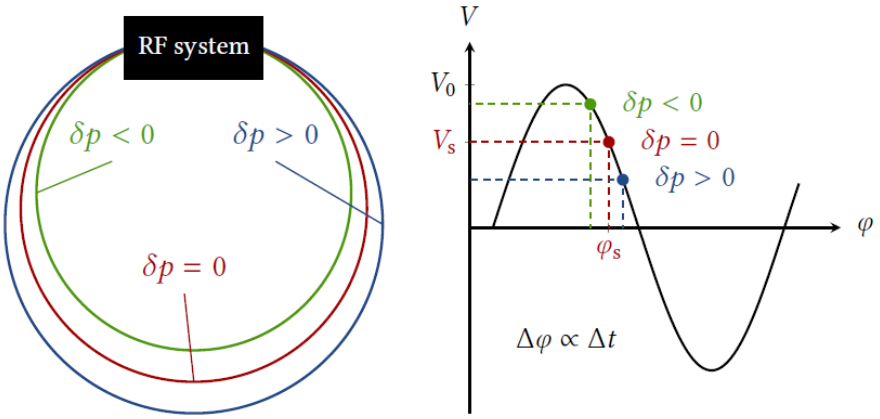


Figure 2.3: Schematic of the RF cavity and the bunches of electrons in the storage ring. The RF cavity is driven at a frequency that is a multiple of the revolution frequency of the beam. Particles that arrive at the RF cavity at the correct timing will gain a total of 0 momentum, while particles that arrive earlier or later will gain or lose energy. Image credited to Tobias Boltz [1]

The sinusoidal modulation of the RF cavity is a key feature of the synchrotron accelerator because it allows for the creation of a 'focusing effect' on the beam. This effect can be intuitively understood with figure 2.3. Charged particles that arrive at the RF cavity in the correct timing will gain a total of 0 momentum, while particles that arrive earlier or later will gain or lose energy. An ideal particle that radiates as much energy as it gains during a revolution is called a synchronous particle. This way, the particles with lower energy than the synchronous particles will arrive late and thus be accelerated more than the particles with higher energy. This focusing effect is not only crucial for the beam to be stable, but also creates regions in the beam where the particles physically come together and form a bunch.

2.5 Microbunching Instabilities

Experiments at synchrotron light sources can benefit from higher intensity radiation by operating with high currents, i.e. by storing more electrons into the accelerator. Additionally, by reducing the size of the bunch, higher quality radiation and more coherent light can be produced. In practice this is done by adjusting the focusing and higher-order magnets.

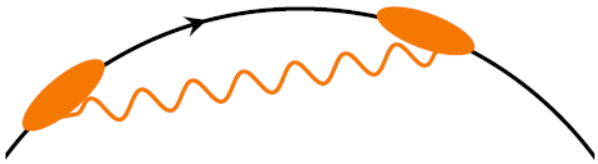


Figure 2.4: A bunch of electron interacting with the self-generated synchrotron radiation. The photons emitted can catch up with the bunch after the bending magnet and induce a longitudinal instability. Image credited to Tobias Boltz [1]

The more electrons are squeezed together, self-interaction phenomena start to appear inside the bunch and cause complex longitudinal dynamics. Specifically, when a bunch tries to pass through a bending magnet, the particles will emit synchrotron radiation and lose energy. The linear path from the tail to the head is a shorter path than the full circular trajectory of the beam, and thus the photons can catch up with the head particles. This simplified version of the process is also shown in figure 2.4.

The effect is non-linear and will cause the bunch to start oscillating longitudinally in an unstable manner. It is called the microbunching instability and can be detected as a fluctuation in the emitted radiation. The emitted radiation will fluctuate at varying timescales like shown in figure 2.5.

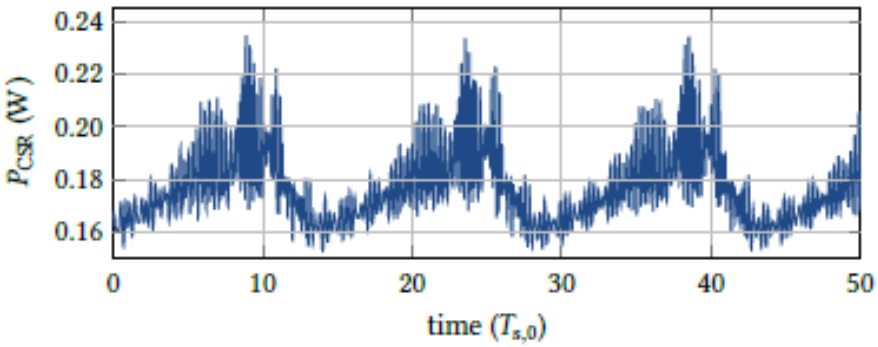


Figure 2.5: A simulation of the power emitted by the electron bunches in the storage ring. The 'noise' and the sawtooth pattern in the emitted radiation is due to the microbunching instability. Image credited to Tobias Boltz [1]

The time axis is measured in units of 'nominal synchrotron period' which was chosen at 14.285 milli seconds as a parameter of the simulation. Therefore, the fast dynamics can be at the range of tens of micro seconds.

The microbunching instability is strongly affected by the beam current in which we are operating. Typically, a storage ring that tries to operate with high currents and very compact bunches will at some point encounter a current threshold. After it hits that threshold, emitted radiation from the beam fluctuates in intensity, in a phenomena known as microbunching instability. More on microbunching instabilities can be found in [6].

2.6 The KAPTURE-2 Data Acquisition System

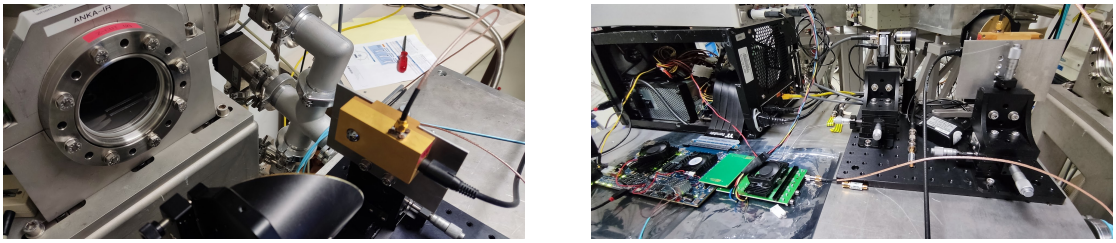


Figure 2.6: A photo of the KAPTURE-2 system. On the left image, light is coming from the steel tube on the left and gets steered to the detector. Optical fibers are used to guide the signal from the detector to the data acquisition system which then processes the signal and outputs the amplitude of the emitted radiation to the PC. [5]

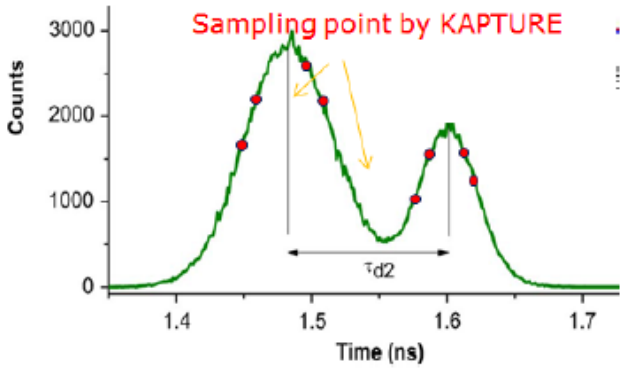


Figure 2.7: A graphical representation measurement of the emitted radiation by the KAPTURE-2 [5] system by a single passing bunch.

The Karlsruhe Pulse Taking Ultra-fast Readout Electronics [5] (KAPTURE) system is a data acquisition system developed at IPE KIT for the KARA storage ring. The system is designed to capture the amplitude of the synchrotron radiation emitted by the electron bunches in the storage ring. The emitted light passes through some IR filtering and then gets focused on the detector. The detector is sensitive from tens of GHz up to 2 THz, has a response time of 100ps and can acquire detection signal fast enough to distinguish between individual bunches.

This system was specifically engineered to capture the microbunching instability in the KARA storage ring. By using KAPTURE-2, we can measure the effect of the instability on the emitted radiation for each individual bunch. Being able to study the behavior of the beam in the THz temporal regime is crucial for trying to control the beam. The system has the ability to monitor multiple bunches at the same time for several seconds, using multiple channels. In figure 2.6 a KAPTURE-2 system is shown.

Figure 2.7 shows this process. The signal from a synchrotron is composed of several peaks that are tens of picosecond long, seperated by several nanoseconds. In order to sample them properly KAPTURE-2 uses an interleaving procedure with the capability of sampling at a selectable time with 3 ps resolution.

Four samples of data are produced for a single bunch at each revolution of the beam by the KAPTURE-2 system and are then transferred with optical fiber to the inference hardware in real time.

Chapter 3

Reinforcement Learning

3.1 Motivation

This chapter aims to given a very basic understanding of the algorithms and ideas used in order to train a reinforcement learning agent to control the RF system. It only severs as an introduction to the concepts and does not reflect the exact implementation used in the project.

In general, systems that deal with the control of machines or processes, mainly through feedbacks, fall into the engineering field of control theory. The main difference from traditional methods is that reinforcement learning (RL) is capable of learning from the interaction with the environment, which is a more general and flexible approach.

This approach is extremely useful in cases where variables exist that are difficult to model or are unknown and outside the control of the system. In this case, the synchrotron accelerator, variables such as the ambient temperature, the humidity, the power supply, the vacuum level and others can affect the performance of the machine. Since the RL agent learns from the interaction with the environment, it can adapt to these changes and optimize the performance of the machine.

Finally, controlling the machine means that actions taken by the agent have consequences on the beam. In order to train the model offline, there is a need for a good enough physics simulation of the events taking place in the machine. Unfortunately, such simulations are either very slow to render, too computationally expensive or too different from the real world. So using a online-trained reinforcement learning approach is in some cases the best way to control the machine. In this case, the data acquisition is happening so fast that we have a plenitude of data to train on.

This will be a very high-level overview of the reinforcement learning method. The main goal is to provide a general understanding of the Actor-Critic method, which is the method used in the implementation of the control system of the synchrotron accelerator and how Neural Networks are used to approximate the policy and the value function. Neural Networks are not the only models that can be use as discussed in [7], but they are the more robust.

3.2 Main Concepts

Reinforcement Learning is a type of machine learning technique together with supervised and unsupervised learning. The loss function is used to train the surrogate model, while the reward function connects the model with the environment in which it operates.

The main difference between reinforcement learning and the other types of machine learning is that the model learns from the interaction with the environment, instead of learning from a static dataset. Every time the model performs a modulation on the RF cavity, it effectively creates the data it needs to learn. This interaction-learning loop is shown in figure 3.1

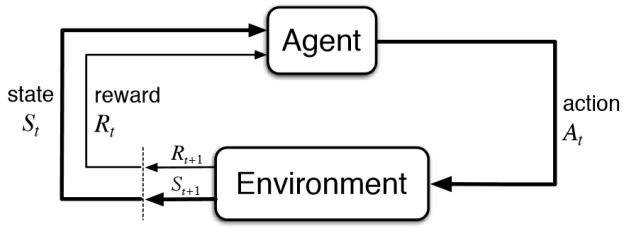


Figure 3.1: A graphical representation [8] of the reinforcement learning agent interacting with the environment. It is a loop where the agent takes an action, the environment responds with a new state and a reward, and the agent learns from this interaction.

The main elements of RL are:

- Agent: The entity that learns and makes decisions by interacting with its environment. It observes the environment, takes actions, and these actions can change the environment.
- Environment: The surroundings or conditions the agent interacts with. The environment may change either due to the agent's actions or on its own.
- State: A state is a complete description of the environment at a specific moment. The environment is partially observed due to the presence of hidden variables. Will be denoted as S.
- Action Space: The set of all possible actions the agent can take in a given environment. If the actions are finite, the action space is discrete; if infinite, it's continuous.
- Reward: A reward is a feedback signal indicating how well the agent is performing at a given step.
- Episode: A sequence of steps from the initial state to the terminal state. The agent interacts with the environment to learn a policy.
- Cumulative Reward: The sum of all rewards received by the agent over a sequence of steps in an episode.
- Policy: The agent's strategy or behavior, mapping states to actions. It defines the agent's way of acting in the environment. In our case the policy is the modulation of the RF cavity and it is denoted by π.
- Value Function: Predicts the total future reward from a given state. It helps evaluate the desirability of a state. The value function can be used to select actions. Denoted like $V(s)$ and so it is state dependent.

3.3 Formal Definitions

Reward Function

Since in this implementation we are dealing with a stabilization of chaotic perturbations of the beam, a logical choice for the reward function [9] would be to try and minimize the deviation of the beam from a reference intensity. This simply translates to minimizing the variance of each observation O at an episode step with respect to the mean of that episode:

$$R_i = -(O_i - \bar{O})^2 \tag{3.1}$$

Cumulative Reward

The reinforcement learning agent will interact with the environment in a discrete time window of $t = 0, 1, 2, 3, ...$ which all together will consist an episode. If we allowed the agent to extract all its reward in a single step then the agent will choose the move that will maximize the reward in that step. However, this is not the optimal strategy because the optimal move could be to sacrifice some reward in the current step to gain more reward in the future. That is why we introduce the concept of cumulative reward which is simply the reward acquired in all the episode steps. In order to somehow constrict the moves of the agent in time, we also introduce a discount factor $\gamma < 1$ as a hyperparameter that will control the importance of future rewards. So the cumulative reward is defined as:

$$CR_t = R_0 + \gamma R_1 + \gamma^2 R_2 + \gamma^3 R_3 + ... = \sum_{k=0}^{\infty} \gamma^k R_k \tag{3.2}$$

By controlling the range of γ between 0 and 1, we can control the importance that the agent assigns to immediate rewards versus future rewards.

Policy

The policy is the strategy that the agent uses to determine the next action to take. It is a mapping from states to actions. The policy can be deterministic or stochastic. In our case, the policy is the modulation of the RF cavity. The policy is denoted by π and it is a function that maps states to actions:

$$\pi : S \to A \tag{3.3}$$

In general we can express a policy that can be learned using some kind of model parameterized by θ like this:

$$\pi_\theta(a|s) \sim \pi[A_t = a|S_t = s, \theta_t = \theta] \tag{3.4}$$

Where we simply expressed that the probability of taking an action a in state s is parameterized by θ.

Value Function

The cumulative discounted reward can be difficult to calculate because it implies that we have to know the future rewards. However, we can approximate it by using the value function. The value function keeps track of the expected cumulative reward that the agent will receive by being in a given state. It quantifies how good it is for the agent to be in a given state s. This is dependent on the policy π in the sense that the actions it has taken lead to this state. The value function is defined as the expected value of the cumulative reward given that the agent is in state s:

$$V^\pi(s) = E_\pi[CR_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_k|S_t = s\right] \tag{3.5}$$

Temporal Difference Error (TD Error)

In order to train an approximation of a value function we can use its recursive properties, i.e. linking past and future expected cumulative rewards. This is called the Time Difference error, denoted as δ and defined as:

$$\delta = R_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \tag{3.6}$$

What we will be using in practice is the delta quantity which is the current reward plus the discounted expected future reward minus the value function. This is a measure of how off our current reward is compared to what we would expect.

3.4 Actor Critic Method

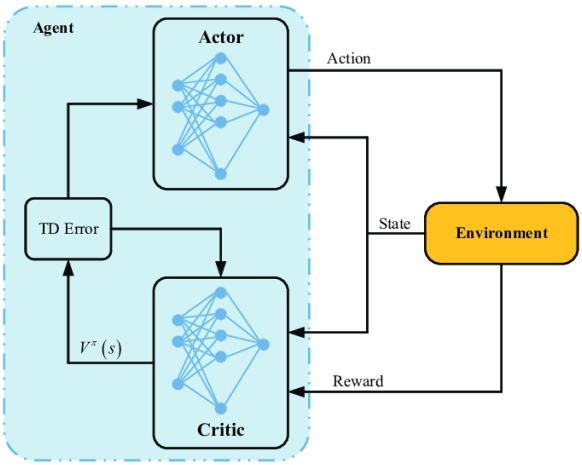


Figure 3.2: The Actor-Critic method [10]. The actor learns the policy and the critic learns the value function.

But how do we acquire an actual function for the policy π and the value function V?. Although we could use Bayesian methods the state of the art method is to leverage the power of backpropagation and stochastic gradient descent. This way we don't have to make any assumptions, explore infinite state spaces or try to write down actual functions. All we have to do is to use a neural network which will learn to approximate the policy and the value function.

The Actor-Critic method uses two deep neural networks, one for the policy and one for the value function. The subset of RL using this function approximation is called deep-RL.

3.4.1 Actor

The actor is the policy-based method that learns the optimal policy. The actor is the model that actually runs inference on the hardware. It takes as inputs the observations of intensity and outputs the modulation of the RF cavity.

The loss function of the actor is defined as the negative log likelihood of the action taken by the policy multiplied by the advantage function defined as:

$$L_{actor} = -E_{\pi\theta}[\log \pi_\theta(a|s)\delta_t] \tag{3.7}$$

And so by using this loss function the model will update its parameters (weights) to increase the log likelihood of actions, scaled by a learning rate α, that lead to higher rewards:

$$\Delta\theta = \alpha\nabla_\theta L_{actor} = \alpha\nabla_\theta \log \pi_\theta(a_t|s_t)\delta_t \tag{3.8}$$

3.4.2 Critic

There is only one last piece of the puzzle missing. The actor network needs an estimate of the value function in order to calculate the TD error and thus to update the variational parameters of the policy model. This is where the critic network comes in place which its sole purpose is to estimate the value function. The critic network aims to minimize the error in predicting the value function. In order to do so it use the squared TD error as a loss function:

$$L_{critic} = \frac{1}{2}(\delta)^2 = \frac{1}{2}(R_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t))^2 \tag{3.9}$$

And thus the weights of the critic network are updated by gradient descent:

$$\Delta\theta = \alpha \nabla_{\theta} L_{critic} = \alpha \delta_t \nabla_{\theta} V_{\theta}^{\pi}(s_t) \tag{3.10}$$

The actions of the actor network can be complimented with a term that encourages exploration. Although in standard literature this is called the entropy term, in our case what its actually used is a noise term that is added to the output of the actor network. This noise term is a random number drawn from a normal distribution with a mean of 0 and a standard deviation given as a hyperparameter. This term is added to the output of the actor network and it is used to encourage exploration of the state space.

The power of Neural Networks and backpropagation

At this point we are free to leverage the power of backpropagation to train complex neural networks. If we have access to the delta quantity, everything else must be able to be handled by frameworks like TensorFlow or PyTorch. Although the simplest models for the actor and the critic are feedforward neural networks, we can use more complex models like convolutional neural networks or recurrent neural networks. We can also use more advanced techniques like batch normalization, dropout, weight initialization and others.

3.4.3 Actor-Critic Algorithm

Finally, in the figure 3.3 below I present the algorithm in steps hoping that it will clear the landscape of the Actor-Critic method as take from the book 'Reinforcement Learning: An Introduction' by Richard S. Sutton and Andrew G. Barto [11]:

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Figure 3.3: The actor critic algorithm

Disclaimer on the implementation

While this provides the basic sketch of the Actor-Critic method, the actual implementation is different due to the fact that the model runs inference online and then it is trained offline on a batch of data that were collected during the inference called 'experiences'. This is done mainly for the reason that backpropagation is far more complex to implement from scratch on hardware than inference.

Whole episodes are kept in memory as states (observations), actions and rewards. I only had limited contact with the python code that implements the Actor-Critic method, and thus I cannot claim full knowledge of the actual implementation.

The python code uses a Proximal Policy Optimization (PPO) algorithm and it is implemented using the Stable Baselines 3 library [12]

Chapter 4

Principles of Computing and the VCK190 Evaluation Kit

In this section I will briefly discuss the different computing paradigms which in the end also correspond to physically different hardware. The purpose of this introduction is to justify the need for the new computing paradigm brought to the market by AMD Xilinx Versal in our specifically implemented use case.

Modern computing devices have achieved extremely fast computation times up to the point that the bottleneck of the system is the data transfer between the different computing elements. This is due to the fact that the data transfer between memory reads and writes between the different computing modules on chip is much slower than the computation time of the elements themselves. Data acquisition systems used in physics experiments have to leverage modern computing platforms and they inevitably will encounter the limitations imposed by hardware as discussed in [13].

4.1 CPUs

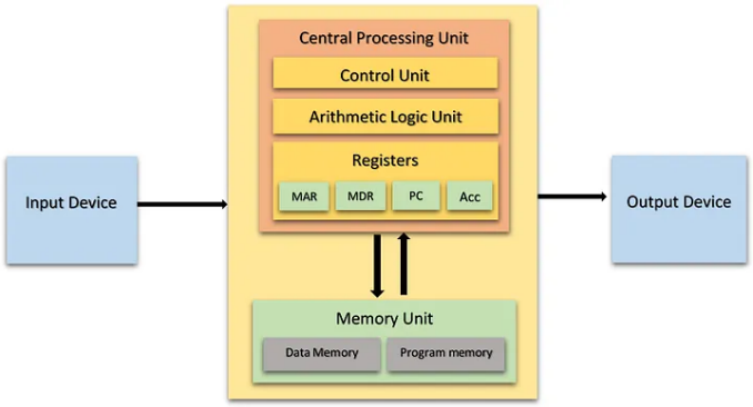


Figure 4.1: The Central Processing Unit [14] is limited by the memory bottleneck architecture

The Central Processing Unit is used as a general purpose computing element. It is designed to be able to execute a wide range of instructions and is optimized for sequential execution. Classical CPU-centered system implementations (such as the computer you are reading this thesis on right now) usually use the 'Von Neumann' architecture (figure 4.1), where the CPU is the main element of the system and the memory is separated from the CPU. This separation of memory and CPU leads to a bottleneck in the system, as the CPU has to wait for data to be transferred between the memory and the CPU. This can limit the performance of the system, especially in applications that require a large amount of data to be processed. Modern CPUs are extremely complex and contain a large number of

different components, including multiple cores, caches, DMAs (discussed further down) and vectorized processing units. This complexity could act as a disadvantage in cases that the user needs to perform a specific and highly optimized task, as the CPU, built for general purpose computing, contains a lot of overhead that is not needed for the specific task. Other modern techniques that include probabilities like prediction branching and out-of-order execution can also be a disadvantage in cases that the user needs to have a deterministic behavior.

Another disadvantage of the CPU is that it is not designed to be massively parallel. This means that it is not well suited for tasks that require a large number of computations to be performed simultaneously. This is because the CPU is designed to execute instructions sequentially, one after the other, and is not optimized for parallel execution. This is a major limitation in many applications, such as machine learning, where large amounts of data need to be processed in parallel. This also limits the ability to create effective data pipelines and to scale the performance of the system.

4.2 GPUs

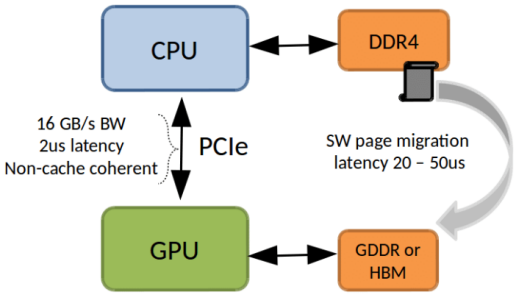


Figure 4.2: A GPU connected to the rest of the computing system via PCI-Express and sharing CPU memory [15]

The Graphics Processing Unit is a specialized computing element that is optimized for parallel computation. A GPU can be considered as a large number of small processing cores, that are optimized for performing the same operation on a large number of data points simultaneously. Modern GPUs contain in the order of a few thousands cores, making them extremely powerful. Moreover, they have their own dedicated memory that is optimized for high-speed data transfer, but only between GPU cores and the GPU memory.

Although GPUs can do 'embarrassingly parallel' tasks very efficiently, the main problem of transferring data from the main memory to the GPU memory is still present. This is because the CPU has to transfer the data to the GPU memory, which can be a slow process for a number of reasons. First, the CPU has to execute the instruction to transfer the data, which means that while this particular set of data is being transferred, the CPU cannot do any other work. Second, the data has to be transferred over the PCI Express bus (figure 4.2), which can act as a bottleneck of the system. Finally, the data has to feed some kind of a temporary buffer in the GPU memory, before computation starts, and this is usually limited by the design of the hardware itself.

So in the end, a consumer GPU mounted on a PCIe slot is not a good solution for real-time data processing, as the data transfer latency is too high. This is why GPUs are used in applications that require a large amount of data to be processed in parallel, such as machine learning, image processing, and scientific computing, but not in real-time applications. Another way to put it would be that GPUs are good for throughput computing, but not for latency-sensitive applications.

4.3 FPGAs

Field Programmable Gate Arrays are a type of computing element that is designed to be highly flexible and customizable. FPGAs are made up of a large number of different configurable logic blocks (CLBs) (figure 4.3) that can be configured to perform a wide range of different tasks. With FPGAs, the user

can design their own custom hardware, which can be optimized for the specific task that they need to perform. Reconfigurable hardware can be programmed using VHDL or Verilog, which are hardware description languages that allow the user to describe the logical behavior of the hardware at a low level. This allows the user to create custom combinatorial logic, state machines, and most importantly connect various inputs and outputs to the hardware. The usage of combinatorial logic can be highly parallel, as the user can create a large number of different paths for the data to flow through the hardware and so FPGAs can be used for highly parallel tasks.

One disadvantage of FPGAs is that, in general, they do not usually operate at the same clock speeds as CPUs or GPUs. This is because FPGAs are made up of a large number of different small memory and arithmetic blocks that are connected together using a large number of different switches. This means that the data has to travel a 'long' distance between the different blocks, which can introduce delays in the system. This is usually mitigated by using dedicated wires that used to carry the clock signal, but still, the clock speed of the FPGA is usually lower than that of a CPU or GPU.

The main disadvantage of FPGAs is that traditionally they were not optimized to do floating point operations, specifically multiplications, which are needed in many scientific and machine learning applications. This is because floating point operations are more complex than integer operations, and so they require a large number of different CLBs to be connected together. Until now, machine learning applications on FPGAs used fixed-point arithmetic, by 'quantizing' the floating point numbers to a fixed number of bits, which can lead to a loss of precision in the results. This is why FPGAs were not used in applications that require high precision, such as scientific computing. Modern FPGA designs contain Digital Signal Processor (DSPs) blocks which can handle multiply-and-accumulate operations.

A task that FPGAs are extremely good at is data transfer. They are actually the go to solution for high-speed data routing in communications and modern physics experiments because of their ability to handle serial data passthrough at high speeds. This is because the developer has complete control over the data flow and can optimize the hardware to minimize transfer latencies bypassing a lot of overhead that would be present in a CPU or GPU and also by utilizing specific modules that are specifically designed for data transfer.

4.4 DMAs

One of these modules is the Direct Memory Access (DMA) Controller which allows the user to transfer data between different memory locations without the need for the CPU to be involved. It is widely used in modern computing systems to transfer data between different devices and also between memory regions. The main advantage of DMA is that it allows the user to transfer data at high speeds, without the need for the CPU to be involved.

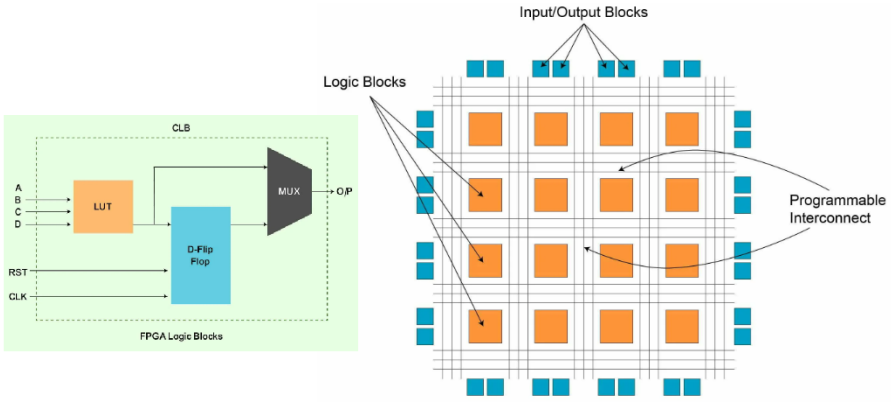


Figure 4.3: The Configurable Logic Block [16] of an FPGA is the main building block of the device. The data is routed through the programmable interconnect to the different blocks which consists the main fpga architecture [17]. A powerful architecture that allows for reconfigurable hardware via writing a hardware description language.

Dedicated DMAs can be found, for instance, transferring data from GPU memory to the GPU cores. The advantage of FPGAs is that the user can design their own custom DMA controller, that can be optimized for the specific task that they need to perform. This can lead to a significant improvement in performance.

The DMA controller functions similarly to a specialized CPU, with its primary role being the rapid transfer of data. In a machine equipped with DMA capability, the main CPU can delegate data transfer tasks to the DMA controller by issuing a few instructions and specifying the data to be moved. Once these instructions are given, the CPU can resume its other tasks, enabling parallel processing. While the CPU continues its operations, the DMA controller concurrently handles the data transfer, subsequently notifying the CPU upon completion of the transfer.

4.5 AXI4-Stream and the TLAST

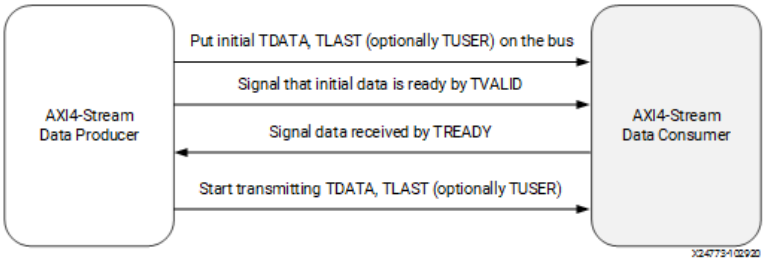


Figure 4.4: The AXI4-Stream interface handshake protocol [18]. The data is transferred between the master (producer) and the slave (consumer) using the TVALID and TREADY signals. Data is sent by the TDATA line. The TLAST signal is used to indicate to the consumer the end of the data stream.

There are different protocols to transfer data between different modules on an FPGA, and one of the most common is the AXI4-Stream which is a simple, high-speed, point-to-point interface.

AXI4-Stream is able to transfer from 32 bits up to a 512 bits of data per clock cycle. This is aligned with floating point numbers, as they are usually 32 or 64 bits long for single or double precision respectively. The AXI4-Stream is based on a very simple concept: the data is transferred between two modules, the master which sends data and the slave which receives the stream, using a handshake protocol. The handshake protocol can be also seen in figure 4.4 and following is a brief explanation of the signals:

- **TVALID**: This signal indicates that the data from the 'master' side is valid and should be read by the receiving module.
- **TREADY**: This signal indicates that the receiving 'slave' module is ready to accept the data on the data bus and should be read by the sending module.
- **TDATA**: This is the continuous data stream, which starts when both TVALID is high and TREADY is high. It will continue uninterrupted until either TVALID or TREADY goes low.
- **TLAST**: This signal indicates that the current data stream is the last one in the sequence. It is used to indicate the end of the data stream.

The handshake takes place when TVALID and TREADY are both high at the rising edge of a clock.

AMD Xilinx specific usage of TLAST in the AXI DMA and the AI Engines

The TLAST is an extremely important signal, as it allows the receiving module to know when the data stream is finished and it works in conjunction with the Xilinx AXI DMA controller [19]. The DMA controller can be programmed to execute a transfer of an arbitrary number of floats continuously but in order to properly end a transfer the TLAST signal must be received.

The same applies for the AI Engine kernels that are used in the Versal architecture. Although we haven't yet discussed the AI Engine, we will see that it boils down to kernels that continuously run a set of instructions until the TLAST signal is received and which will then terminate their execution. Thus, the usage of the TLAST signal will be used to gracefully stop the AI Engine kernels.

The fact that the TLAST is being used a stoping signal means that, when it is passed to the DMA producing data for the rest of the pipeline, it will have to be propagated through the pipeline and all the way to the last element that needs a TLAST.

4.6 The Versal VCK190 Evaluation Kit

We discussed the usage of different computing elements in modern computing systems, and briefly introduced some of the advantages and disadvantages of each, but what if we could combine all of the advantages of the different computing elements into a single system? This is the idea behind heterogeneous computing, which is the use of different computing elements in a single system.

To address this new way of computing, in 2018 AMD Xilinx introduced the first Adaptive Compute Acceleration Platform, commonly known as an ACAP or another term that is used to describe the same concept is 'heterogeneous computing'. The ACAP provides the robust functionality of an FPGA with adaptable programming for any application. More specifically, we will be using the Versal AI Core Series VCK190 Evaluation Kit which is development board that contains a series of different computing elements that can be used in a single system. The main features of the Versal VCK190 include:

- A Dual Arm Cortex-A72 - Application Processing Unit (APU)
- Dual Arm Cortex-R5F - Real-Time Processing Unit (RPU)
- Programmable Logic - The FPGA fabric
- AI Engine Array
- Programmable Network on Chip (NoC)
- DDR4 Memory controllers

The APU

The ARM Cortex-A72 is a central processing unit (CPU) implementing the ARMv8-A 64-bit instruction set. It is mostly used to run the custom made Linux operating system, which can manage the different computing elements on the board. Additionally, running bare-metal application is also possible.

Programmable Logic

Provides the FPGA elements that we discussed earlier. It is mainly used to create custom combinatorial logic and state machines that can be optimized for the specific tasks. It can be used to design special modules and specific interfaces in order to handle connectivity and inputs/outputs. Gives the possibility to effectively pre-process data before it is sent to the AI Engine or the central processor and also to post-process the data after it is received from the AI Engine by leveraging the usage of DMAs and the AXI4-Stream interface.

AI Engine

The AI Engine block is the most interesting part of the Versal architecture and the novel computing element that was introduced. It is a fundamental component of this work and it will be described in the next section.

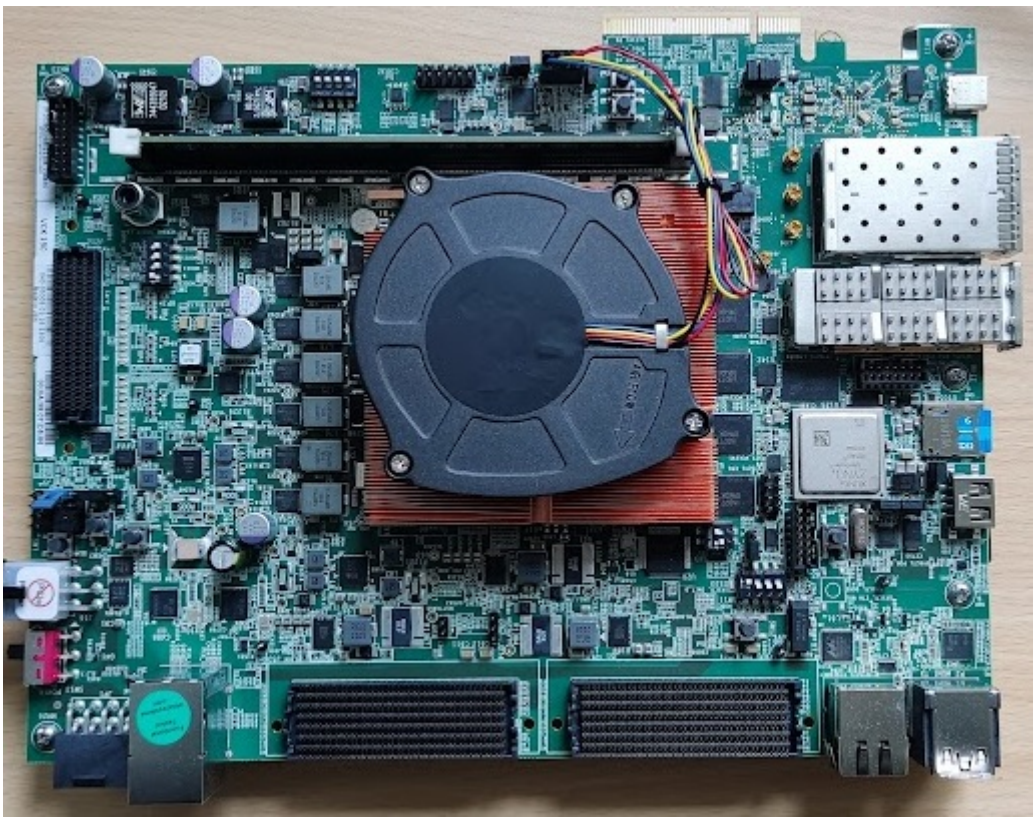


Figure 4.5: The Versal VCK190 Evaluation Kit that I used for my internship. The Versal chip is located under the fan and the heat sink but the memory is clearly visible. The board also includes every possible input connection that you might need, from HDMI to multiple Ethernet, optical and USB/JTAG connections.

NoC

The NoC is also a novel interconnect that is used to connect the different computing elements on the board. We could say it is an expansion of the idea of the connected array elements of the FPGA fabric. The NoC in a Versal adaptive SoC is a high-speed communication system that efficiently transfers data between various components like processors, memory controllers, and other integrated blocks within the chip. It employs the AXI4 protocol to manage data flow across the device, converting it into a 128-bit wide packet format that moves horizontally and vertically within the chip. The NoC ensures seamless data movement, particularly between the DDR memory controller and other parts of the device (which means that they share the same memory), with the ability to manage multiple data requests. Configuration is automated and handled during the early boot process by the platform management controller. An example of how Vivado shows the NoC connections is show in figure 4.7.

4.7 Vivado HDL Design, Applications and the Vitis IDE

4.7.1 Vivado HDL Design

The Network on Chip (NoC) is the interconnect that brings together the AI Engine block and the rest of the Versal. In order to route the data from the input ports of the board, through the Programmable Logic and the memory, to the AI Engine block and then to the output ports, we need to use the standard FPGA tool of Vivado.

Modules like the ARM Cortex-A72 cores, the AI Engine block, the DDR memory controller, the NoC and any custom or standard IP that the user may want to use, must be designed and connected together in the Vivado IDE. This is a complex process and requires a deep understanding of the FPGA architecture and the Vivado tool which is beyond the scope of this work.

The key point is that once the design is ready, the hardware designer can generate a ‘hardware

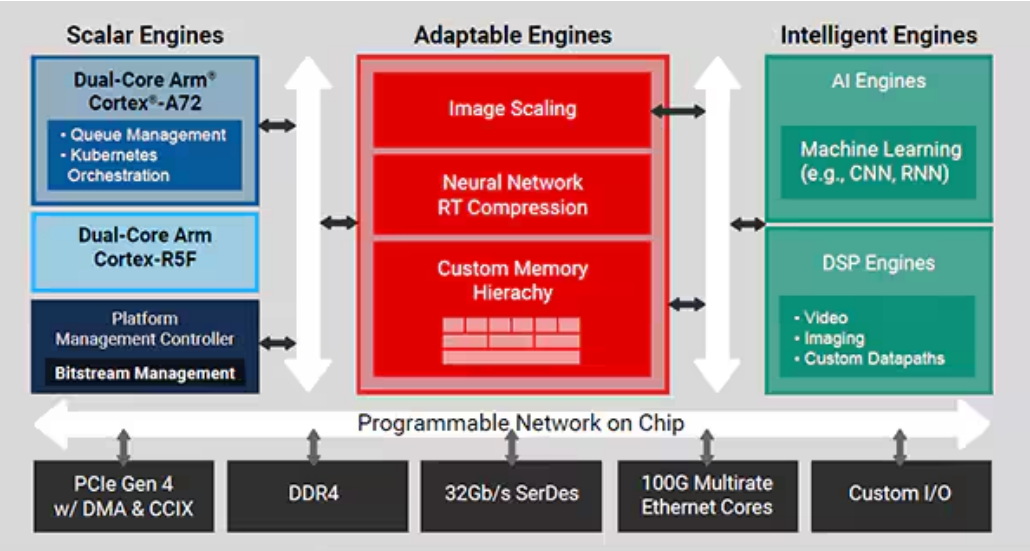


Figure 4.6: The Versal architecture [20]. The different computing elements are connected via the NoC with each other and with input/output devices.

platform', a .xsa file that gets exported by Vivado. This file describes the main memory topology and the input/outputs. Data streams are made available by leaving AXI4-Stream ports unconnected. The hardware platform file is then passed to the Vitis IDE, in order to compile the final applications and connect PL data processing blocks and the AI Engines.

4.7.2 Petalinux

Petalinux is a tool that compiles a custom Linux distribution for the ARM Cortex-A72 cores of the Versal. The user can specify which packages to include in the distribution, the root file system and many other parameters. The important part is that it takes as input the hardware platform file and the user's custom code and generates a bootable image that can be run on the Versal. This is relevant because address spaces and registers that are used to control the AI Engines and the Programmable Logic are, in a general application, accessible through the Linux kernel and thus, when compiling the custom distribution, critical information is passed from the hardware platform to the Linux kernel.

4.7.3 Creating Applications and Compiling in the Vitis IDE

AMD Xilinx provides an IDE tool in order to handle all the complexity of bringing together the ARM Cortex-A72 cores and the AI Engine block.

Among others, the Vitis IDE, gives a graphical user interface to the aiecompiler. The aiecompiler

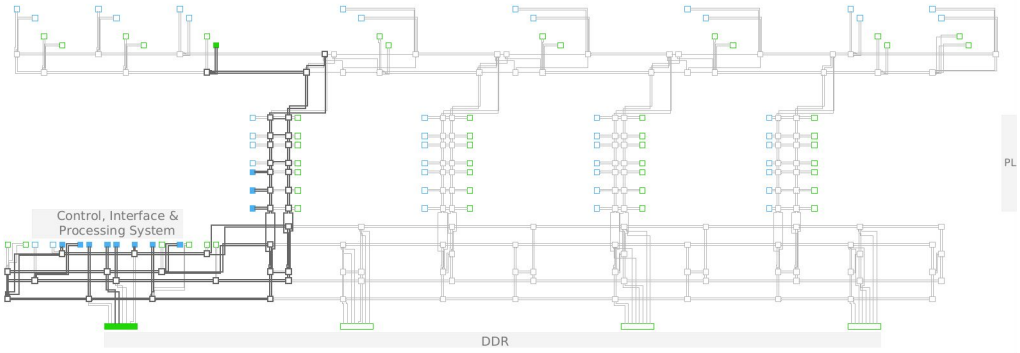


Figure 4.7: The NoC is used to connect computing elements. Here, the DDR Memory, the DMAs, a register for the scheduler and the AI Engine are connected to the CPU though the NoC routing. Picture taken from Vivado and it is the hardware platform used for the GRU implementation.

is the provided tool by AMD Xilinx to convert the C++ code written to instructions that can be executed by the AI Engine.

After having compiled the code, there is a need for a way to control the AI Engine block. That is to instruct the kernels to start executing or to stop gracefully and to be able to pass Run Time Parameters (RTPs) to the kernels through the a Linux shell command. Furthermore, the developer can create any possible control mechanism that is needed to manipulate PL modules or CPU processes through the Linux kernel. This is done by creating an application in the Vitis IDE, using C++ code.

So key note here is that the Vitis IDE is the tool that brings together the AI Engine block, custom data-processing PL blocks, and the ARM Cortex-A72 cores and allows the user to control the AI Engine block through the Linux kernel.

4.7.4 Development Flow Overview

Finally, the development flow goes as follows:

1. Develop the hardware platform in Vivado
2. Configure the addresses of the PL modules in the hardware platform
3. Export the hardware platform into the .xsa format
4. Build the petalinux distribution using the .xsa file
5. Develop the C++ drivers for the petalinux distribution using Vitis IDE (DMA Controller, Scheduler Configuration, AI Engine Controller)
6. Develop the AI Engine kernels
7. Test using the x86 Simulation
8. Test using the AI Emulation
9. Build bootable image and flash it to an SD card

4.8 The Promise

Most, out of the box computing systems are designed to be general purpose and so they contain a lot of overhead that is not needed for the specific task. This overheads are coming mostly by the necessity to fill some kind of buffer before the data processing starts, which is a limitation induced by the hardware design itself.

With the Versal VCK190 Evaluation Kit, we have the ability to more precisely customize data pipelines that can be optimized for the specific task that we need to perform. Most importantly, we can combine the different computing elements to leverage the parts that are best suited for the specific task. For example, we can use the FPGA fabric to pre-process the data before it is sent to the AI Engine, while using fast data transfer techniques to move the data between the different computing elements.

Finally, amongst all the different computing elements, the AI Engine will be the computational workhorse for machine learning applications due to the need to perform a large number of floating point operations. The matrix-vector multiplication is the most common operation but not the only one, and striking the perfect balance between distributing the workload between the different computing elements and between the different AI Engine kernels is the key to achieving the best performance.

Chapter 5

AI Engine

5.1 The AI Engine Tile

The AI Engine block was designed to carry the majority of the numerical calculations and arguably to solve a known problem in the FPGA world: floating point operations, specifically multiplications. The whole AI Engine Block of the Versal, consists of basic computational and memory units called AI Engine 'Tiles'. On the VC1902, the tiles are placed in a topology that resembles a 8x50 array to form a total of 400 tiles. This architecture ultimately affects the type of programs that we will write and the computations that can be performed, through memory and connectivity constraints.

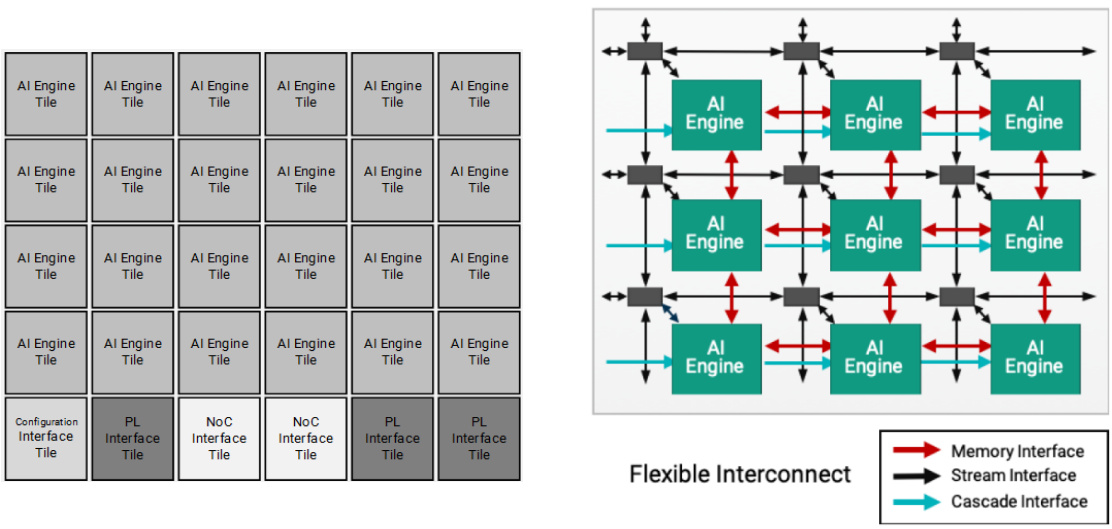


Figure 5.1: On the left a representation of how the AI Tiles are placed in an array format taken from the AMD Xilinx guide [21]. The last row of tiles provides interfacing with the rest of the device, whereas the rows that contain the AI Tiles are interconnected with each other and is used to perform intense computations. On the right, a representation of how a few AI Tiles are connected to each other with three possible ways [22].

The main hardware features of an AI Tile are:

- The scalar and vector processing units, where logic and arithmetic operations are performed
- Versatile data connections that handle connectivity to other tiles, the programmable logic and the memory
- Memory banks

Note that this is a simplified view of the AI Engine tile, and the actual hardware is much more complex. In the next sections, I will give a brief overview of the main components of the AI Engine tile and how to use them.

Vector and Accumulator Registers

A special set of registers are used to perform operations on data. This data needs to be loaded/stored from memory in vector format. These data range from 128-bit to 1024-bit wide as shown in figure 5.2. Accumulator registers are used to store the partial result of the vector integer data path. They are 384-bit wide which can be viewed as 8 vector lanes of 48-bit each. The idea is to have 32-bit integer multiplication and accumulate results to not overflow the size of the register. The table below shows how the registers are composed from smaller register units.

128-bit	256-bit	512-bit	1024-bit	
vrl0	wr0	xa	ya	N/A
vrh0				
vrl1	wr1			
vrh1				
vrl2	wr2	xb		yd (MSBs)
vrh2				
vrl3	wr3			
vrh3				
vcl0	wc0	xc	N/A	N/A
vch0				
vcl1	wc1			
vch1				
vdl0	wd0	xd	N/A	yd (LSBs)
vdh0				
vdl1	wd1			
vdh1				

384-bit	768-bit
aml0	bm0
amh0	
aml1	bm1
amh1	
aml2	bm2
amh2	
aml3	bm3
amh3	

Figure 5.2: On the left, a list of the vector registers [23], how they are composed and how many bits they are. On the right, the list of the accumulator registers [24].

The Vector Processor

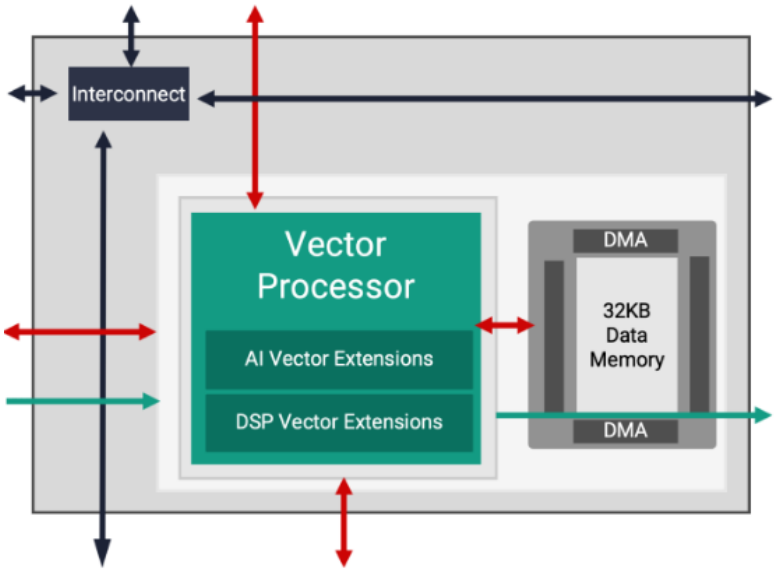


Figure 5.3: The AI Engine Vector Processor [22]. The vector processor is capable of performing arithmetic and logic operations on vectors of 8 floating point elements at the same time and has access to a local memory of 32 KB (and all neighboring tiles memory banks).

Each AI Tile comes with a 128-bit very long instruction word (VLIW) vector processor (usually referenced as AI Engine). This gives the AI Engine the ability to perform, in each clock cycle:

- two vector reads (loads)
- one vector write (store)

- one vector operation
- one scalar operation

It can run up to speeds of 1 and up to 1.3 GHz and can perform operations on vectors of size 256 bits. For 32-bit floating point operations, this means that a single AI Engine Tile can perform arithmetic and logic operations on vectors up to 8 floating point elements in a single clock cycle.

5.2 C++ Intrinsics

The way that this vectorized operation is achieved is through the use of C++ intrinsics. These are functions that directly map to an underlying hardware functionality. Specific data structures are used to represent the vectors and most intrinsics are overloaded to work with multiple data structures. In my implementation I have extensively used the 'v8float' data type, which is a vector of 8 32-bit floats. There is a moderate amount of intrinsics that allows for the manipulation of these vectors and can be found on the relevant Xilinx documentation (AI Engine Intrinsics User Guide) [25]. Specific intrinsics used for the GRU implementation will be discussed in the relevant chapter.

5.3 Graphs and Kernel Programming

When designing a program to be executed by the AI Engine the user must think in terms of computational graphs. A graph is comprised by a set nodes connected by directed edges. Each node, gets a set of data as inputs, performs some operations on the data and outputs the result. Since we are talking about computational graphs, the nodes are called 'kernels' and the edges will represent data transfers between kernels.

So from now on, when I refer to a 'kernel' I mean a sub-program that performs some operations on the data, and when I refer to a 'stream' I mean the data that is being passed from one kernel to another. This sub-program is completely defined by the developer and can be as simple as a single multiplication or as complex as an activation function. The kernel programming, also contains the mode in which data is accepted and passed to the next kernel (or to the output of the AI Engine). Also contains possible Real Time Parameters that can be passed to the kernel during runtime. So there is a lot of flexibility in the way that the kernels can be programmed and connected to each other but the main idea is that the user must think in terms of developing kernels that build graphs.

The whole AI Engine flow is contained in the top level graph. The top level graph can contain multiple sub-graphs, connected in different ways. This way the user can have a modular approach on the design of the AI Engine program, or can invoke 'tricks' such as conditional execution of sub-graphs or recurrent graphs. All these are extremely advanced topics and can only be achieved by a deep understanding of the AI Engine hardware, the AI Engine compiler and the Vitis IDE tool.

Graphs are described using C++ classes based on the ADF library provided by AMD Xilinx. The source code of graphs contains the call to the relevant kernels and how they are connected. RTPs are passed to the kernels at this level.

Kernels that constitute graphs, are written in C++ and can contain intrinsics to specifically employ the AI Engine hardware but also any operation that is useful to the kernel. When the developer writes the kernel source code then it can be used multiple times in the same graph, or any other graph.

When invoking a kernel in the graph, the user can specify a runtime ratio. By using a ratio of 1, this implies the usage of a single AI Engine tile. This is extremely important to remember, as the AI Engine tiles are limited in number, connectivity and memory.

5.4 Connectivity

The AI Engine tiles can be connected with each other in three possible ways: AXI4-Stream Connections, Cascade Stream Connections, and Windows.

5.4.1 AXI4-Stream Connections

The user has the possibility to connect the AI Engine tile with the Versal on-board memory, the Programmable logic and any other AI Engine tile (even non neighbouring) through the use of AXI4-Stream connections. FIFOs are used as data buffers in these connections. The user can constrain the size of the FIFOs and if they increase in size then dedicated DMAs are used to leverage the on-board memory (also used for non neighbouring AI Tiles). All this of course, comes with a cost in terms of latency and throughput.

Although AXI4-Stream connections are the most flexible way to connect the AI Engine tiles, they can transfer only 32-bit data per clock cycle. Another implication is that data that comes in and out the AI Engine block must pass through the AI Engine Interface block FIFOs and this can be a bottleneck in the system.

A nice feature of using the AXI4-Stream is the usage of the 'TLAST' signal. This signal is used to indicate the end of a data stream and can be used to stop the execution of the kernel. Furthermore, AXI4-Stream allows data to be transferred to one or more tiles in the AI Engine array with deterministic latency. Each AI Engine tile has two input and two output streams.

5.4.2 Cascade Stream Connections

Cascade Stream connections are a way to connect AI Engine tiles in a more direct way. The user can connect the output of one AI Engine tile to the input of another AI Engine tile by passing the contents of an accumulator register directly on the next tiles memory. and can do so passing a whole 384-bit vector in one clock cycle. This can be extremely fast and has minimal buffering, which means that the data is passed only when the next AI Engine tile is ready to accept it.

The mild downside of this connection is unidirectional between consecutive AI Engine tiles in the same row. A downside is that there is no inbuilt way to signal the end of the data stream thus the user must implement a way to stop the execution of the kernel.

In this implementation a special vector is used to signal the end of the data stream. This vector is filled with zeros and is passed to the next AI Engine tile and in order to signal the end of the stream it flips the first element of the vector to 1. This way the next AI Engine tile can detect the end of the stream and stop the execution of the kernel. This needs to be done in every data transfer thus taking at least one clock cycle.

5.4.3 Windows

Windows are probably the best way to connect neighbouring AI Engine tiles. They can be thought of as blocks of data that are passed from one tile to another by using shared memory blocks and the consumer and producer must be correctly synchronized. Windows provide an alternative to streams for data handling in AI Engines. Unlike streams, where data is processed immediately upon arrival, data transferred via windows is first written to the AI Engine's local data memory. There are two types of windows: synchronous and asynchronous. Synchronous windows require the entire window to be written before the kernel can access the data, unlike streams where computation can start as data arrives. Asynchronous windows, on the other hand, allow the AI Engine to perform other tasks while waiting for data to arrive, which can be advantageous in certain scenarios. Synchronous windows, which must be at least 16 bytes in size, with a recommendation that the total size of concurrent windows be less than 50 % of the available local memory to enable double buffering. Double buffering allows the next set of data to be loaded while the previous one is still being processed, thereby

increasing throughput. However, both buffers must fit into one AI Engine's memory, limiting the maximum window size for double buffering to 16 KB.

In this work, I didn't use windows as they induce more complexity to the design thus I will not discuss them further.

5.4.4 Kernel Connectivity

This is where the limitations of the AI Engine hardware come into play. The AI Engine tiles are limited in the number of connections that they can have. Namely, an AI Engine tile, and therefore a kernel, can have only two stream connections as input and two stream connections as output. Furthermore, although a tile can use all connections as AXI4-Streams, it can have only one input and one output connection of cascade stream type.

This means that the designer should have a clear view on when to use AXI4-Stream connections and when to use cascade stream connections.

5.5 Memory Banks

Each AI Engine features 16 KB of program memory, capable of storing 1024 instructions, each up to 128 bits wide.

Each AI Engine tile contains eight data memory banks, with each bank consisting of 256 words of 128-bit single-port memory, totaling 32 KB per tile. An AI Engine can access its own data memory and three neighboring tile memories, providing a total of 128 KB of accessible memory. The stack, which is a subset of this data memory, and the heap size can be customized through compiler settings or source code constraints. The 128 KB memory can be logically viewed as either a single 128 KB block or four 32 KB blocks, with each block further divided into odd and even banks. On the periphery of the AI Engine array, engines have fewer neighboring tiles and thus reduced memory access. Each memory port supports operations in 256-bit/128-bit vector mode or 32-bit/16-bit/8-bit scalar mode, with the 256-bit mode formed by pairing even and odd memory banks. Additionally, read-modify-write instructions handle 8-bit and 16-bit stores, and all three ports can operate concurrently, provided they access different memory banks.

5.6 Pipelining

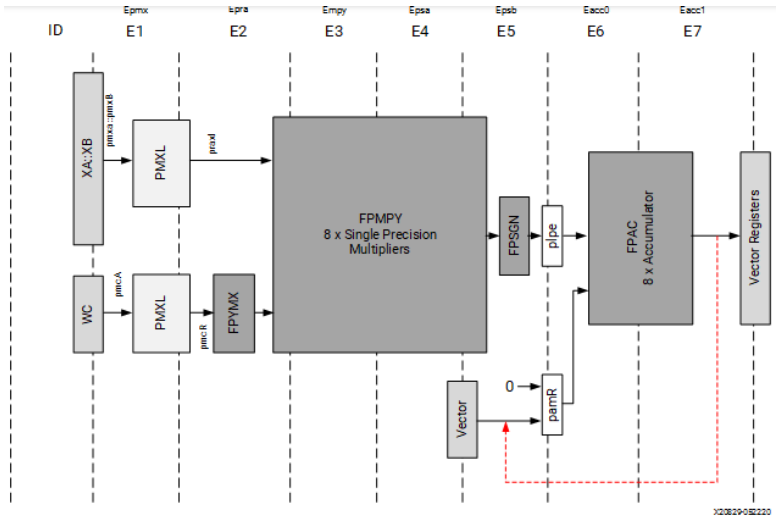


Figure 5.4: The AI Engine Vector Processor pipeline [26]. The vector processor can operate on 8 elements at the same time but the operation can take up to 8 clock cycles to complete. Nevertheless, we don't have to wait for the operation completion to start the next one. The moment data are moved from one stage to the next, new data can enter the pipeline.

Despite that the vector unit can operate on 8 elements at the same time, any operation can take multiple clock cycles to complete. The true power of the hardware comes from the fact that we can schedule operations before the previous one is completed. This is called "pipelining". This way the vector unit can operate at full speed, i.e. process values on each clock cycle, and the user can achieve maximum throughput.

This now becomes an interplay between the user and the AI Engine compiler. The developer must write C++ code in such a way that the compiler can understand which operations can be pipelined and which cannot. The compiler will try to schedule the operations in such a way that the vector unit is always busy and the user can achieve maximum throughput but this can only happen if there are no dependencies between the operations, or at least, create the least possible dependencies.

The simplest example of writing code that can be pipelined by the compiler is the ability to unroll loops. For instance, writing a for loop that consists of a number of operations may be slower than writing each operation in a for loop. With the latter approach, the compiler can see which variables are dependent in each cycle and can schedule them accordingly.

Another way that the user can inform the compiler about the intent of the code is through the use of pragmas. Pragmas directly affect the assumptions made by the compiler and should be used with extreme caution. It can be a powerful tool in the hands of an experienced developer.

5.7 Parallelism

Every AI Engine tile compute independently from the others. This means that the user can write a program that can be executed in parallel using multiple AI Engine tiles. This can be achieved by inputting data in multiple kernels. Execution dependence is introduced only by the connections between the kernels. This can cause serious problems or leveraged into conditional execution of kernels. It really depends on the user's intent and skill. In this implementation, a lot of the computations depend on the stall of the data stream, and execution of the kernels is mostly sequential. Creative ways to use parallelism is an open field for exploration.

Chapter 6

The Existing Implementation

6.1 The Experiment

Our goal is to cancel out the microbunching instability for that specific bunch by reading the coherent synchrotron radiation fluctuations read by KAPTURE-2. In general, we would like to use a modulation of the RF cavity voltage, specifically tailored for one bunch that circulates the storage ring.

On the machine there has to be a perfect synchronization between different parts. The hardware that will perform inference on the observation of the emitted radiation has to be very close to the accelerator. Observations need to be transferred to the inference hardware in real time and this is done with a purposely installed fiber optic cable. At the same time, every couple of seconds, data accumulated on the inference hardware has to be transferred to a computer on the control room to perform training of the neural network. Similar work done in IPE just like [27] and [28] was considered.

6.2 Timing

The optimal timing would be an observation and an action on the RF system on every single pass of the bunch so that the RF system can (attempt to) correct the fluctuations in real time.

Concerning the KAPTURE-2 system, it has the ability to produce a set of data points of the emitted radiation in every single pass. This means that the frequency of the data acquisition is equivalent to the revolution frequency of the bunches. The revolution frequency is ~ 2.72 MHz.

The RF system has a frequency of $\sim 499.7MHz$ on the main storage ring.

The design harmonic number is $h \sim \frac{500}{2.72} \sim 184$. This means that the RF system can attempt a correction to 184 bunches in a revolution, but since we are targeting only one bunch, at a first glance it seems that there is enough time to perform the correction.

But the reality is that there is latency both in all the data transfers and in the processing of the data on the Versal board. Furthermore, the RF system should be able to keep all the other bunches stable while it is attempting to correct the fluctuations of the specific bunch. This means that we would want to modulate a small time window of the RF system to correct the fluctuations of the specific bunch while the rest of the bunches should be unmodulated. Due to the bandwidth of the RF system, we can apply a modulation every 6 revolutions and thus $\frac{f_rev}{6} = \frac{2.72MHz}{6} = 453,333.33Hz \sim 2.2059 \times 10^{-6}s$, so we can only operate on a single bunch. This is nonetheless sufficient for a preliminary study.

An important point is that, since the inference hardware is a pipeline of data which is fixed in runtime, the ratio in which data exits the pipeline and data enters the pipeline should match the frequency that we can act on the RF cavity. This means that the design of the data pipeline should be such that the pipeline never stalls to wait for data to exit the pipeline (as modulation actions), while at the same time never stalls to wait for data to enter the pipeline (as samples from KAPTURE-2).

This is controlled by the decimation and interpolation factors of relevant kernels: the decimation kernel, which actively downsamples the data, and the interpolation kernel which can upsample the data. The decimation and interpolation factors are given as Run Time Parameters (RTPs) to the kernels. Both should have a collective effect that matches the minimum timing in which we can act on the RF system.

For instance, since the best we can do is have 1 action every 6 revolutions, the decimation kernel should downsample the data by at least a factor of 6, and any combination of factors that respects this ratio is acceptable.

6.3 Existing Implementation on the AI Engine

Although the AI Engines will be presented in a more detailed way in the next chapter, I will give a brief overview of the existing implementation of the data processing pipeline. The inputs are observations of the emitted radiation from the bunches. The outputs concern the modulation (amplitude and phase) of the RF cavity. In this section I will discuss some important design choices on the processing of the data and will ignore how the data enters and outputs the hardware which invokes the usage of IP blocks and the AMD Xilinx Vivado tool, thus, the following is a high level overview of the data processing pipeline that is implemented on the AI Engine. This means that each of the following steps is implemented as a separate kernel.

Downsampling

The KAPTURE-2 system will provide observations of the emitted radiation and thus a filtering and downsampling of the data is necessary. The filtering is done by employing a Finite Impulse Response (FIR) filter, which is a technique of signal processing that performs a sliding dot product of fixed coefficients b in the acquired signal:

$$y[n] = \sum_{i=0}^{N} b_i x[n-i]$$

Here, $y[n]$ is the filter output, $x[n]$ is the filter input signal and b_i are the filter coefficients. N is called the "order" of the FIR.

What we are targeting, is to keep the dynamics that are relatively slow and filter out the fast fluctuations. The motivations for this is that the RF system is limited on how fast modulations it can perform and thus trying to correct fast fluctuations is not possible. The FIR coefficients were calculated using a python library, using a specific frequency threshold. Thus we are hoping to filter out the fast dynamics and then approximate the intensity curve with only a few sample points. The downsampling is done by removing filtered data samples by the given decimation coefficient. The same process can be employed more than once, into a couple sequential kernels.

Observations Buffering

Once the input data is filtered and downsampled, it is stored in a buffer. The buffer operates as a First In First Out queue and it is used to store the last 64 observations. A FIFO will push in new data and push out the oldest data. It is also interesting to notice that: if we image all possible generated data that will enter the buffer as a static time series (like a dataset), the buffer then acts as a sliding window.

The data stored in the buffer will be used as an input vector to the next kernel.

Feed Forward Neural Network

This is where observations are transformed to actions. The buffered processed data is fed to a feed forward neural network. The neural network is comprised of three layers: the input layer that is of the same size of the buffer, a hidden layer and an output layer of two neurons. The output layer is the modulation of the RF cavity.

The neural network is trained offline using a commercial CPU on the control room and then the weights are transferred to the AI Engine.

Gaussian Noise

Due to the nature of the Reinforcement Learning algorithm that is used, in order for the model to explore possible better solutions, Gaussian noise is added to the output of the neural network. The noise is being generated on the Programmable Logic (PL) side, using a custom implementation.

6.3.1 Storing Inputs and Outputs to Train

In order to be able to train the neural network, the inputs and outputs of the neural network are stored in the memory of the hardware device. This is also the main factor of the frequency that is required to transfer the data to the control room and perform training. The memory available on the Versal board is in general 8GB but only a subset of it, is reserved from the Operating System and can be used to store the data. By diving the size of available memory with the size of the data that are being produced during inference we can calculate how often we need to transfer the data to the control room.

6.3.2 Performing Offline Training

As discussed in the Reinforcement Learning chapter, the neural network is trained offline using a commercial CPU using PyTorch and the standard stablebaselines3 library. The value and policy networks are trained using the Proximal Policy Optimization (PPO) algorithm. When the training is done, the weights of the policy network are transferred to the AI Engine and the next set of inference episodes can begin.

6.4 Feedback Modification

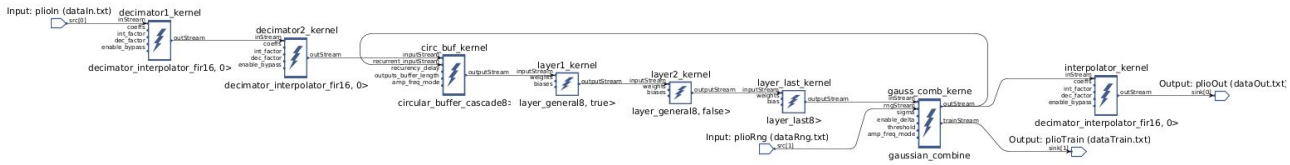


Figure 6.1: The feedback modification on the existing implementation. The feedback is the output of the neural network action predictions plus the gaussian noise added after the 'gaussian combiner' kernel. The reader can see the connection running from the 'gaussian combiner' kernel to the 'circular buffer' kernel.

As a preliminary step to designing the Gated Recurrent Unit (GRU), an effort was made to modify the existing implementation to include a feedback of the action outputs into the neural network. The motivation behind this is to provide context to the neural network as to the history of the actions that were taken. This gets more important as we are targeting slower dynamics that may persist over multiple revolutions of the bunch.

The way I implemented this was by copying the functionality of the circular buffer into an additional buffer that stores the last N actions that were taken. Once I store the last N actions, the actions are concatenated to the observations in a single input vector and feed it to the neural network.

The feedback is the output of the neural network action predictions plus the gaussian noise and was implemented as a second write from the last kernel and a read to the circular buffer kernel as shown in 6.1. The design choice of using the same output port to send data to two different kernels, impacts greatly the performance of the system. First, because as long as the data is not accepted by both receiving kernels, further program execution is halted and second, because it now potentially forces us to connect two, non-neighbouring AI Engine tiles, which means we will have to use AXI4-Stream connections.

An additional drawback is that when the observations buffer runs for the first time, it will try to read the last action that was taken but obviously the data pipeline has not yet produced any action. This is solved by introducing an if statement with a flag that checks if the circular buffer runs for the first time and if so, it will delay the read of the action buffer for at least one cycle.

The feedback modification was deployed live on the KARA machine and unfortunately it did not provide any improvement on the performance of the system, but it was a good exercise to understand the data pipeline and the AI Engine and how to design a system that requires a feedback.

Chapter 7

Benchmarking the AI Engine Hardware

7.1 Motivation

There are several reasons for which a developer would like to have a tool that can run tests, in real time, on the physical AI Engine hardware. One of the main reasons is to be absolutely sure that the AI Engine is running correctly and as intended by the developer. Using the simulation tools provided by Xilinx is a good way to test the AI Engine, but it is not the same as running the AI Engine on the physical hardware. The simulation tools are not perfect and can suffer bugs and limitations that are not present in the physical hardware. This may be less relevant in some other applications but it is extremely important in big physics experiments where deployment time of the test may be limited, expensive and planned months in advance, only to find out that something is not running correctly on the last moment.

A second reason is to be able to test applications that run outside the AI Engine but work cooperatively with it. In this implementation, I will be using Direct Memory Access (DMA) to transfer data between the AI Engine and the Processing System, use an application to pass Run Time Parameters (the weights of the Neural Network, for example) from memory to the AI Engine, and start and stop the AI Engine. These applications are not part of the AI Engine but are crucial to it's operation. This also implies the possibility to measuring latency and throughput of both the AI Engine and the Programmable Logic (PL) block that comprise the whole system.

A very important third reason is that using the simulation tools the user cannot control the timing of the data that is being fed to the AI Engine block, and the input operates always at maximum speed. This is extremely important in most of the real word applications in which data are not always available at the same time. In our case, for example, data from KAPTURE-2 are coming at bursts when the bunches pass the detector.

For these reasons, I have developed a benchmarking tool that can run on the AI Engine hardware and can be controlled by the user to test the AI Engine and the whole system. It also served as an introduction to the AI Engine hardware and the Vitis IDE tool, and to concepts of memory management, data transfers, writing in Verilog and C++, and others.

7.2 The Simulation 'Arsenal'

Before we delve into the details of the benchmarking tool, it can be beneficial to briefly introduce the available tools that can be used to simulate and test the kernels running on the AI Engine hardware.

7.2.1 x86 Simulation

The x86 functional simulation is a software simulation of the AI Engine hardware, the computational graph and of all the kernels code are executed on the host machine. This is the fastest simulation

and thus makes it ideal to debug the C++ code of the kernels. The most common usage of the x86 simulation is to validate the numerical operations of the kernels and detect deadlocks on the flow of the data. It can also provide some tracing information about the execution of the kernels, but it is not easy to read and interpret. Maybe the easiest way to use the x86 simulation effectively is to use it in conjunction with strategically placed 'printf' statements in the code to the debug the kernels. This can be cumbersome, especial when using v8floats as data types.

The downside of the x86 simulation is that it is not timing accurate and thus cannot measure latencies and the routing delays and FIFO lengths are not modeled in the x86 simulation. x86 simulations are composed of multiple threads sharing the x86 processor and so memory access conflicts, and stream access conflicts are not modeled. When targeting the x86 simulator, the AI Engine compiler translates all vector processor specific instructions into a series of x86 instructions. Therefore, the x86 simulator will not estimate any design throughput. Furthermore, software pipelining is not modeled in the x86 simulator.

Because memory access is not modeled in the x86 simulation, illegal access during the execution of the kernels will not be detected. This will lead to segmentation faults when running the simulation, but the user can use a debugger to find the source of the error. During the implementation of the GRU kernel, I have encountered this problem and had to invoke the use of 'gdb' to find the source of the error.

7.2.2 AI Engine Emulation

The complete emulation of the AI Engine should be considered as the most accurate representation of a hardware run. Hardware emulation combines SystemC and RTL co-simulation to balance simulation speed and accuracy, though it doesn't fully replicate hardware behavior. This means that you might still notice differences in performance or functionality when comparing emulation to actual hardware. It's a valuable tool for debugging hardware, especially for interactions between components like the AI Engine and memory.

Hardware emulation is significantly slower than real hardware, so it's recommended to use smaller data sets during testing. Developers can use the Vitis hardware emulation flow to simulate complete systems, including AI Engine and PL logic. The flow uses a mix of functional and cycle-accurate models, but sacrifices accuracy for speed in some cases, particularly in modeling communication and latency. It should consist the last round of testing just before deployment

For advanced users, emulation can provide detailed insights with waveform viewers like the Vitis Analyzer.

7.2.3 The Vitis Analyzer

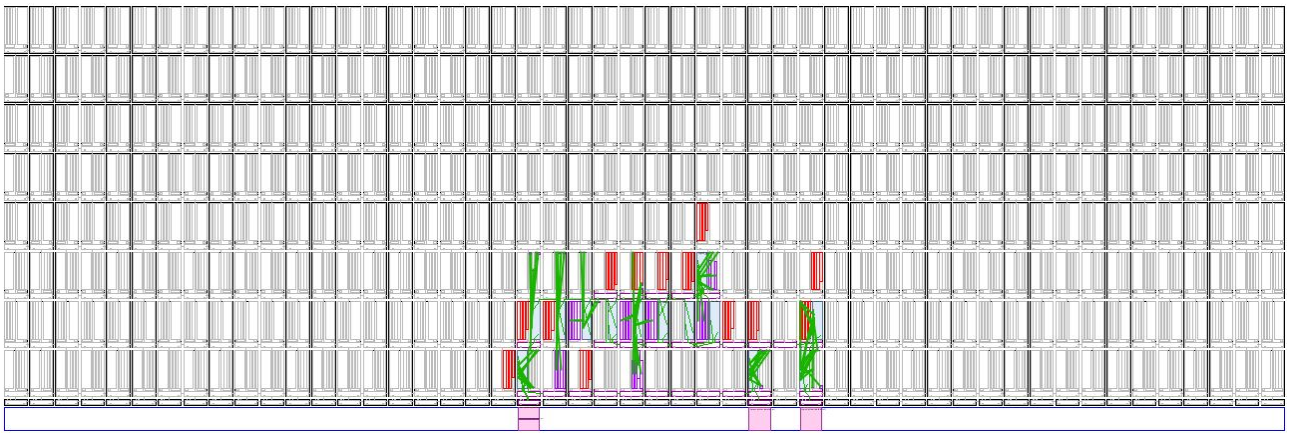


Figure 7.1: One of the features of the Vitis Analyzer is the graphical representation of the exact transfer of data between tiles and memory banks. Many other information are available.

The Vitis Analyzer is a dedicated tool provided by AMD Xilinx in order to help the user analyze the performance of the AI Engine kernels. One of the features of the tool is the graphical representation of the AI Engine Tiles inside the AI Engine Block as show in figure 7.1. In this representation, the allocated stack memory and the exact flow of data between tiles can be visually inspected and interact with. All connections between the tiles, with specific FIFO depths are presented as information to the user.

The tool also compiles a visual representation of the graphs, subgraphs and the kernels can be extremely useful to debug the data flow.

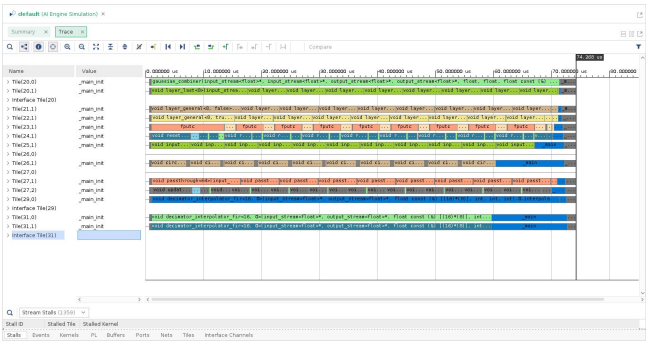


Figure 7.2: Tracing events during the execution of the AI Engine. This function is available for Hardware Emulations.

A key feature is the ability to trace the execution of events at the level of kernel abstraction. This particular mode, allows the user to see exactly the inputs and outputs of the kernels and to measure the time it takes from one event to the other. Even more important, is the ability to detect stalls in the data flow and to specifically identify the source of the stall. Although all this information is relevant and available, this specific mode of the tool maybe be sometimes difficult to use and interpret, because of the sheer amount and format of the information that is presented to the user. The format looks a lot like the traditional simulation traces used in FPGA tools, but instead of waveforms, the user is presented with a lot of text, numbers and blocks that represent the kernels and the data flow. Nevertheless, if someone has the patience and time to identify the source of the problematic behavior it is possible to use the Vitis Analyzer to debug the kernels.

7.3 Custom Hardware Benchmark

The benchmarking tool is composed of two main parts. The programable logic design shown in figure 7.3 which contains two custom logic modules, the ‘timestamper’ and the ‘scheduler’, and the necessary connections to the AI Engines. These can be a simple input and output connections, or complimentary to the design like random number generators and different data routings.

The second part is a python script that handles memory management, simulated data generation or loading, the execution of the applications that control the PL design modules and the AI Engine. The python script also handles metadata generation and collection and the generation of the final file export.

7.3.1 Reserving Memory for Tests

The first step is to make sure that enough memory exists to generate or load data that will be used as input to the test and then to store the output of the test. The Versal has a DDR4 memory controller that can be used to allocate memory used by the DMAs handling data transfer of the tests but first we have to make sure that we have access to a contiguous block of memory. We must know the physical address of this block of memory and make sure that it is not used by any other process and specifically by the Operating System.

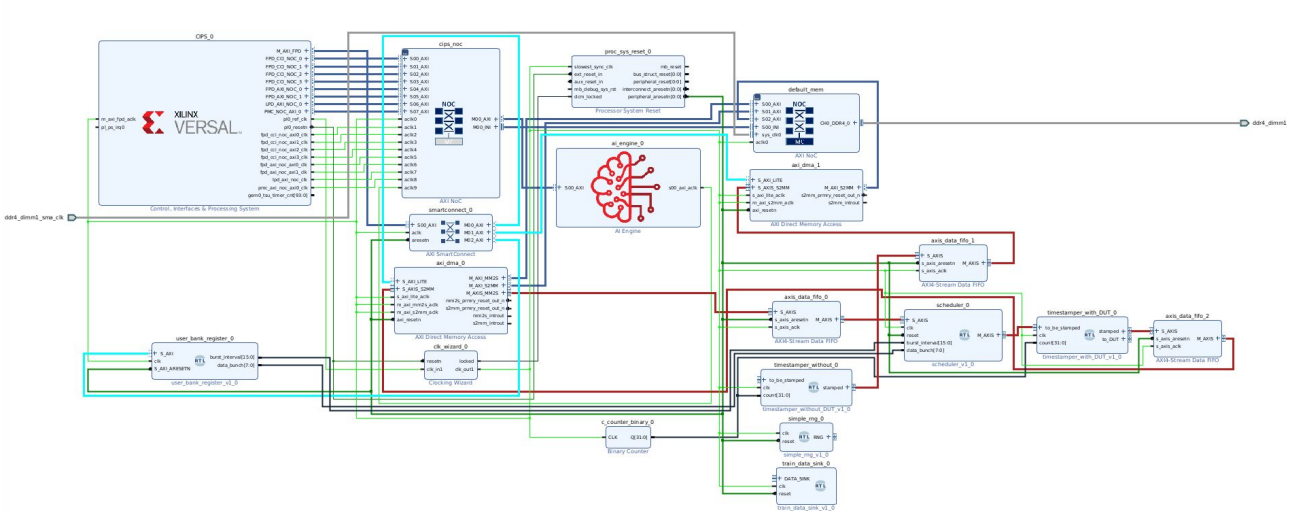


Figure 7.3: The actual block design of the benchmarking system in Vivado. The reader can see that the main components like the CPU (CIPS block), the AI Engine (unconnected block), the NoC and the AI Engine exist as IP blocks that need to be connected with AXI4 interfaces. On the lower right side of the block design, a set of DMA → FIFO → Scheduler → Timestamper modules before and after the AI Engine. The AI Engine remains unconnected on Vivado design and Vitis IDE will connect the AI Engine with the PL modules.

In order to do that, we specify an option in the kernel device tree file that instructs linux to not use that memory region. Finally, 4GB of memory is reserved for the tests, and by using the 'mmap' function, the memory is mapped to the user space.

Memory addresses for the Programmable logic modules should also be specified in the hardware platform during the development in Vivado and then passed to be built in the petalinux distribution.

7.3.2 Custom PL Modules

In order for the benchmarking tool to work, the Programmable Logic (PL) needs to contain the necessary connections to the rest of the platform. Two of these connections are going to be absolutely essential for any design and are required to pass data from the PL side to the AI Engine and vice versa: the AI Engine data input and data output. One more crucial wiring, must be connecting the memory with the system but this could be considered standard. On top of these connections, the hardware platform will have to contain any additional custom connections to the AI Engine. In other words, we need to contain and simulate the behavior of every connection that would also work during the actual implementation.

For example, in our case, we must have a connection for the random number generator as input to the AI Engine and a connection for the data sink that accumulates training data from the output of the AI Engine. So any implementation specific connections must be added to the hardware platform.

Except the connections to the AI Engine, the crucial components are a set of DMA controllers and FIFOs. DMAs are used to transfer data between the memory and the PL modules and FIFOs are used to regulate the flow of data.

All PL modules use AXI4-Stream protocol to transfer data, which means that they are operating at most at the speed of one float per clock cycle. The clock frequency is 250MHz.

The Timestamper

The timestamper is a custom logic module, that is used to create a copy of the data that passes through the module and concatenate a timestamp. After the data has been timestamped it is stored to memory. For the purposes of the benchmarking tool, the timestamper module needs to be implemented both before the input and after the output of the AI Engine.

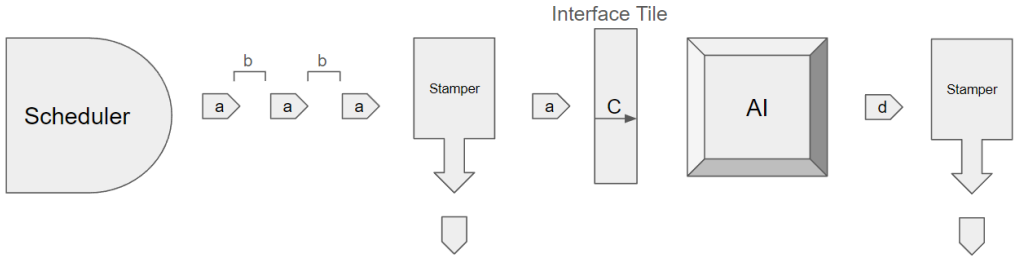


Figure 7.4: A graphical representation of the PL design. The scheduler sends data of length a with a delay of b clock cycles. Then the timestamper module timestamps the data and sends a copy to the memory. Before the data enter the AI Engine they have to go through the interface tile, which can only accept data at a rate of c clock cycles. Finally, a copy of the outputs of the AI Engine are timestamped and sent to the memory. All data transfers from and to memory are handled by the DMA controllers.

Two separate DMAs are used to transfer data. The first DMA transfers the test input data that are stored in memory and are going to be used as inputs to the AI Engine. This DMA serves a double purpose, as it produces the input of the first timestamper and stores the timestamped copies in memory. The second DMA transfers data from the output of the second timestamper to a specific location in memory. By knowing the exact memory addresses where the DMAs stored the timestamped data, the user can then read the data and calculate the latency and throughput of the AI Engine.

The timestamps are generated by a binary counter module outside the timestamper module and uses 32 bits to represent the time. The clock module is a simple counter that increments every clock cycle and is reset by the user thus allowing to count up to 21 seconds which are more than enough for our purposes.

The tricky part of the logic is to handle correctly the AXI4-Stream protocol. This is because the AXI4-Stream handshake signals should propagate correctly through the timestamper module and the in-between FIFOs

Also the DMAs are using the AXI4 Stream protocol to transfer data and the TLAST signal plays a crucial role in the correct stopping of the DMAs.

The Scheduler

The scheduler is a more complex module that is used to control the timing of the data inputs into the AI Engine. The scheduler is placed right before the first timestamper module and will create burst of data of specific lengths. This is supposed to simulate how data would be produced from the KAPTURE system in which data are produced at each revolution of the bunch.

The inner workings of the scheduler are not so simple and are based on a state machine of two states, an idle state and a burst state. The tricky part, is to count the correct number of clock cycles for the burst state because we need to take into account cases where the AI Engine is not ready to accept data but the scheduler is in the burst state. This means that a valid count is tied with the state of the AI Engine and this is not trivial to implement. For example, there is a internal delay of two clock cycles in the scheduler that is impossible to remove, due to state change, and this has to be taken into account when calculating the burst length. At the same time, all AXI4-Stream signals should also be propagated correctly through the scheduler.

The scheduler accepts as input a burst length and a burst rate. A single register is used and pass the parameters to the scheduler. In order to pass the parameters, a C++ program was written that uses the mmap function to write to the register through the Linux kernel.

7.3.3 Top Level Test Manager in Python

Python as a scripting language, is ideal to control the execution of the tests. The initial idea was to use a modular approach and to develop a class for each of the three main components of the test:

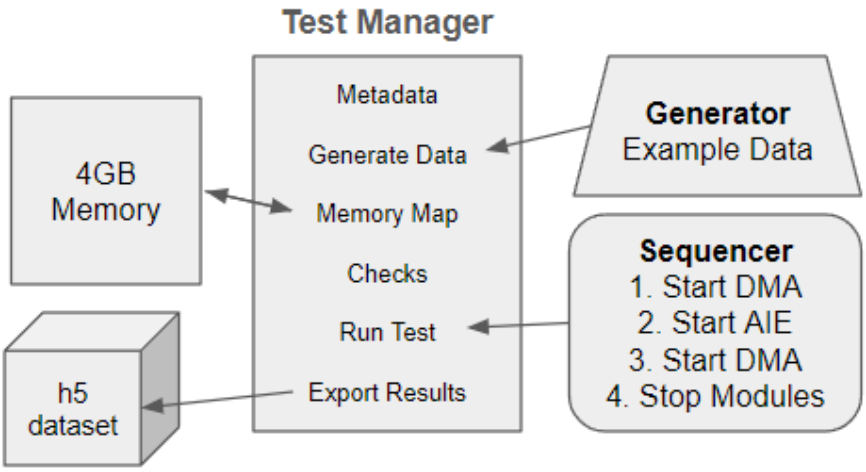


Figure 7.5: A representation of the python script.

the generator, the sequencer and the test manager. Instead, due to time pressure, the test manager contains the sequence of the test. The test manager is the script that controls the execution of the test as a whole and a separate class was used in case the user wants to generate specific data. So the script is somehow modular but not in the way that was initially intended but nevertheless works and it is only mentioned as a possible future work to be completed.

Initialization

The entry point of the script is to initialize the mmap object that will be used throughput the script. The init function also initializes a dictionary that will be used to store metadata about the test. Metadata is considered: the name of the test, the method used to generate input data, time and date of the test and every other information that can be useful to the user like the printouts of the C++ applications that control the PL modules.

Writing and reading to the mmap object is done through the 'mmap' module of python and is fairly simple to use. The complexity lies on the correct usage of the page size and the length of data that can be written or read at once.

Generator / Loader

The next step is to load on memory the data that will be input to the AI Engine test. The user can choose between two methods: to generate data or to load data from a file. Both choices will use numpy arrays.

In order to load data from a file, the user must provide a file that contains the data in a specific format.

In order to generate data, the user must provide a distinct generator object that needs to contain specific methods. In the generator class the user can define exactly how the data will be generated and it just needs to be passed as an instance to the main script. Leveraging the ease of numpy, I developed two generators. A random number generator and a serial generator which generates numbers in a range. The serial number generator is useful to validate data, since is predictable and easy to reproduce in a separate setting.

Once the data are available, the script will use the mmap object that initialized in the beginning to write the data to the memory.

Execution of the Test

A test proceeds as follows. The script will use the subprocess module to start the C++ applications that control the PL modules and the AI Engine in the reverse order of the design, i.e. initializing the last block first so that data is not produced while the following blocks are uninitialized.

1. Write the relevant parameters to the scheduler module, i.e. burst size and data transfer period. If 0 and 0 are passed as parameters, the scheduler will use free flow of data and will not control the timing.
2. Start the DMA controller that will transfer data from the outputs of the AI Engine and the timestamper to memory. The moment the DMA controller starts, it is ready to accept data and pass data. The user doesn't have to do anything else.
3. Start the AI Engine controller application. The AI Engine starts and is ready to accept data and start processing.
4. Start the DMA controller application that will transfer data from memory to the input of timestamper and then the AI Engine. The moment the DMA controller starts, data will start streaming to the AI Engine. At this point the user waits for the test to finish.

If the AIE code was properly implemented, the TLAST will stop the DMAs and all the kernels in the AI Engine. Usually, this is the phase that the user will have to debug the most. The script also collects the printouts of the DMAs and the AI Engine controller and stores them in the metadata dictionary. The DMA controllers need as input the exact memory addresses to read and write data and this is automatically handled by the script.

Arguably, this is the most important part of the test. If it works, the user can be sure that the applications and the hardware are working correctly.

Exporting the Results

The final step is to export the results of the test. Since we have two sets of data, timestamped inputs and timestamped outputs, and a set of metadata relevant to the test, a very convenient way to store all this information is to use the h5py module, to store the data in a single HDF5 formatted file. This file format, uses a tree structure to store data and allows to couple data with metadata.

7.3.4 Analyzing the Results

There are two types of analysis that can be done on the results of the test. The first is to validate the numerical operations of the AI Engine and the second is to measure the latency and throughput of the AI Engine.

In this work, latency is defined as the amount of time it takes for a computational pipeline to produce a result. To start measuring latency we also need to establish a convention on when it is appropriate to start measuring time. Therefore, it seems natural to choose as a starting point the time the first input is accepted. So to recap, the difference between the time of the first input of a potentially series of inputs and the time of the affiliated output, constitutes a latency measurement.

Throughput is defined as the rate that the AI Engine will accept data during a fixed period of time. Since FIFOs are located at the interconnect interfaces, we should be extremely careful when we are measuring throughput to not mistaken caching effects as speed.

7.4 Dot Product Test

In order to verify the functionality of the system, a dot product between 64 inputs and 64 weights at a time was used as an initial test design. Care must be taken in the choice of the number of data points to be processed. If the AI Engine design is programmed so that it has deterministic timing

performance, one just needs enough data to probe FIFO saturation effects. This can be performed by running the interface at maximum throughput. As such, the test is repeated 10 times for a total of 640 inputs.

During some preliminary testing of the benchmarking tool I wanted to run a normal amount of test runs, at maximum back pressure (so no scheduler, just maximum flow). By running this simple test I can measure max throughput of the interface tile.

The results of the test are shown in the following figures.

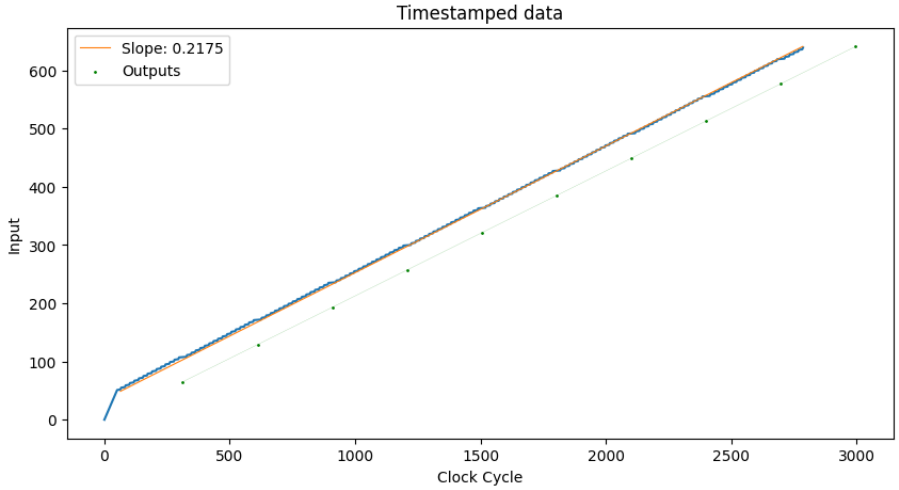


Figure 7.6: A plot with all the input and output samples. By fitting a slope on the region after FIFO saturation we can measure throughput. Thus with maximum backpressure the AI Engine can accept

First, it is useful to visualize how is the AI Engine block accepting data from the PL regime. A single AXI4-Stream connection, feeds input data to the AI Engine at the frequency of the PL clock. The data corresponds to one 32-bit float per clock cycle, with a frequency clock of 250MHz or 4 nano seconds. Thus, I can plot the inputs timestamps, normalized by globally subtracting the initial timestamp. The result is the slope seen in figure 7.6 and corresponds to the throughput of the system, namely how many Bytes can be fitted into the AI Engine per second. Since I have acquired the slope in arbitrary units all I have to do is convert is to B/s like this:

$$0.2175 \times \frac{4\ [\text{Bytes}]}{4 \times 10^{-9}\ [s]} \sim 217,5\ \text{MB}/s$$

Zooming in the first inputs in figure 7.7 we can recognise the effect of a FIFO which accepts data 'instantly', at a rate of one clock cycle, until it is full. Note that the axis are inverted.

With FIFO saturation the true data acceptance rate is revealed. It is also measured by the slope in figure 7.6 but if we just zoom in in a region after saturation (figure 7.8 we can see that the AI Engine accepts four inputs and then comes to a halt.

A very interesting result shown in figure 7.9. The AI Engines will output the results in pairs of two. The AI Engine waits until the second result is ready before outputting both results. I also ran a test with a dummy output of a single float as 0, and the AI Engine will output the results in pairs of two, even if no computation is done for the second one. This behavior is due to the fact that the AI Engine waits to store both values in the output tile and only they come as outputs to the PL fabric.

Finally, to measure latency we need to already know how many data it takes to complete a calculation and simply calculate the difference of the first input timestamp to the relevant output timestamp. In this simple example, we know that it takes 64 inputs to start the calculation of a dot product and since the operation of the AI Engines is deterministic it is easy to relate the outputs to inputs. The figure 7.10 below shows the latency acquired in this mode:

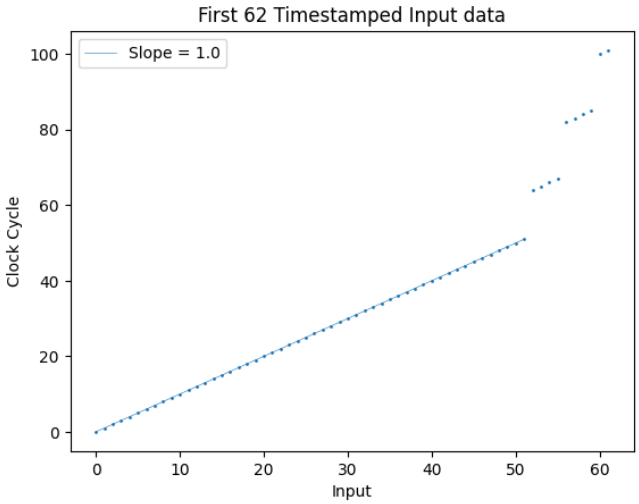


Figure 7.7: A zoom in on the first 62 inputs. It is clearly shown the effect of the FIFO, which accepts data at every clock cycle and then saturates.

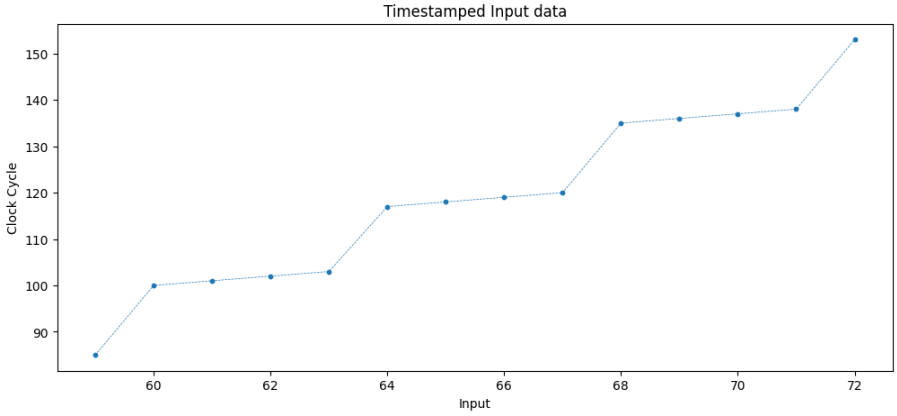


Figure 7.8: A zoom in on the inputs when the FIFO is saturated. The fit will show the rate but in the zoom in it is clear that 4 floats are accepted every 18 to 19 clock cycles.

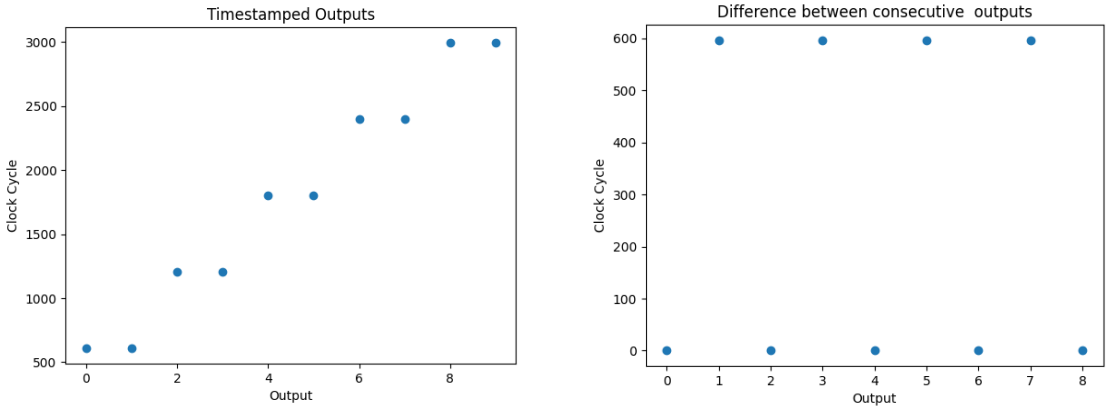


Figure 7.9: A timings plot of the dot product outputs as they come in pairs. On the left, a better view of the effect as the difference between consecutive outputs is calculated.

The first latency measurement in figure 7.10 is lower because the data is processed as soon as it is available to the AI Engine and I consider this an outlier when calculating the average latency. All following data will be first accepted by the FIFO and so will wait there before it is processed by the AI Engine. This has the effect of increasing the effective latency. It is worth noticing how the data rate, FIFO depth, and latency are all connected quantities.

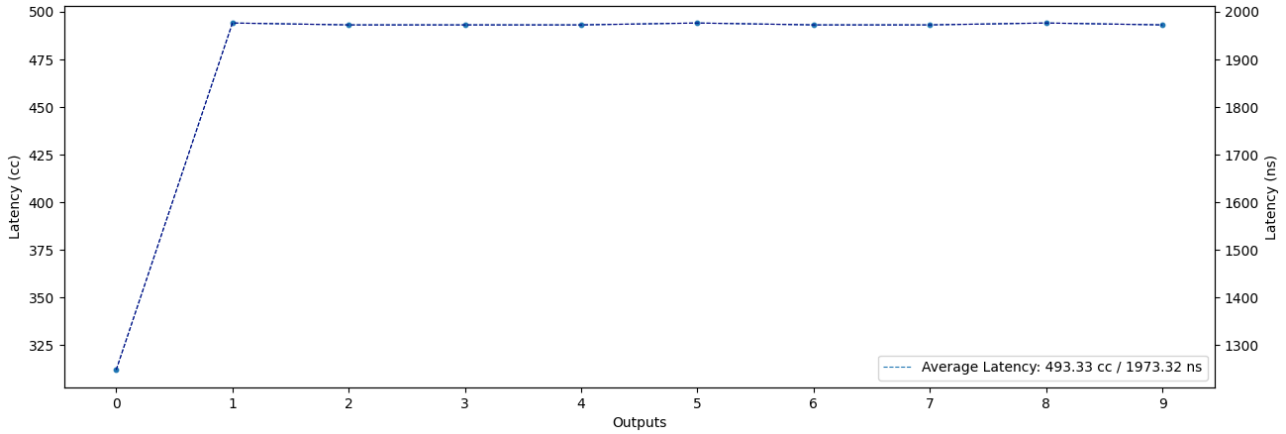


Figure 7.10: Time difference between the first input of the computation and the relevant output.

In real applications, the FIFO depth is an important parameter of the system and can be tuned accordingly in order to leverage the caching effects. It should be used with caution when passing data between AI tiles because they have depth limits that when crossed then performing DMA transfers to the memory is involved which will affect latency.

Chapter 8

Implementation of the Gated Recurrent Unit

8.1 Motivation

The motivation to use a dedicated RNN model is primarily because we would like to somehow encode temporal information to the transformation process of the intensity observations to amplitude and phase actions. We saw that by simply giving feedback of the actions to the Dense layers was not enough to have significant impact on the performance of the agent. But ultimately, a more sophisticated model could be leveraged that both captures and passes on accumulated context acquired by previous observations in the form of a encoded vector. From this point of view we will see that the GRU is going to be an excellent candidate.

As discussed in the previous section about the AI Engine pipeline, some preprocessing kernels are located just before the Dense Feedforward Neural Network. By ignoring the function of the decimators and interpolators for now, we focus on the function of the circular buffer. A quick recap is that this kernel stores the values that are produced by the decimated and filtered observations in a First In First Out order just like it is show in figure 8.1. The size of the buffer, is limited by the size of the Dense Network's input layer. This way of preparing the inputs, behaves like a rolling window in the sense that, produced values that enter the buffer from the left and exit from the right, have the similar effect as if we parsed a given time series of data in a fixed step. This is very similar to how many Recurrent Neural Networks architectures inputs are being prepared and so no modification of the current implementation is necessary.

Most architectures include a Dense FNN after the RNN module, that either transforms the outputs of the RNN into values for regression tasks, or encoded classes for classification tasks. This way, all we have to do is to prepare the computational graph of the GRU, match the inputs and outputs of the graph to the outputs of the observations buffer and the inputs to the Dense FNN layer respectively, and then finally use the graph as a sub-graph of the existing pipeline simply connecting the edges to the kernels.

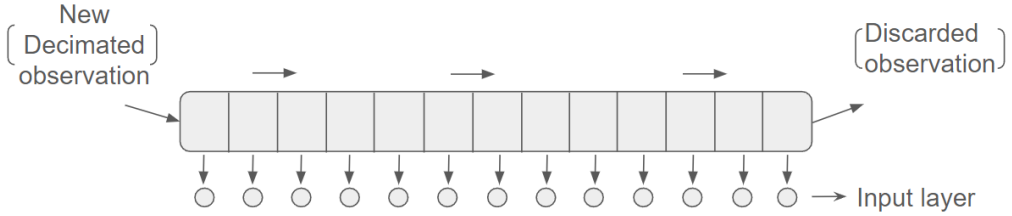


Figure 8.1: The observations buffer

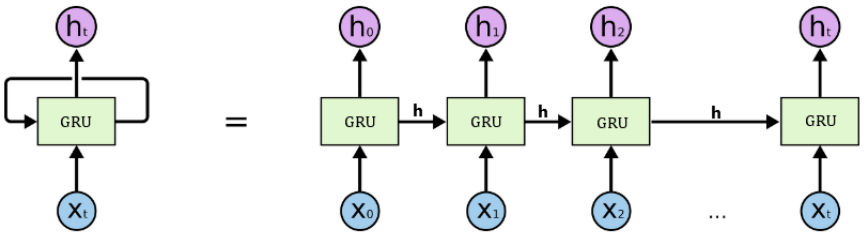


Figure 8.2: The GRU cell used in a recurrent manner to process a sequence of input vectors X [29]. This is called the 'temporally unrolled' rnn.

Subgraph

This subgraph solution sounds elegant in theory but in practice it induces some extra complications with passing the Real Time Parameters which I will discuss in a later section. Nevertheless, we gain in modularity in the sense of a bounded abstraction that allows you to handle only code relevant to the GRU, and future reusability since you would only need to connect the edges. Finally, if someone would like to make a 'deep' GRU architecture, that is to use multiple GRU layers, just instantiate the graph multiple times and connect them.

8.2 Introduction to the GRU

The Gated Recurrent Unit is usually represented as a 'cell' that contains a set of 'gates' that process information. In the top level, the GRU cell can be seen as a function that takes as input the previous hidden state h_{t-1} and the current input x_t and produces the current hidden state h_t. In this sense, the GRU cell is used recurrently to process a sequence of inputs in a timely manner as shown in figure 8.2.

In the special case of initializing the cell, the hidden state h_0 is usually set to all zeros or to a random vector. In this implementation we will use the zero vector.

If we 'open the hood' of the GRU cell, we will see a set of gates represented as a computational graph.

8.2.1 The Reset and Update Gates

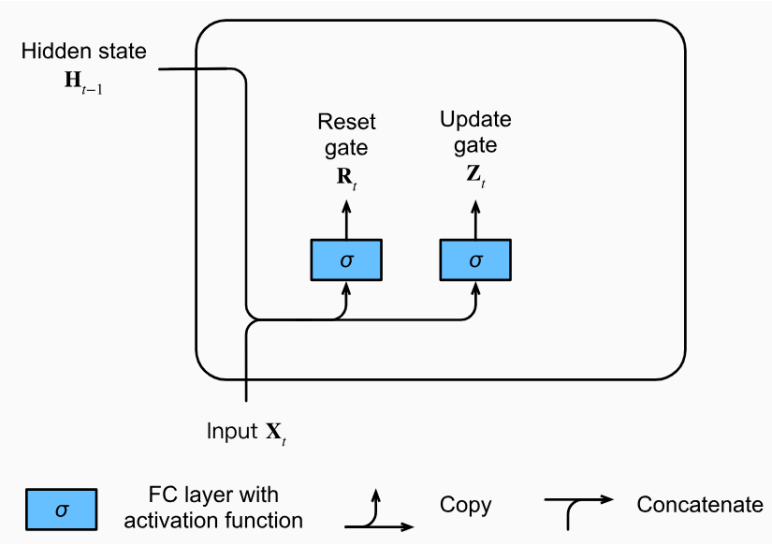


Figure 8.3: The Reset and Update gates [30]

I will be using a set of images to better visualize the flow of the information and the transformation that the data undergoes. Images are taken from [30] which also has great material on the subject.

As mentioned above, the inputs are the previous hidden state h_{t-1} and the current input x_t. The inputs in general are not scalars but vectors. The hidden state is chosen as a hyperparameter of the GRU model by the user. The larger the hidden state, the more information can be stored and retained through the time steps, in the cell. The current input vector length depends a lot on the application. In our case, the input vector is limited by the size of the observations buffer.

For the sake of completeness, I will denote the number of batches as b, but during inference we will be using only one bunch.

In general, the hidden vector will be of size $h \in \mathbb{R}^{b \times h}$ and the input vector will be of size $x \in \mathbb{R}^{b \times n}$ where n is the size of the input vector.

The first step of the GRU cell is to calculate the reset and update gates. The reset gate R_t and the update gate Z_t are calculated by the following equations:

$$R_t = \sigma(W_r h \cdot h_{t-1} + W_r x \cdot x_t + b_r) \tag{8.1}$$

$$Z_t = \sigma(W_z h \cdot h_{t-1} + W_z x \cdot x_t + b_z) \tag{8.2}$$

σ is the sigmoid function that squashes the values between 0 and 1. The variational parameters of this step are:

- $W_{hr} \in \mathbb{R}^{h \times h}$
- $W_{xr} \in \mathbb{R}^{n \times h}$
- $b_r \in \mathbb{R}^{1 \times h}$
- $W_{hz} \in \mathbb{R}^{h \times h}$
- $W_{xz} \in \mathbb{R}^{n \times h}$
- $b_z \in \mathbb{R}^{1 \times h}$

A detail of the computations of the gates is that the bias vector addition only holds dimensionally if the number of batches is 1. For more than one batch, the bias vector is broadcasted to the size of the hidden state. Broadcasted means that the bias vector is simply copied to the size of the batches to form a matrix of size $b \times h$.

8.2.2 The Candidate Hidden State Gate

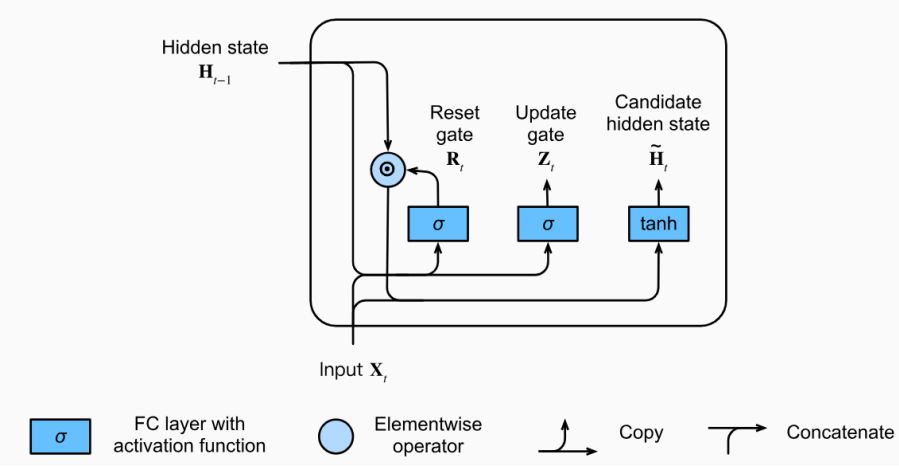


Figure 8.4: The Candidate Hidden State [30]

Once the reset and update gates have been calculated, the reset gate is used to calculate the candidate hidden state $\widetilde{H_t}$. Let me point out the the update gate is not used in the calculation of the candidate hidden state and I will leverage this on the hardware implementation. By observing the equation below, it becomes evident why the previous operation is called 'reset gate' since it acts as a mask to the previous hidden state.

$$\widetilde{H_t} = \tanh(W_{xh} \cdot x_t + (R_t \odot h_{t-1}) \cdot W_{hh} + b_h) \tag{8.3}$$

The $\odot$ operator denotes the element-wise multiplication of the two vectors and the tanh function squashes the values between -1 and 1.

The variational parameters of this step are:

- $W_{hh} \in \mathbb{R}^{h \times h}$
- $W_{xh} \in \mathbb{R}^{n \times h}$
- $b_h \in \mathbb{R}^{1 \times h}$

8.2.3 The New Hidden State Gate

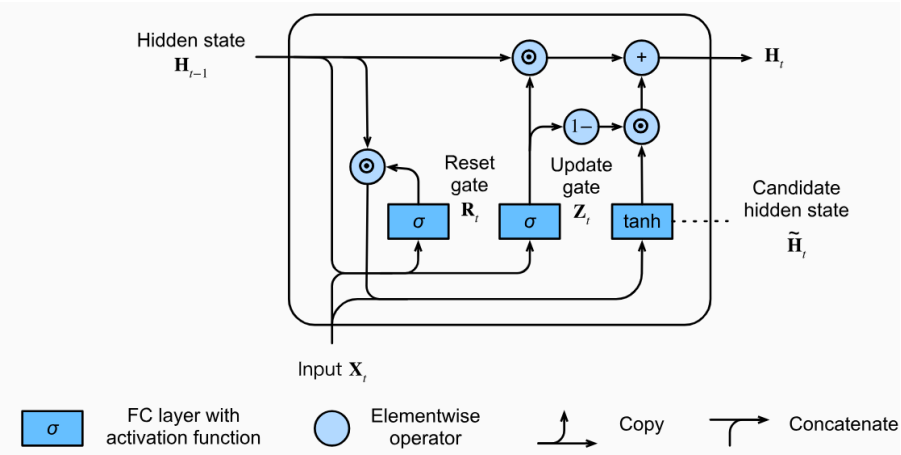


Figure 8.5: The New Hidden State completes the GRU Cell [30]

The final step of the GRU internal operations is the calculation of the new hidden state h_t. The new hidden state uses both the update gate calculated at the first step and the candidate hidden state. The following equation describes the calculation of the new hidden state:

$$h_t = Z_t \odot h_{t-1} + (1 - Z_t) \odot \widetilde{H_t} \tag{8.4}$$

In this step there are no variational parameters, only the products of the previous operations are used. The result, is the final hidden state of the GRU cell of this time step.

With this step we have completed the description of the GRU cell. Our goal now is to implement these operations on the AI Engines.

8.3 Implementation on the AI Engine

As pointed out earlier, the development tools provided by AMD Xilinx allow for the creation of graphs that contain a fixed number of kernels inside them. So, we can use the same idea of a GRU cell, implemented as a graph that contains kernels that perform the operations of the reset and update gates, the candidate hidden state and the new hidden state. Developing the GRU in this mode, allows for the reuse of the graph in different applications and also allows for the easy connection of the graph

to the rest of the pipeline. It also allows for the developer to focus on the internal operation of the GRU cell, which will always be the same, and not worry about the rest of the pipeline.

8.3.1 Memory

There are two main memory constraints. First of all, the variational parameters of the GRU itself, we are limited by the size of the stack memory that can be allocated for a single kernel. Each AI Engine can access three of the memories from neighboring tiles plus its own data memory for a total of 128 kB which means that we can store a maximum of 32768, 32-bit floats. Out of all these floats, some of them are used for the variational parameters of the GRU cell, some of them are used for the intermediate calculations and some of them are used for the inputs and outputs of the kernel. So memory is the main limiting factor of how big the input vector and the hidden state vectors can be, since the variational parameters of the GRU are derived from the lengths of these vectors.

I will use the existing implementation of the dense neural networks of an input layer of 64 neurons and thus the hidden state will also have to be the same size of 64. By fixing the hidden size, the input vector size is now limited to the memory leftover. Heuristically, I have found that the input vector size of 24 does not run out of memory but the size of 32 does, so I will use the size of 24 for the input vector.

In general, I prefer to use a larger hidden state than the input vector since we are operating in a very fast data acquisition environment. The input vector can be a small window of the data, but the hidden state can accumulate more temporal information. Thus, its safe to stop storing observation data that have been already encoded in the hidden state if the hidden state can retain it enough iterations of the GRU, and the only consideration concerns the 'context window' of the model.

If more memory is needed then the only solution would be to use more kernels. This may not be a problem if each kernel can handle a specific GRU operation, but it can be a complex issue if the matrices are split between kernels.

Program Memory

The AI Engine tile is also limited by the size of the program memory. The program memory is the memory that stores the instructions of the kernel and is limited to 16 kB so it is important to keep the code simple and efficient. In my implementation I had exactly this problem for the reset and candidate gate kernel, since the operations of the two gates invoke four matrix vector multiplications and the implementation of two activation functions, among other operations like addition and moving data and so I ran out of memory.

The solution was to implement the matrix vector multiplications and the activation functions in a set of templated functions that can be called in the kernel. This way, the instructions of the functions are stored in the program memory only once and the kernel can call them multiple times. This way, the program memory is used much more efficiently.

There is one implication thought of using this solution. Because AMD Xilinx is using a hierarchical system of compilation for hardware, that is that the user specifies the top level graph and from there everything is compiled down to the kernels, the compiler cannot call the source code of the functions that are being included in the kernel. In a sense, the scoping of the adf graph stops at the kernel level and the source code of the functions that are being called are not visible to the compiler. What is visible is the C++ code that actually includes the header files of the functions.

So the solution is to include the source code of the functions in the header files and then include the header files in the kernel. This way, the source code of the functions is visible to the compiler and the functions can be called by the kernel code.

Finally, this solution, turns out to be very efficient and allows for previously impossible implementations to be done.

8.3.2 The graph and Connections Between Kernels

In the section above, in which I describe the candidate hidden state gate, I mention that the update gate is not used in the calculation of the candidate hidden state. I can leverage this by cramming the calculations of the reset and candidate gate in the same kernel. This way, I can use the output of the reset gate to calculate the candidate without having to lose clock cycles in the communication between the kernels. The graph looks like this:

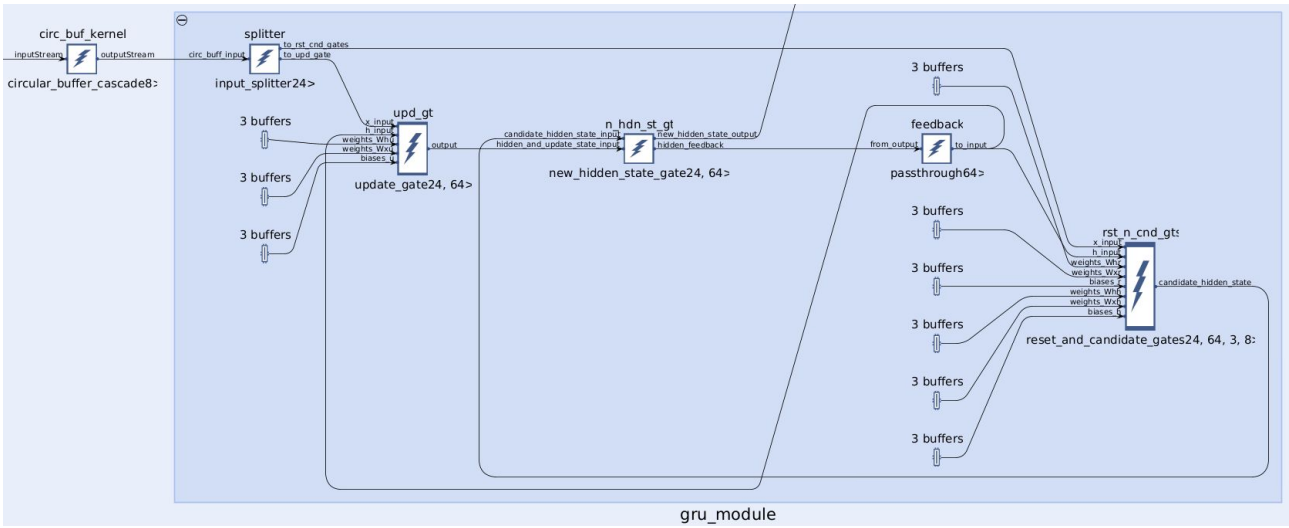


Figure 8.6: The GRU Graph

Outside the sub-graph, the circular (obseravtions) buffer kernel that feeds the size 24 vector input to the GRU graph is clearly visible. In the existing implementation, the circular buffer would feed the input to the Dense FNN kernel and this was done using a cascade stream. The cascade stream it is a very fast way to pass chucks of data between kernels, specifically 8 float at a time, but it suffers from the fact that it cannot pass a TLAST signal, since it is not an AXI4-Stream. In order to mitigate this problem, a special command is being passed after every interaction of the cascade stream to the kernel that is being fed. This command is a v8float vector that contains all zeros and it is used to signal the stop of operations of the kernel by changing the first element of the vector to 1.

Also, recall that each kernel can have up to 2 inputs and 2 output. This is important because if the circular buffer passes the input as a cascade stream, then inside the GRU graph the input vector splits as separate streams to the reset and candidate and update kernel respectively, but only one of them can continue as a cascade stream. I feed the reset and candidate kernel with the cascade stream and the update kernel with a normal AXI4-Stream, for the sole reason that the reset and candidate kernel incorporates 2 gates and thus it will take more time to compute the output.

In general all the connections of the hidden state to kernels are done using the AXI4-Stream.

After the new hidden state kernel has performed the necessary operations, the hidden state is passed to the input layer of the dense neural network kernel and to handle the feedback of the hidden state to the next iteration of the cell, I employ the use of a dedicated kernel that simply passes the hidden state to the reset and candidate and update kernel gates. This has the purpose of helping with the potential stalling in passing the hidden state to the next iteration, and also makes the graph more modular.

As the reader can see, the graph is very different from the internal operations of the GRU cell as presented at the beginning of the section. In the future, it would be interesting to see a codebase that automatically generates the needed graph, given the relevant parameters of the GRU cell, such as input vector size, hidden state size, bi-directional etc. The implementation should then figure out how to split the calculation into multiple kernels and how to connect them in the most efficient way.

Although a lot of though has been put into the design of the graph, it is possible that more op-

timizations can be done. One possibility would be to implement the usage of windows instead of AXI4-Stream for the hidden state.

8.3.3 A Special Connection: The Hidden State Feedback

As explained in the introduction of the GRU, once the hidden state has been calculated, it is passed to the reset and candidate and update kernel gates. The need to feedback the hidden state that we calculated to the next iteration of the GRU cell, breaks the pipelining philosophy of the AI Engine. This issue is really impossible to avoid since the hidden state is calculated at the end of the graph and the next iteration of the GRU cell absolutely requires the previous hidden state to start computation. The only possible solution is to minimize the latency.

Initialization of the Hidden State

The first iteration of the GRU pipeline is a special case which if not addressed properly will cause the whole pipeline to stall. A simple solution is to have a dedicated if statements inside both the update and reset and candidate gates kernels that check if the computations are being done for the first iteration. If this is the case, then the hidden state is initialized to the zero vector, and this is also consistent with how the GRU is usually being trained in the offline model. This is also the easiest to implement and does not require any extra outside information.

The code snippet below shows how the hidden state is initialized to the zeros in the first iteration of the GRU pipeline, while it usually reads data from the input stream:

```
bool first_iteration_flag = true;

    for (;;) {

            // Read Input Data
            for(int i=0; i< x_input_size; i++)
                    x[i] = readincr(x_input, tlast);

            if (first_iteration_flag) {
                    for(int i=0; i < h_input_size; i++)
                            h[i] = 0;
                    first_iteration_flag = false;
            }
            else {
                    for(int i=0; i < h_input_size; i++)
                            h[i] = readincr(h_input);
            }

...

...

}
```

The second choice, which is more elegant, is to pass the previous hidden state as a Real Time Parameter to the GRU subgraph. This way, the hidden state does not need to converge every time to the 'correct' value. This solution provides more information to the GRU cell and it is a better way to treat the hidden state, especially if we consider the fact that training occurs relatively frequent when compared to the inference. Thus, initializing the hidden state always to zero could affect the performance of the model during inference more.

This valid concerns are not addressed in this implementation, but it is a good idea to keep them in mind for future implementations and of course they should be tested in a real test implementation.

8.3.4 Pipelining the Matrix-Vector Multiplication

The main computational bottlenecks of the GRU cell are the matrix - vector multiplication and the application of the non-linear activation functions. In this section, I will show how to pipeline the matrix - vector multiplication in order to increase the throughput of the kernel by using the C++ intrinsic functions of the AI Engine API.

The computational workhorse of the matrix - vector multiplication will be the intrinsic function `fpmac` that performs the multiplication of two vectors and the addition of the result to a third vector. The function can perform 8 multiplications and 8 additions in parallel by leveraging the SIMD capabilities of the AI Engine and the specialized hardware. This is also the reason why the input and hidden state vectors are conveniently set to sizes that are multiples of 8.

The first thing that must happen in order to leverage the `fpmac` function is to convert most variables to the `v8float` type. This type is a vector of 8 floats that is compatible with the `fpmac` function and thus any time we are pointing to an array of v8floats we are effectively referring to 8 floats at a time. From the point of view of v8floats the weight matrices of the GRU should be viewed as sets of vectors of 8 floats.

In order to leverage the power of the vectorized computations of the AI Engine, I must discuss the `fpmac` arguments. Here is a direct screenshot from the API documentation:

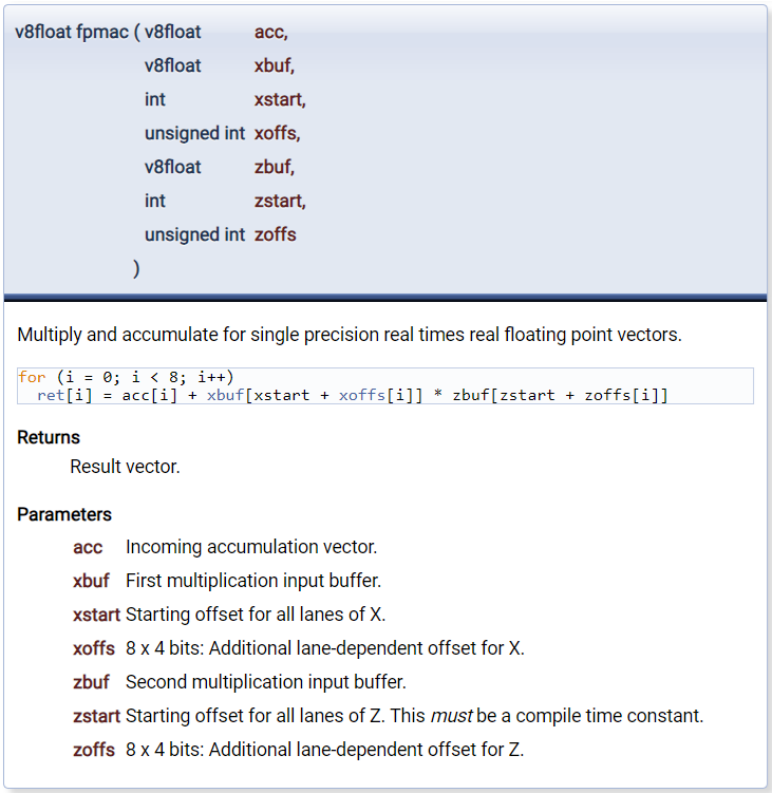


Figure 8.7: The fpmac function. Picture taken from the AI Engine Intrinsics User Guide [25]

The accumulator, xbuff and zbuff are all v8float vectors and are the data inputs to the multiply and accumulate operation. Additionally, the AI Engine vector processor has the capability of shuffling the elements in the operands before the multiplication. The important arguments are the `start` and `offs`. These parameters allow the user to specify, for the xbuff (lets say first vector) and zbuff (the other vector of 8 floats) respectively, the starting point and the 'permutation' of the elements vector. The starting point is simply the index of the first element of the vector that we want to start the multiplication. By specifying the starting point index, the whole vector is transposed so that the first element of the vector is the element at the starting point index.

An example: if we have a v8float vector that contains the elements [0, 1, 2, 3, 4, 5, 6, 7] and we specify

the starting point index to be 3, then the vector will be shuffled to [3, 4, 5, 6, 7, 0, 1, 2].

The offs parameter is used to specify the permutation or 'shuffling' of the vector. An example will clearly illustrate this. If we have a v8float vector that contains the elements [0, 1, 2, 3, 4, 5, 6, 7] and we specify the offs as 0x76540123 (you can pass them as a hex), then the vector will be permuted to [7, 6, 5, 4, 0, 1, 2, 3].

Thus, we can also have the combined effect of the starting point and the offs parameter.

I will leverage only the starting point parameter for the matrix - vector multiplication. The code snippet below demonstrates how this is done:

```
for (int i = 0; i < h_vector_size; i++)
    result[i] = null_v8float();

// Cycle over input vectors
for (int inp_vec = 0; inp_vec < vector_size; inp_vec++)
    // Cycle over input vector elements
    for (int inp_elem = 0; inp_elem < 8; inp_elem++)
        // Cycle over output buffer
        for (int out_vec = 0; out_vec < h_vector_size; out_vec++)
            result[out_vec] = fpmac(

                result[out_vec],
                vector[inp_vec],
                inp_elem,
                0x0,
                matrix[out_vec+inp_elem*h_vector_size+inp_vec*h_vector_size*8],
                0x0,
                0x76543210
                );
```

There is one subtle convention used in the code above, and that is the way that the weight matrices are stored. In physical memory, any array is stored in a linear fashion, that is a block of memory that contain all the elements of the array. But in the python code matrices will be access as 2D arrays, so the convention used to parse the rows and columns to just a single index is important. This convention will also define the way we construct the 'function' that access the correct element of the matrix. In this case, the matrix is passed in column-wise order which is different than the default order of Pytorch layers (which are row-wise). This is due to the fact that AI Engines miss an intrinsic efficiently summing the elements of a v8float. So, this is how the function that calculates the index of the weight matrix is derived and it always points to the correct 8 floating point numbers that is needed for the multiplication. Finally, the starting point and the offs parameters are set to 0x0 and the correct order of elements 0x76543210 respectively, since we are not interested in any permutation or transpose of the elements.

The variable 'result' is the output of the matrix - vector multiplication and it has to be initialized to the zero vector before the multiplication. It is also used as the 'accumulator' of the fpmac function and it is a v8float vector.

The reader can spot that the usage of the start point and offs arguments for the input vector are different from the defaults used for the matrix. This is a crucial point to leverage the vectorized computations of the AI Engine. By specifying 0x76543210 for the matrix and 0x0 for the vector we are effectively instructing the fpmac to multiply all the elements of the matrix vector with the same element of the input vector. Which element? Exactly the one specified by the starting point index. In order to better understand how this parses through the matrix and the vector I will provide a graphical explanation in figure 8.8.

The figure 8.8 shows how the loop has been constructed in order to leverage the vectorized computations of the AI Engine. The inner loop is represented by the blue enclosures and will parse the accumulator vector from top to bottom. At every step the middle loop takes care that the correct single element of the input vector is multiplied with the correct 8 elements of the matrix vector. The

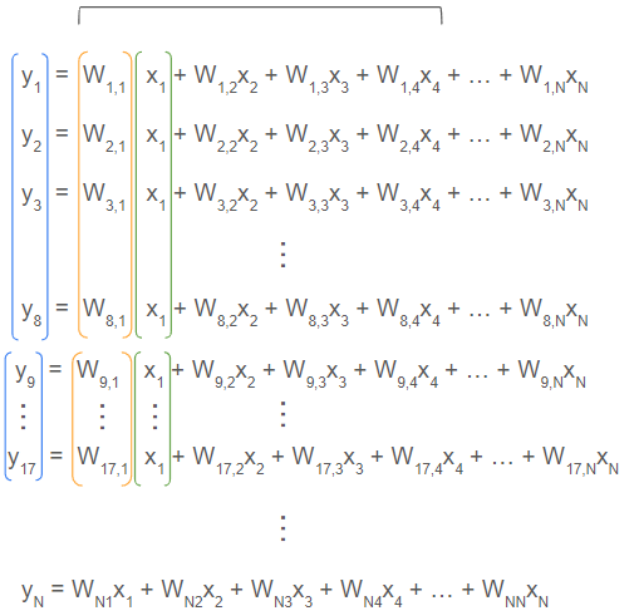


Figure 8.8: Element parsing using the fpmac operation. A column containing 8 matrix elements (yellow) and the same instance of the input vector element (green) get multiplied until they reach 'the bottom'. Then the next column multiplications start and accumulate on the previous result. Repeat for all columns.

single input vector element does not change while the inner loop is parsing the relevant column. The outer loop is responsible to advance the input vector to the next 8 relevant elements in which the middle loop will parse. The user can think about this as a long series of multiplications that are waiting to be computed and the operation is computing them for every element of the output vector. Once this its done it moves on to the next columns of operations, all the way to the end of the matrix.

8.3.5 Activation Functions

Calculating the activation functions is probably the biggest computational bottleneck of the GRU cell, namely because it uses the exponential function. The sigmoid and hyperbolic tangent function are the two activation functions that are used in the GRU cell and are defined like this:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{8.5}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{8.6}$$

In general there are two design choices in order to implement activation functions. If the developer has access to a lot of memory, the precomputed values of the activations functions can be stored in a look-up table and accessed in a single cycle. This is a very fast way to calculate the activation functions, but depending on the available memory it can be very granular. If the developer has access to a lot of computational power, the activation functions can be approximated by a polynomial and calculated in a few cycles. This is a slower way to calculate the activation functions but it is more memory efficient. Also the polynomial approximation, written in a specific way, can be pipelined to increase the throughput of the kernel.

In this implementation, the memory of single AI Engine tiles is extremely limited and thus the polynomial approximation is a sensible way to go.

Applying non linear activation functions, is trivial in Python, but the AI Engine miss both a vector division and exponential instrinsics functions in your disposal. What you do have is the ability to

perform multiplications and additions, which naturally leads to the idea of approximating the functions with polynomials. Additionally, since both activation function are odd, each term in the polynomial will have to scale by a power factor of 2.

Since the model that runs on hardware is coupled to its Python counterpart, which trains offline, the implementation of the activation functions introduce some extra complications. Namely, PyTorch calls activation functions implemented in C++ and are not available for customization in the source code. This means that the activation functions that are used in the training of the model are not the same as the ones that are used in the hardware implementation and in this point there are a couple of options.

The first option is to use the same approximated activation functions used in the hardware implementation in the training phase of the model. This is a good idea since the variational parameters will train to the same activation functions that will be used in the hardware implementation. The problem lies in the fact that in order to use custom activation functions in PyTorch, you will also have to implement a custom GRU cell, and since it is not running the C++ implementation, we are losing a lot of the optimizations that are done in the C++ implementation. Slower training, means more time that the inference model will be using old weights. Nevertheless, this is probably the 'safest' route, to be sure that the model will perform as expected of its python twin.

The second option is to try and approximate the activation functions in hardware as close as possible to the real ones and thus use the real activation functions in the training. This approach is more difficult to pipeline mainly because you will have to divide the activation function into regions in which different polynomials are used. This is because you cannot rely on a single Taylor expansion to approximate the function in the whole relevant domain. Thus, designing a pipeline that takes advantage of the vectorized computations of the AI Engine, can be a challenge.

A third option could be to use the exact activation functions in the training and then 'naively' use the approximated activation functions in the hardware implementation hoping that either the degree of error is small or that the model is robust to the error. I did try this approach in the python implementation, in a simple test case but I cannot guarantee that it will work in the real world. A smarter approach would be to train on the exact activation functions until the model converges and then switch to the approximated activation functions for the last few training epochs. This way, the model takes advantage of the exact activation functions in the beginning and then it is robust to the approximated ones in the end. I didn't try this approach.

In the end, due to the convenient use of existing AI Engine intrinsic functions, and the ability to create the custom GRU cell in PyTorch, I will use the first approach.

Polynomial Aproximation

First of all, I will need to understand the polynomial and how to construct it. The classic way to approach this would be to start expanding the function as a Taylor series around zero, (the Maclaurin series expansion) and then use a heuristic cutoff point in which the function is always a steady value.

The Taylor expansion of the tanh is defined as a series of odd powers of x and formally invokes the use of Bernoulli numbers. The final form of the expansion is:

$$\tanh(x) = x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \frac{62x^7}{2835} - \dots \tag{8.7}$$

and for the sigmoid, since it can be expressed as a translation and a scaling of the tanh function, the expansion is:

$$\sigma(x) = \frac{1}{2} + \frac{1}{2}\tanh(\frac{x}{2}) = \frac{1}{2} + \frac{x}{4} - \frac{x^3}{48} + \frac{x^5}{480} - \frac{17x^7}{80640} + \dots \tag{8.8}$$

The problem with this approach is that because we are expanding around zero, the approximation is not very good for large values of x. By using a cutoff scheme as show in equation 8.9, I can probe the behavior of the approximation.

$$\tanh(x) = \begin{cases} -1 & \text{for } -\infty < x \leq -\alpha, \\ x - \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \frac{62x^9}{2835} - \dots & \text{for } -\alpha \leq x \leq \alpha, \\ 1 & \text{for } \alpha \leq x < \infty \end{cases} \tag{8.9}$$

In the figure 8.9 below, I show how the polynomial 'explodes' for large values of x, just because the last, and higher order term of the expansion dominates.

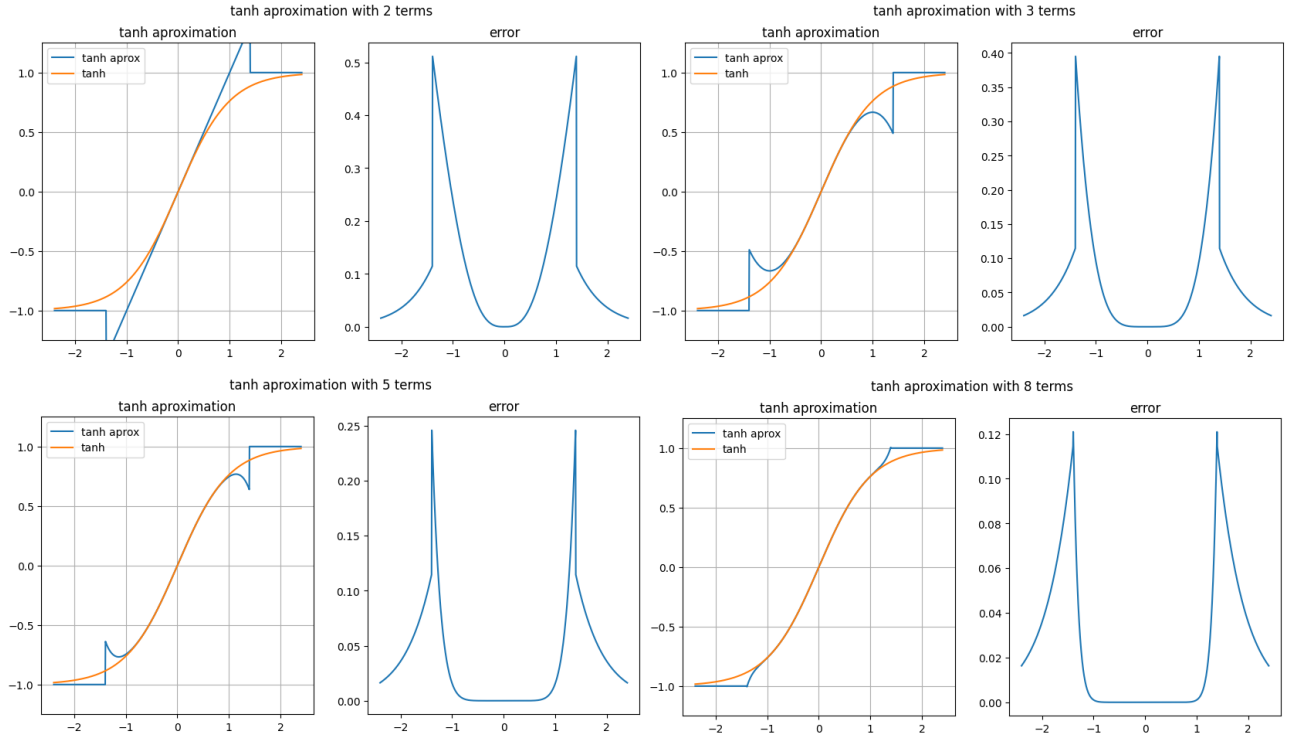


Figure 8.9: The Taylor Expansion of the tanh function with a cutoff like equation 8.9. The polynomial will explode as the high order term takes effect.

Spline Approximation

To tackle this problem of exploding polynomials, I will approximate the terms of the expansion with a polynomial of 3rd degree and by using a fixed 'cutoff' value. Having these two parameters, I can construct a spline that uses a 3rd degree polynomial for the region around zero and impose a fixed value exactly at the cutoff point. When implementing the spline, I can suppress all values of x that are larger or smaller than the cutoff points and just compute the polynomial exactly at the cutoff point. This way, I can avoid the 'explosion' of the polynomial, have a stable approximation and smooth transition and I don't have to use scalar operations to check the value of x.

I don't have to use the same cutoff point for both the activation functions but in this preliminary implementation I will. The cutoff points are important parameters of the approximation and optimal values do exist. In this case, I will use the value of 2 and -2 for the both the tanh and the sigmoid function.

So the problem at hand is: Calculate the coefficients of a 3rd degree polynomial:

$$f(x) = a_0 + a_1 x + a_2 x^3 \tag{8.10}$$

Such that these conditions hold:

For the tanh function:

- $f(-2) = -1 \Rightarrow a_0 - 2a_1 - 8a_2 = -1$
- $f(2) = 1 \Rightarrow a_0 + 2x + 8x^3 = 1$
- $f'(2) = f'(-2) = 0 \Rightarrow a_1 + 12a_2 = 0$

With solutions:

- $a_0 = 0$
- $a_1 = \frac{12}{16}$
- $a_2 = -\frac{1}{16}$

And for the sigmoid function:

- $f(-2) = 0 \Rightarrow a_0 - 2a_1 - 8a_2 = -1$
- $f(2) = 1 \Rightarrow a_0 + 2x + 8x^3 = 0$
- $f'(2) = f'(-2) = 0 \Rightarrow a_1 + 12a_2 = 0$

With solution:

- $a_0 = \frac{1}{2}$
- $a_1 = \frac{12}{32}$
- $a_2 = -\frac{1}{32}$

And thus the final splines are:

$$\tanh(x) \sim \frac{12}{16}x - \frac{1}{16}x^3 \tag{8.11}$$

$$\sigma(x) \sim \frac{1}{2} + \frac{12}{32}x - \frac{1}{32}x^3 \tag{8.12}$$

We would like to use intrinsic functions to squash the values of x that are larger than 2 to exactly 2 and then compute anyway the polynomial, and the same for values smaller than -2. This way, there is no needto check in which region the value of x is and I can use the vectorized computations of the AI Engine.

The final splines, compared graphically to the real functions are shown below:

Pipelining the Activation Functions

I mentioned above that the splines where chosen because they can be pipelined. The reason for this is that the polynomial is of 3rd degree and thus the computation of the polynomial can be split into stages. First stage is the use of the intrinsic functions `fpmin` and `fpmax` to squash the values of x that are larger than 2 to 2 and the values that are smaller than -2 to -2. Second stage is to use `fpmul` two times to multiply the x values with themselves and effectively calculate the x cubed values. Final stage is to use `fpmac` to calculate the polynomials by first using as an accumulator the 0th order coefficients, and then multiplying x with the corresponding 1st and 3rd order coefficients.

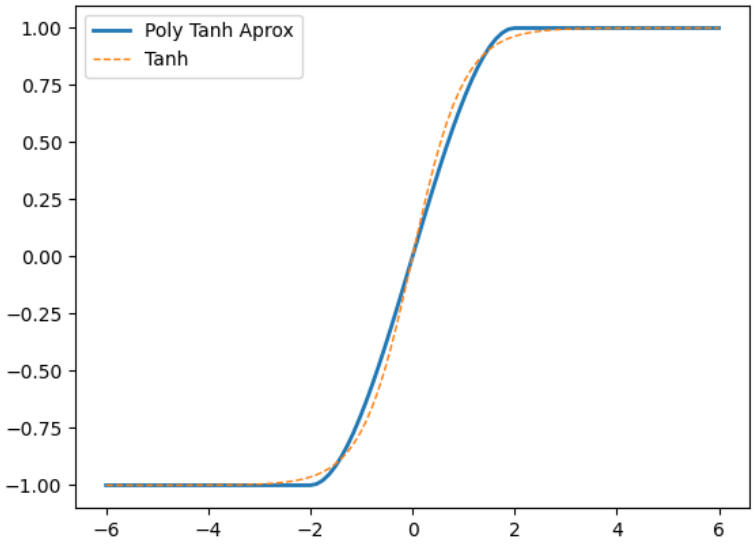


Figure 8.10: The Spline Approximation of the tanh function

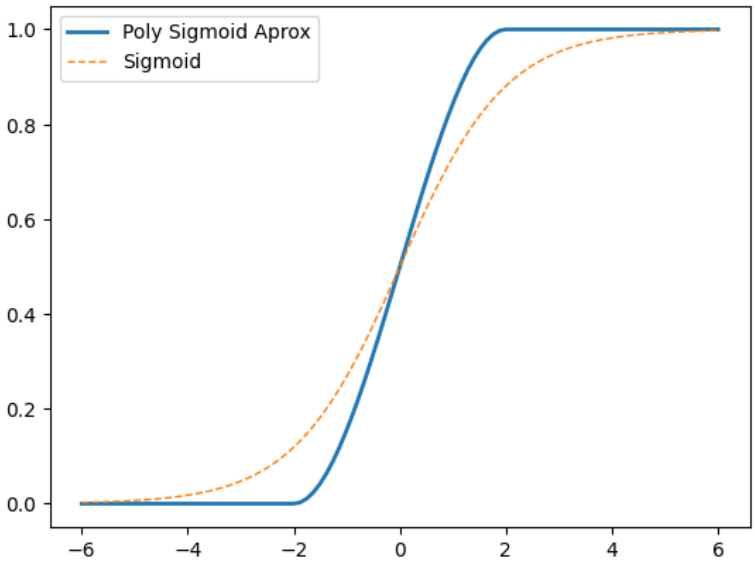


Figure 8.11: The Spline Approximation of the sigmoid function

An important prerequisite for compiler optimization is to not use operations that need to wait for a result of a previous operation. This is why instead of using a for loop that contains the stages in sequence I will use a for loop for each stage. This way, the compiler can figure out which results are ready in order to begin computation of the next stage. This was tested and it is indeed faster than using a for loop that contains all the stages, by a margin of a few hundreds of nanoseconds.

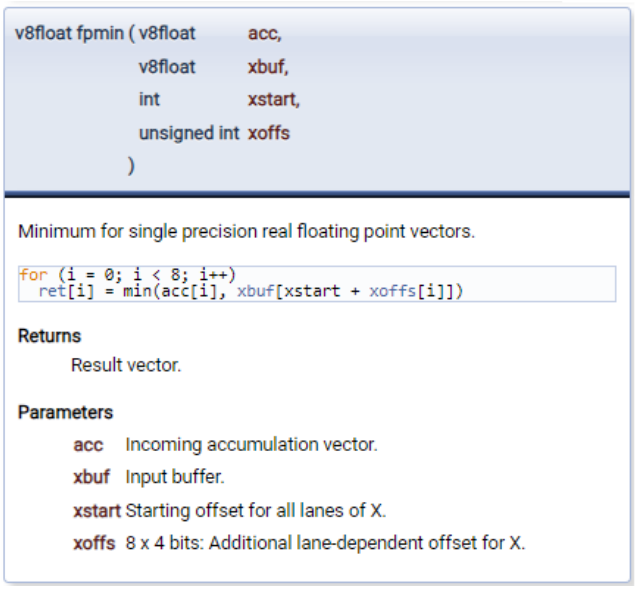
The activation function code was conveniently developed as a template function so that the internal loops know the size of the input vector which is the same with the hidden vector. The user can choose which activation function to use by passing the correct coefficients to the function. The code snippet below demonstrates how this is done:

```
template<int vector_size>
void act_func(v8float* input,
              v8float* output,
              float max_range,
              float c0,
              float c1,
              float c2) {
```

```
        v8float pos_range;
        v8float neg_range;
        v8float coeff_0;
        v8float coeff_1;
        v8float coeff_2;

        // Preparing the comparing vectors and the
        // coefficients as v8floats using the update_elem function
        for(int i=0; i<8; i++){
            pos_range = upd_elem(pos_range, i,  max_range);
            neg_range = upd_elem(neg_range, i, -max_range);
            coeff_0 = upd_elem(coeff_0, i, c0);
            coeff_1 = upd_elem(coeff_1, i, c1);
            coeff_2 = upd_elem(coeff_2, i, c2);
        }

        v8float input_3; // This is the x^3 vector

        // Stage 1 - Squashing the values of x that are larger
        // than max_range to max_range and the values
        // that are smaller than -max_range to -max_range
        for(int i=0; i<vector_size; i++){
            input[i] = fpmin(input[i],
                             pos_range);};

        for(int i=0; i<vector_size; i++){
            input[i] = fpmax(input[i],
                             neg_range);};

        // Stage 2 - Multiplying the x values with themselves to acquire x^2
        for(int i=0; i<vector_size; i++){
            input_3 = fpmul(input[i],
                            input[i]);};

        // Multiply the previous result x^2 with x one more time to acquire x^3
        for(int i=0; i<vector_size; i++){
            input_3 = fpmul(input_3,
                            input[i]);};

        // Stage 3 - Calculating the polynomial
        for(int i=0; i<vector_size; i++){
            output[i] = fpmac(coeff_0,    // Using coeff_0 as the accumulator
                              input[i],
                              coeff_1);};

        for(int i=0; i<vector_size; i++){
            output[i] = fpmac(output[i],
                              input_3,
                              coeff_2);};
    };
```

The intrinsics used above are very similar on their usage with the fpmac that we discussed above. Some of them, like the `fpmin` and `fpmax` have the ability to also permute the elements of the `xbuff`. For the sake of completeness, figures 8.12, 8.13, and 8.14 are taken from the documentation of the intrinsic functions used in the activation function code above:

Here, I would like to note a possible further optimization that can be done, since these intrinsics can also be used with larger vector data types, like v16floats. This means that the activation function can be calculated for 16 floats at a time, using a comparison vector of only v8floats. This is another example of how the AI Engine can be used to its full potential but it takes time and effort to understand and test the capabilities of the hardware.



```
v8float fpmin ( v8float     acc,
                v8float     xbuf,
                int         xstart,
                unsigned int xoffs
              )
```

Minimum for single precision real floating point vectors.

```
for (i = 0; i < 8; i++)
  ret[i] = min(acc[i], xbuf[xstart + xoffs[i]])
```

Returns

Result vector.

Parameters

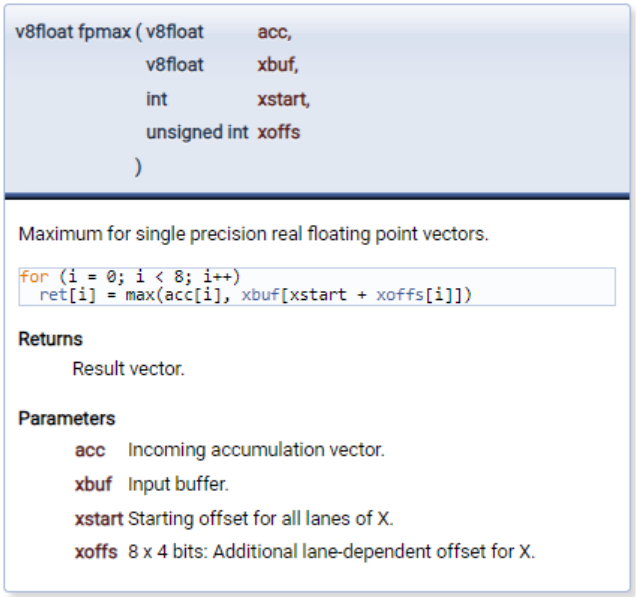
acc Incoming accumulation vector.

xbuf Input buffer.

xstart Starting offset for all lanes of X.

xoffs 8 x 4 bits: Additional lane-dependent offset for X.

Figure 8.12: The fpmin function as given by the AI Engine Intrinsics User Guide [25]



```
v8float fpmax ( v8float     acc,
                v8float     xbuf,
                int         xstart,
                unsigned int xoffs
              )
```

Maximum for single precision real floating point vectors.

```
for (i = 0; i < 8; i++)
  ret[i] = max(acc[i], xbuf[xstart + xoffs[i]])
```

Returns

Result vector.

Parameters

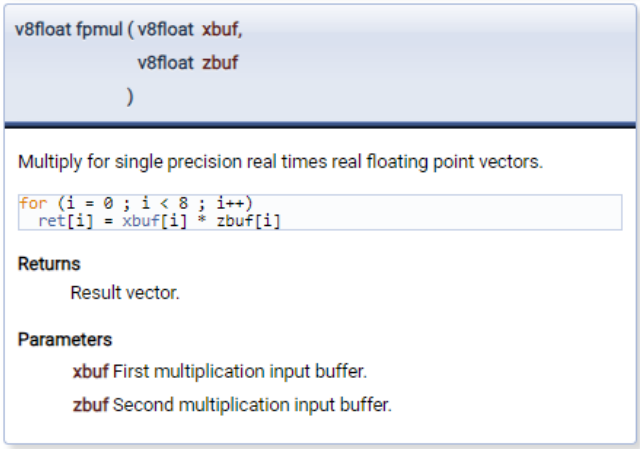
acc Incoming accumulation vector.

xbuf Input buffer.

xstart Starting offset for all lanes of X.

xoffs 8 x 4 bits: Additional lane-dependent offset for X.

Figure 8.13: The fpmax function as given by the AI Engine Intrinsics User Guide [25]



```
v8float fpmul ( v8float xbuf,
                v8float zbuf
              )
```

Multiply for single precision real times real floating point vectors.

```
for (i = 0 ; i < 8 ; i++)
  ret[i] = xbuf[i] * zbuf[i]
```

Returns

Result vector.

Parameters

xbuf First multiplication input buffer.

zbuf Second multiplication input buffer.

Figure 8.14: The fpmul function as given by the AI Engine Intrinsics User Guide [25]

8.3.6 Passing Real Time Parameters to the Subgraph

I'm including this section because this is not covered in the official documentation of the AI Engines and it boils down to this: If you implement a subgraph how do you pass the variational parameters of the relevant weight matrices of the GRU, from the main graph to the subgraph? It turns out that the answer is a little more complicated than it seems because you will have to first pass the parameters through the main graph and then through the subgraph.

The first step is to pass the parameters as parameter ports in the main graph but scope them into the subgraphs instantiation parameter ports, in this example called 'gru_module'.

```
adf::connect<adf::parameter>(weights_Whr, adf::async(gru_module.weights_Whr));
adf::connect<adf::parameter>(weights_Wxr, adf::async(gru_module.weights_Wxr));
adf::connect<adf::parameter>(biases_r, adf::async(gru_module.biases_r));
adf::connect<adf::parameter>(weights_Whh, adf::async(gru_module.weights_Whh));
adf::connect<adf::parameter>(weights_Wxh, adf::async(gru_module.weights_Wxh));
adf::connect<adf::parameter>(biases_h, adf::async(gru_module.biases_h));
adf::connect<adf::parameter>(weights_Whu, adf::async(gru_module.weights_Whu));
adf::connect<adf::parameter>(weights_Wxu, adf::async(gru_module.weights_Wxu));
adf::connect<adf::parameter>(biases_u, adf::async(gru_module.biases_u));
```

Then inside the subgraph, you will have to re-define and re-pass the parameters to the subgraph kernels:

```
adf::connect<adf::parameter> (weights_Whr, adf::async(rst_n_cnd_gts.in[2]));
adf::connect<adf::parameter> (weights_Wxr, adf::async(rst_n_cnd_gts.in[3]));
adf::connect<adf::parameter> (biases_r, adf::async(rst_n_cnd_gts.in[4]));
adf::connect<adf::parameter> (weights_Wxh, adf::async(rst_n_cnd_gts.in[6]));
adf::connect<adf::parameter> (weights_Whh, adf::async(rst_n_cnd_gts.in[5]));
adf::connect<adf::parameter> (biases_h, adf::async(rst_n_cnd_gts.in[7]));

adf::connect<adf::parameter> (weights_Whu, adf::async(upd_gt.in[2]));
adf::connect<adf::parameter> (weights_Wxu, adf::async(upd_gt.in[3]));
adf::connect<adf::parameter> (biases_u, adf::async(upd_gt.in[4]));
```

So the 'flow' of the parameters is from the main graph to the subgraph and then to the subgraph kernels.

Finally, in order to update them in real time using an AI Engine application or the main graph source code using one of the available simulations, you will have to only update the parameter ports of the main graph and they will automatically flow to the subgraph and the subgraph kernels.

8.4 The Custom GRU Cell in PyTorch

The custom GRU implementation was absolutely needed in order to use the custom activation functions approximated as splines. Furthermore, it can be used as a numerical validation tool for the hardware implementation. The custom GRU cell was implemented as a subclass of the torch.nn.Module class and it is shown below:

```
import torch
import torch.nn as nn

class GRU(torch.nn.Module):

def __init__(self, input_dim, hidden_dim, sigmoid, tanh):
    super(GRUCell, self).__init__()
    self.sigmoid = sigmoid
    self.tanh = tanh
    self.input_dim, self.hidden_dim = input_dim, hidden_dim
    self.relevance_whh, self.relevance_wxh, self.relevance_b = self.create_gate_parameters()
    self.update_whh, self.update_wxh, self.update_b = self.create_gate_parameters()
    self.candidate_whh, self.candidate_wxh, self.candidate_b = self.create_gate_parameters()
```

```
def create_gate_parameters(self):
    input_weights = nn.Parameter(torch.zeros(self.input_dim, self.hidden_dim))
    hidden_weights = nn.Parameter(torch.zeros(self.hidden_dim, self.hidden_dim))
    nn.init.xavier_uniform_(input_weights)
    nn.init.xavier_uniform_(hidden_weights)
    bias = nn.Parameter(torch.zeros(self.hidden_dim))
    return hidden_weights, input_weights, bias


def forward(self, x, h):
    output_hiddens = []

    for i in range(x.shape[1]):
        relevance_gate = self.sigmoid(
                                    torch.matmul(h,self.relevance_whh)
                                    + torch.matmul(x[:, i],self.relevance_wxh)
                                    + self.relevance_b)

        update_gate = self.sigmoid(
                                   torch.matmul(h,self.update_whh)
                                   + torch.matmul(x[:, i],self.update_wxh)
                                   + self.update_b )

        candidate_hidden = self.tanh(
                                    torch.matmul((relevance_gate*h),self.candidate_whh)
                                    + torch.matmul(x[:, i],self.candidate_wxh)
                                    + self.candidate_b)

        h = (update_gate * candidate_hidden) + ((1 - update_gate) * h)

        output_hiddens.append(h.unsqueeze(1))

    return torch.concat(output_hiddens, dim=1), h
```

The GRU cell will return the hidden states of all the time steps as a sequence and the final hidden state. The activation functions are passed as arguments to the constructor of the class since I was exploring their effect on the training of the model.

8.4.1 The Custom Activation Functions

In order to use custom activation functions in PyTorch, the user has to define them by using the PyTorch torch.autograd.Function class. This is a class that has two static methods, the forward and the backward method. The forward method is used to calculate the output of the function and the backward method is used to calculate the gradient of the function. The gradient is nothing less than the derivative of the input with respect to the polynomial.

In order to be able to handle a sequence of inputs, the custom activation functions have to be vectorized. This is achieved by using masks that are applied to the input tensor. The numpy equivalent of this operation is the np.where function.

Finally, in order to use the custom activation function, I am invoking the function as a class method and passing the input tensor as an argument. This is done, because the custom activation function is a class and not a function, in which the we have to use the method apply to invoke the function.

The code snippet below shows how the sigmoid is implemented as a custom function using the spline approximation:

```
class SigmoidPolyFunction(torch.autograd.Function):
@staticmethod
def forward(ctx, x):
    glue_point = 2
    ctx.save_for_backward(x)
    output = (
    torch.mul((x <= -glue_point).float(), 0.0) +
```

```
    torch.mul((x >= glue_point).float(), 1.0) +
    torch.mul(((x > -glue_point) & (x < glue_point)).float(), (0.5 + (12/32)*x - (1/32)*x**3))
    )
    return output

@staticmethod
def backward(ctx, grad_output):
    glue_point = 2
    x, = ctx.saved_tensors
    grad_input = (
        torch.mul((x <= -glue_point).float(), 0.0) +
        torch.mul((x >= glue_point).float(), 0.0) +
        torch.mul(((x > -glue_point) & (x < glue_point)).float(), (12/32 - (3/32)*x**2))
    )
    return grad_output * grad_input

# Define a module using the custom polynomial sigmoid function
class SigmoidPoly(nn.Module):
    def forward(self, input):
        return SigmoidPolyFunction.apply(input)
```

In exactly the same way the tanh function can be implemented.

Finally, I devised a simple test to check how the approximated activation functions perform in the training of the model. The test was done in a simple dataset of a nonlinear amplitude envelope of a sine wave as show in the plot 8.15.

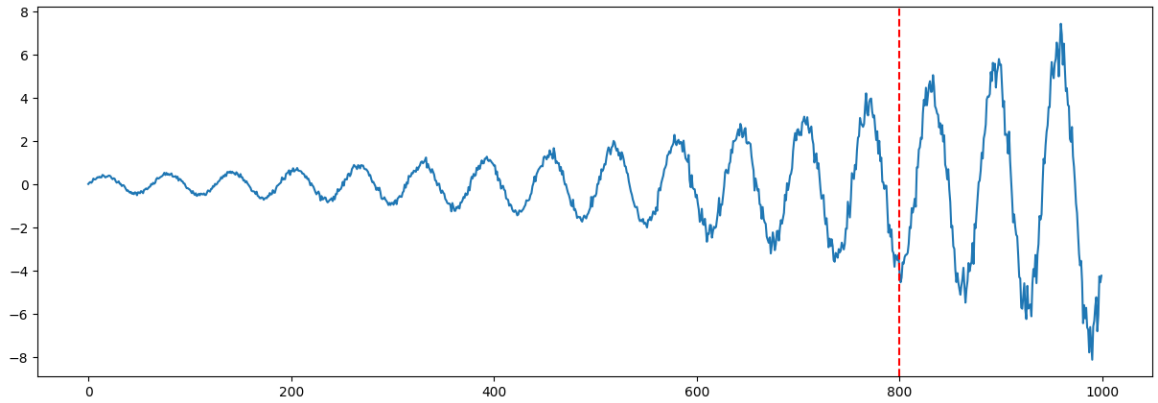


Figure 8.15: The dataset used to test the custom GRU implementation. The red line marks 80% of the dataset used for training at the left and the rest is used for testing. A performance measure was how good were the custom GRU in predicting the test dataset with respect to the default PyTorch module.

It is interesting to check, if the spline approximated activation functions converge to the same error as the real activation functions. I used three implementations of the GRU cell, one with real activation functions and using the PyTorch GRU module, one with the approximated activation functions and using the custom GRU module and a third one with the exact PyTorch activation functions but using the custom GRU module. The results are shown in the figures 8.16 and 8.17 below:

By calculating the last values of the Mean Squared Error of the predictions of the models, I get the results shown in table 8.1:

The approximated activation functions converge to the same error as the real activation functions fairly quickly and although the training set converges faster using the exact activation functions, trying to predict on the test set using the custom ones is actually converging faster at first but gets outperformed by the exact implementation in PyTorch. This is, of course an example specific result, and in a fairly simple benchmarking dataset but it could indicate that having a cutoff value before the actual asymptotic value of the activation function can be beneficial in some cases. Another interesting observation is that although the tanh is far better approximated by the spline than the sigmoid, the

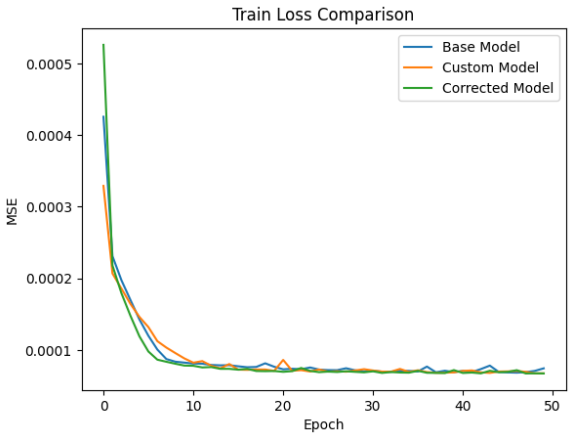


Figure 8.16: Mean Squared Error of the training set.

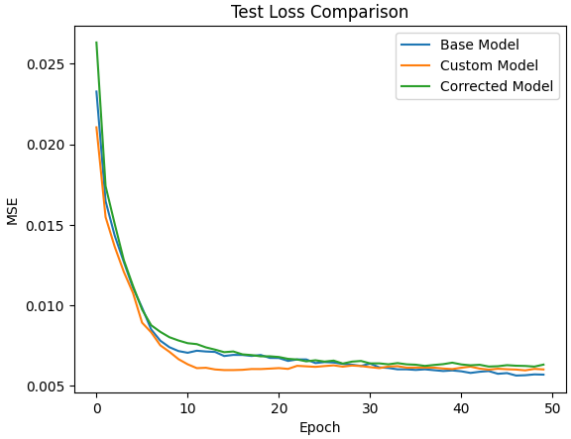


Figure 8.17: Mean Squared Error of predicting the test set.

Model	**Mean Squared Error (MSE)**
Base Model (PyTorch Module)	1.48
Custom Activation Functions with Custom Model	1.57
Exact Activation Functions with Custom Model	1.66

Table 8.1: How combinations of custom activation functions and custom GRU cell are perfoming with respect to the original PyTorch module.

model is fairly robust, and thus we can hypothesize that the model is not very sensitive to the exact values of the sigmoid.

8.5 Benchmarking the GRU in the existing implementation

Finally, by placing the GRU subgraph between the observations buffer and the Feed Forward Neural Network, I can measure the latency of the complete system figure 8.18. The final graph looks like this:

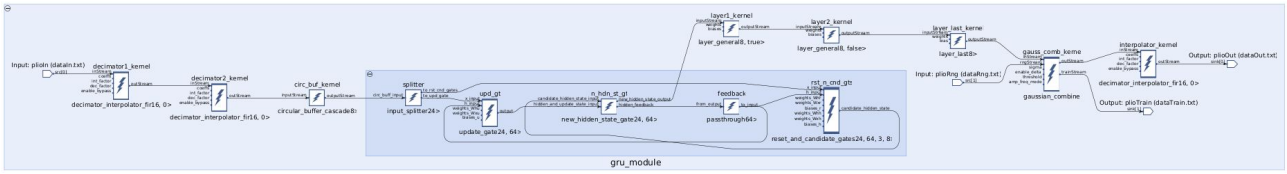


Figure 8.18: The complete system. The subgraph of the GRU is visible in the middle of the existing implementation.

The test comprised by 'dummy' inputs using numpy.arange function since I was not interested in validation but in just simple input-output time difference to measure latency. Once again, in order to relate the first of the inputs that relate to an output, we need to know how many inputs are needed. In this implementation we calculate the amount of inputs that correspond to an output by knowing that the decimation and interpolation factors are. Specifically, as also shown on the graph, two decimation kernels (named 'decimator') are employed after the inputs which will downsample $6 \times 6 = 36$ inputs down to 1 and insert it to the observations buffer. The interpolator is bypassed and therefore it has no effect. Finally we enstablish that for one output we need 36 inputs.

I generated 6000 inputs which are not exactly divisible by 36 on purpose in order to crash test the TLAST signal. Therefore, I expect $\frac{6000}{36} = 166.66 \times 2 \sim 333$ outputs, where the $\times 2$ comes from the fact that we are expecting two scalar outputs of amplitude and phase.

Next, I must also calculate the frequency in which the DAQ is producing samples. Since the actual

revolution frequency of the beam is 2.72 MHz and the frequency of the PL clock runs at 250 MHz, we can calculate the equivalent clock cycles as $\frac{250}{2.72} = 91.91 \sim 92$. Thus, we can set the scheduler to send one piece of data every 92 clock cycles in order to simulate a real application.

Finally, I can measure the latency as shown in the 8.19 below:

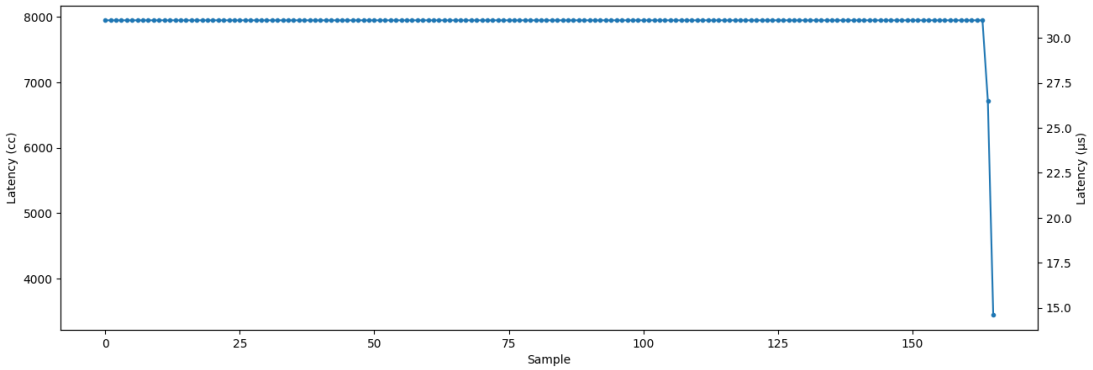


Figure 8.19: Latency of 6000 inputs and 333 outputs (considering only one of the two at a time since they output together). Mean latency is 7948 clock cycles, which are $\sim 31,8$ μs

The last output is the 'end computations' signal and the penultimate drop in latency only because the inputs are not aligned to 36 and thus the computations terminate early.

Note that we should distinguish latency from the ability to output the result of a computation. Given enough data to initiate the computation, the pipeline will always produce a result in finite time. In figure 8.19 the difference between consecutive outputs is 3312 cycles or 13.25 μs.

I also ran an unscheduled inputs test, which creates maximum backpressure on the pipeline, just to test the system. In this test, caching effects of the FIFOs in each tiles interface can be seen in 8.20 which induce a lot of latency on the system:

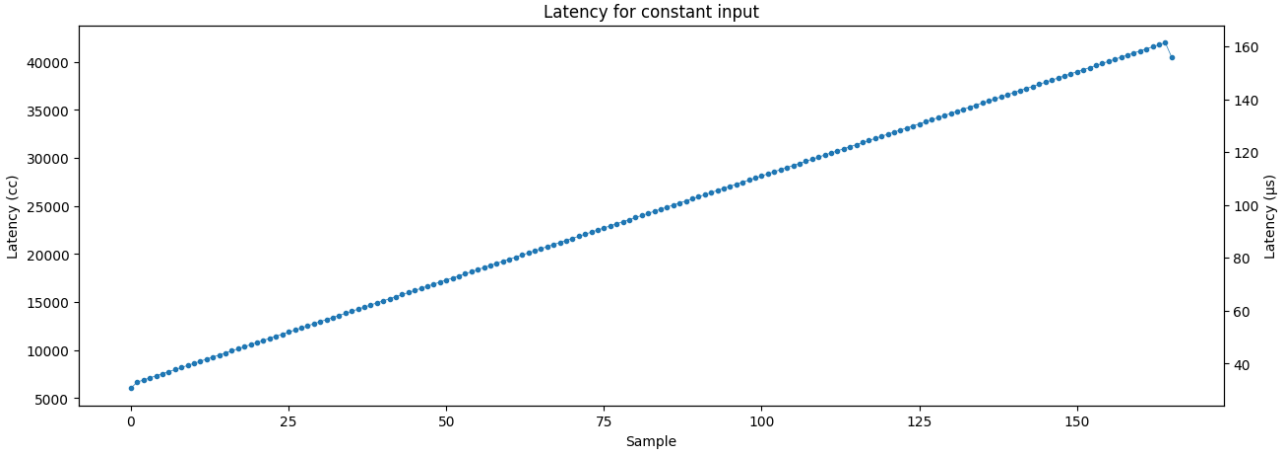


Figure 8.20: A constant flow of data destroys the latency constrains of the pipeline by accepting data faster than in can process them and produce an output. The difference keeps building up until eventually this caching effect will saturate all involving FIFOs and will hit a plateau. At this point, it is not possible to accept data anymore.

Here, the difference between outputs in this test is 1762 cycles or 7.05 μs. This means that this is the speed limit of this implementation (the fastest it can produce a result by leveraging the pipeline at its maximum).

Chapter 9

Conclusions

9.1 Overview and Final Remarks

One of the goals of this thesis was to gain hands-on experience with Versal and the development tools. During my limited stay at the Institute for Data Processing and Electronics (IPE) in the Karlsruhe Institute of Technology (KIT), I was able to learn a lot about the underlying hardware design, which is not trivial for a trained physicist. I benefited greatly from the accumulated experience, and thus I believe that the goal of the internship was achieved.

The custom benchmarking tool served both as an introduction to the design, development, and simulation tools, and as a more straightforward way to test the implementations on hardware. Understanding the design flow took a good deal of my internship time, as did developing the Python code for the benchmark. The hands-on experience acquired by accessing memory, running programs, and in general developing automated systems was invaluable. Concerning the final result, this tool could have real-world value since AMD Xilinx does not provide a hardware testing platform, but only simulation tools that are arguably hard to use and interpret.

The main contribution was a preliminary development of the Gated Recurrent Unit (GRU). The GRU has been implemented on FPGAs before, both from scratch and by using the HLS4ML [31] framework, but it has never been done (at least to my knowledge) in the AI Engine. I say preliminary because it can be further optimized, and an automated mechanism to split the operations into multiple kernels is needed. It was a gradual process of learning many intermediate steps and overcoming countless failures that led me to this result. In general, Recurrent Neural Networks are among the best possible non-linear function approximators and are especially efficient in temporal-dependent implementations. So the question of its efficiency is not in the model itself but mostly in the data inputs, hyperparameters, and size. The latency measured is not terribly far from the initial implementation, and it could target slow oscillations.

Unfortunately, I wasn't able to deploy the GRU in a live setting, but I was granted two days of testing on the KARA Synchrotron prior to the development of the GRU, which greatly enhanced my understanding of the research process.

Regarding beam control, we were not able to actually induce control over the microbunching instabilities. This is a challenging task on its own and an active research field. It has to do with how fast we are acting on the bunch, with what kind of modulation, and which machine we are using for this modulation. Most likely, we need a specific modulator just for this task as an harmonic cavity or a laser system.

Due to the sheer amount of technical knowledge required to develop a top-level solution from scratch to hardware implementation, many details that can be crucial were left out of the thesis. This thesis should be considered more as a summary of my internship experience rather than a complete guide to hardware development.

To conclude, I presented my internship work at a level that I deemed appropriate for someone who does not have hardware experience and may not be familiar with reinforcement learning. This is why many introductions were needed. I also tried to keep it minimal in definitions and mathematical formulas since the main task was a software engineering one. A comprehensive understanding of the GRU was provided, and some preliminary latency results were presented.

Concerning GRU Latency Results

The GRU latency was measured at 31.8 μs. This is around the same order of magnitude of the fast oscillations of the microbunching instability. As such the system represent a good starting point for the control experiments planned in the future. Latency is greatly affected by model size and the way the developer designs the system. The GRU implementation, although complete from a basic functionality aspect, can still bear multiple optimizations.

9.2 Future Work

Benchmarking Tool

In the future, this tool could be provided with a graphical user interface, more functions, and the ability to track where the script fails and perform latency analysis. It will not replace the simulation tools but will expand the testing capabilities of the developer.

GRU

Designing the model implementation from scratch every time is tedious, prone to errors, and non-optimizable. The subgraph implementation provides some modularity, but it is not enough. There is a need for a framework like HLS4ML [31], but targeting AI Engines. A framework like this should automatically calculate memory and computational needs to design an optimal pipeline using the AI Engine array. This would likely split the computations into multiple kernels for maximum parallelization and memory efficiency. Furthermore, the AI Engine eliminates the need to quantize the variational parameters of the models, which can be extremely useful for importing models from high-level frameworks like PyTorch or TensorFlow. Moreover, a 'block design' could also be implemented to connect kernels or top-level machine learning modules.

There are also modern realizations of Recurrent Neural Networks that try to parallelize computations as much as possible, or use quantized weights to be more light weight and faster. It would be extremely interesting to explore these implementations on hardware.

Another significant step forward would be to run backpropagation, and therefore model training, on the Versal board itself. This could be done either using the machine learning frameworks on the ARM Cortex-A72 CPU or, ideally, by storing the accumulated gradients in the AI Engines and performing gradient descent parameter updates there. This would be excellent for online training, and more advanced models could greatly benefit from a platform like this.

Beam Control

During a live deployment of the agent, there absolutely needs to be a tuning of the hyperparameters of the model and the reinforcement learning framework. Until now, this was done by hand. A Bayesian optimization scheme that monitors the model and updates the hyperparameters accordingly could be an elegant solution.

Furthermore, data acquired during beamtime could be greatly beneficial for analysis in order to enrich the input features of the model. The possible use of autoencoders could pretrain instances of the model.

Physics-Informed Models

Physics-informed networks (PINNs) are models based on neural networks that integrate physical laws, expressed as partial differential equations (PDEs), directly into the training process. Unlike traditional machine learning models, which learn patterns from data alone, PINNs incorporate the underlying physics governing a system, allowing them to generalize better. In practice, PINNs solve complex physics problems, such as fluid dynamics, electromagnetism, and quantum mechanics, by ensuring that the network's predictions adhere to the physical constraints, thus improving accuracy and interpretability.

The application of physics-informed networks at the edge, where computational resources are limited, holds significant promise. By embedding physical laws into the network, PINNs may be key to controlling physical phenomena (in contrast to complex systems).

Models that Exploit Parallelism

In this work, only a handful of tiles were actually used to run the data pipeline. This was also a design decision since more tiles means extra connections and thus clock cycles 'lost' in data transfer. But it is clear that a lot of computing potential was left out, also because of the fact that the GRU is not a parallelized computation, since the next iteration of the cell requires the previous hidden state.

It stands to reason then to explore models that can be easily parallelized, like the state-of-the-art Transformer model.

9.3 Beyond this Implementation

Controlling the microbunching instabilities may be the context of this work of running models on the Versal, but the true power of reinforcement learning lies in the fact that it is agnostic to the machine it tries to control (at least in this implementation). The agent's objective is to minimize an appropriate loss function. Therefore, the same principles of this implementation, or even the same model, could be applied to other machines.

Fusion

The closest example would be nuclear fusion in Tokamak machines, because these machines operate with plasma beams manipulated via magnetic fields. As fusion is achieved, the mixture of elements changes rapidly, and the beam destabilizes immediately. Being able to predict and control beam instabilities could be a significant milestone in achieving power extraction from fusion.

Quantum Devices

Another technological frontier is quantum computers. However, for the time being, they are noisy and of low fidelity due to unwanted interactions. Even though engineers go to great lengths to decouple them from every possible perturbation, it boils down to the fact that, in order to drive qubits using electromagnetic radiation, you will perturb other qubits in the vicinity as well. Implementations like the one in this work could also be used to create modulations of optimal control in quantum machines.

Bibliography

[1] Tobias Boltz. *Micro-Bunching Control at Electron Storage Rings with Reinforcement Learning.* Phd thesis, Karlsruher Instituts für Technologie (KIT), November 2021.

[2] S Y Lee. *Accelerator Physics.* WORLD SCIENTIFIC, 3rd edition, 2011.

[3] Marvin-Dennis Noll, Dima El Khechen, Johannes Leonard Steinmann, Erhard Huttel, Marcel Schuh, and A.-S. Müller. Longitudinal bunch diagnostic at the kara booster synchrotron. Vortrag gehalten auf 10th MT-ARD-ST3 Meeting (2022), Berlin, Deutschland, 7.–9. September 2022, 2022. 54.11.11; LK 01.

[4] MERIL Mapping of the European Research Infrastructure Landscape. Test facility and synchrotron radiation source (anka), 2019. Original image.

[5] M. Caselle, L.E. Ardila Perez, M. Balzer, A. Kopmann, L. Rota, M. Weber, M. Brosi, J. Steinmann, E. Bründermann, and A.-S. Müller. Kapture-2. a picosecond sampling system for individual thz pulses with high repetition rate. *Journal of Instrumentation*, 12(01):C01040, jan 2017.

[6] Miriam Brosi. *In-Depth Analysis of the Micro-Bunching Characteristics in Single and Multi-Bunch Operation at KARA.* PhD thesis, Karlsruher Institut für Technologie (KIT), 2020. 54.01.01; LK 01.

[7] Jan Kaiser, Chenran Xu, Annika Eichler, Andrea Santamaria Garcia, Oliver Stein, Erik Bründermann, Willi Kuropka, Hannes Dinter, Frank Mayet, Thomas Vinatier, Florian Burkart, and Holger Schlarb. Learning to do or learning while doing: Reinforcement learning and bayesian optimisation for online continuous tuning, 2023.

[8] https://deepanshut041.github.io. Interactions between actors and the environment, 2023. Original image.

[9] L. Scomparin, E. Blomley, T. Boltz, E. Bründermann, M. Caselle, T. Dritschler, A. Kopmann, A. Mochihashi, A.-S. Müller, A. Santamaria Garcia, P. Schreiber, J.L. Steinmann, and M. Weber. KINGFISHER: A Framework for Fast Machine Learning Inference for Autonomous Accelerator Systems. In *Proc. 11th Int. Beam Instrum. Conf. (IBIC'22)*, number 11 in International Beam Instrumentation Conference, pages 151–155. JACoW Publishing, Geneva, Switzerland, 12 2022.

[10] Hoang Giang, Tran Hoan, Pham Thanh, and Insoo Koo. Hybrid noma/oma-based dynamic power allocation scheme using deep reinforcement learning in 5g networks. *Applied Sciences*, 10:4236, 06 2020.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[12] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[13] Timo Dritschler. *High-Performance Commodity Data Acquisition Systems for Scientific Applications in the Terascale Era.* PhD thesis, Karlsruher Institut für Technologie (KIT), 2023. 54.12.02; LK 01.

[14] UG Pawara Siriwardhane. Von neumann architecture, 2020. Image cropped from the original article.

[15] Mark Stephenson, Siva Hari, Yunsup Lee, Eiman Ebrahimi, Daniel Johnson, David Nellans, Mike O'Connor, and Stephen Keckler. Flexible software profiling of gpu architectures. *ACM SIGARCH Computer Architecture News*, 43:185–197, 06 2015.

[16] Dheeraj Punia. Fpga design, architecture and applications, 2023. Image cropped from the original article.

[17] Haobo Ye. Accelerating convolutional neural networks: Exploring fpga-based architectures and challenges. *Journal of Physics: Conference Series*, 2786:012004, 06 2024.

[18] Vitis High-Level Synthesis User Guide (UG1399). Axi4-stream handshake, 2024. Original image.

[19] AMD Xiling. *AXI DMA LogiCORE IP Product Guide (PG021)*, 2024.

[20] AMD Xilinx. Network accelerator with versal™ premium series - versal adaptive soc implementation, 2023. Image cropped from the original article.

[21] Versal Adaptive SoC AI Engine Architecture Manual (AM009). Hierarchy of tiles in a ai engine array, 2023. Cropped from the original image.

[22] AMD Xiling. AI Engine: Meeting the Compute Demands of Next-Generation Applications. Ai engine architecture, 2023. Cropped from the original image.

[23] AMD Xilinx. Ai engine vector registers, 2023. Image cropped from the original article.

[24] AMD Xilinx. Ai engine accumulator registers, 2023. Image cropped from the original article.

[25] AMD Xilinx. Ai engine intrinsics user guide (aie) r2p22, 2022. Image cropped from the original article.

[26] AMD Xilinx. Ai engine vector unit, 2023. Image cropped from the original article.

[27] Luca Scomparin, Edmund Blomley, Tobias Boltz, Erik Bründermann, Michele Caselle, Timo Dritschler, Andreas Kopmann, Akira Mochihashi, Anke-Susanne Muller, Andrea Santamaría García, Patrick Schreiber, Johannes Steinmann, and Marc Weber. Preliminary results on the reinforcement learning-based control of the microbunching instability. In *Proc. IPAC'24*, number 15 in IPAC'24 - 15th International Particle Accelerator Conference, pages 1808–1811. JACoW Publishing, Geneva, Switzerland, 05 2024.

[28] Weija Wang, Michele Caselle, Tobias Boltz, Edmund Blomley, Miriam Brosi, Timo Dritschler, Andreas Ebersoldt, Andreas Kopmann, Andrea Santamaria Garcia, Patrick Schreiber, Erik Bründermann, Marc Weber, Anke-Susanne Müller, and Yangwang Fang. Accelerated deep reinforcement learning for fast feedback of beam dynamics at kara. *IEEE Transactions on Nuclear Science*, 68(8):1794–1800, 2021.

[29] Hozan Hamarashid. Modified long short-term memory and utilizing in building sequential model. *International Journal of Multidisciplinary and Current Research*, 05 2021.

[30] Zhang Aston, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning interactive deep learning book with code, math, and discussions, 2022. Image cropped from the original article.

[31] FastML Team. fastmachinelearning/hls4ml, 2023.