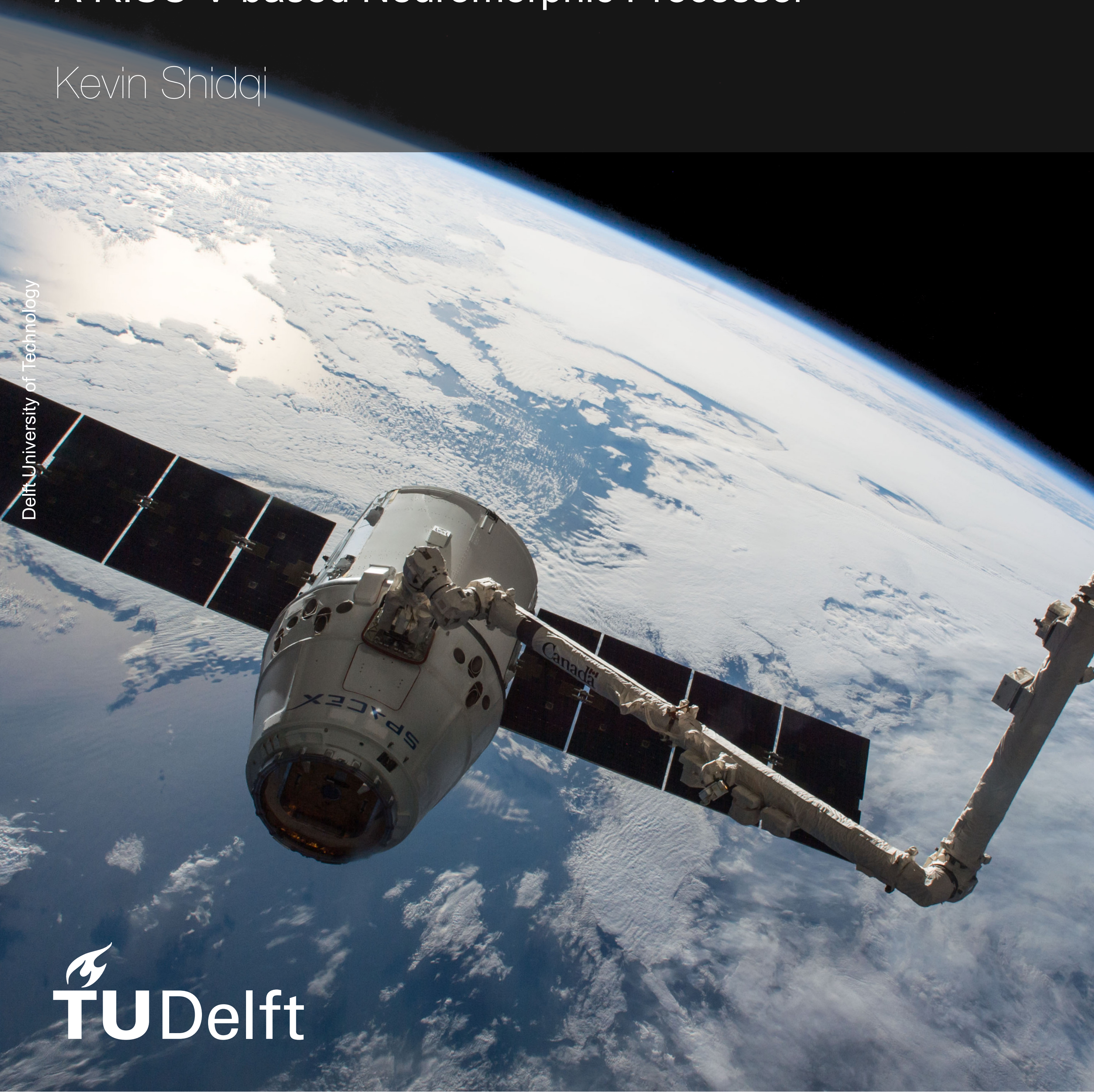
Benchmarking and Algorithm Optimization for SENeCA

A RISC-V-based Neuromorphic Processor

Kevin Shidqi



Delft University of Technology

TUDelft

Benchmarking and Algorithm Optimization for SENeCA

A RISC-V-based Neuromorphic Processor

by

Kevin Shidqi
Embedded Systems: Embedded Computer Architecture
Student Number: 5234948
Email: KevinShidqi@student.tudelft.nl

Supervisor: Professor Said Hamdioui
Duration: December 2021 - November 2022

Abstract

With recent breakthroughs in AI (Artificial Intelligence) technology, the impact of AI on society can be felt in various fields. The market for AI software, for example, reached a valuation of $62 billion in 2022. A growing number of new computer architectures specialized in running these AI software were also developed. At first they were run on conventional CPUs (Central Processing Unit) and GPUs (Graphical Processing Units), but then more specialized hardware emerged, such as the TPU (Tensor Processing Unit). However, since algorithms in these AI software are generally data-intensive, the power consumption became a problem. Therefore, as many of these algorithms were based on biological neural networks, there is a growing interest to develop hardware similarly based on principles found these networks as well to replicate their efficiency. This new architecture is known as neuromorphic architecture.

However, a new architecture does not come without challenges. As a nascent and fragmented field, neuromorphic computing in general lacks a standardized benchmarking suite or methodology. In other, more mature fields, benchmarks are a standard way of evaluating the performance of different designs objectively and fairly. This thesis aims to propose and demonstrate a benchmarking methodology and implementation flow for neuromorphic processors. This methodology aims to measure the important performance metrics for a neuromorphic processor, both on the small scale of individual synaptic operations, and the large scale of performing an actual workload. The chosen workload is a keyword spotting program based on a simple DNN architecture, which detects a specific phrase in an audio recording. This workload was chosen due to its potential application in an environment where energy is limited, such as an embedded device.

The neuromorphic processor that is the target of this benchmarking is SENeCA (short for Scalable Energy-efficient Neuromorphic Computer Architecture), a flexible and scalable design developed at IMEC The Netherlands. To implement the keyword spotting program on SENeCA, the keyword spotting program was rewritten and parsed. Since no physical chip implementation of SENeCA exists at the time of writing, the program was run on SENeCA using a HDL simulator. The execution time of the program is measured in detail, taking into account not only the total time, but also the time required to complete the specific stages of program. Afterwards, the power consumption of SENeCA during the execution of the program was measured using a power estimation software, both for the entire chip and its individual components. This is done both in average mode, obtaining the average power consumption over the total execution time, and in time-based mode, providing insight to the peak power and fluctuations over time. Then, the energy to solution is calculated using the execution time and power consumption. This process is done in multiple iterations, with a specific optimization done each iteration using SENeCA's accelerators. This provides insight into the impact of each optimization to power consumption and performance. Finally, a measurement of the energy consumption of SENeCA per individual synaptic operations is also done, allowing estimates of the energy consumption of future implementations.

The work in this thesis was supported by IMEC The Netherlands. Their cooperation is hereby gratefully acknowledged.

Contents

List of Figures

List of Tables

Preface

This M.Sc. thesis report concludes the amazing two years I spent in Delft as an Embedded Systems student. The subject of neuromorphic engineering sparked my interest the first time it was mentioned in a lecture, and during the work performed for this thesis, it has only grown. I hope that you, the reader, can also learn about it and AI in general while reading this thesis. While this thesis is long, more than 100 pages, I have tried my best to keep it as engaging to the reader as possible.

There are multiple people I would like to thank since it was their support that made most of the work for this thesis possible. First I would like to thank my supervisors, Prof. Said Hamdioui and Anteneh Gebregiorgis, for their advice and feedback during the few progress meetings that we had, and also for proofreading and giving suggestions to improve this thesis. I would also like to thank my supervisors at IMEC The Netherlands, Amirreza Yousefzadeh and Gert-Jan van Schaik, for your enourmous help during the implementation. I realize that this work is only a small part of the SENeCA project being developed at IMEC, but I hope that my small contribution can be of use.

Furthermore, I would like to thank my friends at DH06, especially Johanna, Bas, Boris, Niels, and Maddy, and my fellow Indonesians at PPI Delft, you made an academic year during the pandemic bearable. Also, I would like to thank my fellow thesis students at IMEC, Prithvish and Alexandra, for making the long hours in the library and office enjoyable. I would also like to thank my family on the other side of the world for their unconditional and unending love and support. Last but not least, I would like to thank Aina, for always being by my side.

Kevin Shidqi
Delft, November 2022

1 Introduction

1.1. Motivation

The market of Artificial Intelligence (AI) software is growing rapidly, reaching a valuation of $62 billion in 2022. That is an increase of around 21% compared to 2021 [38]. Meanwhile, the revenue from AI for enterprise applications has increased more than tenfold between 2017 and 2022, and is expected to continue to increase in 2022 and beyond [31], as shown in Figure 1.1.

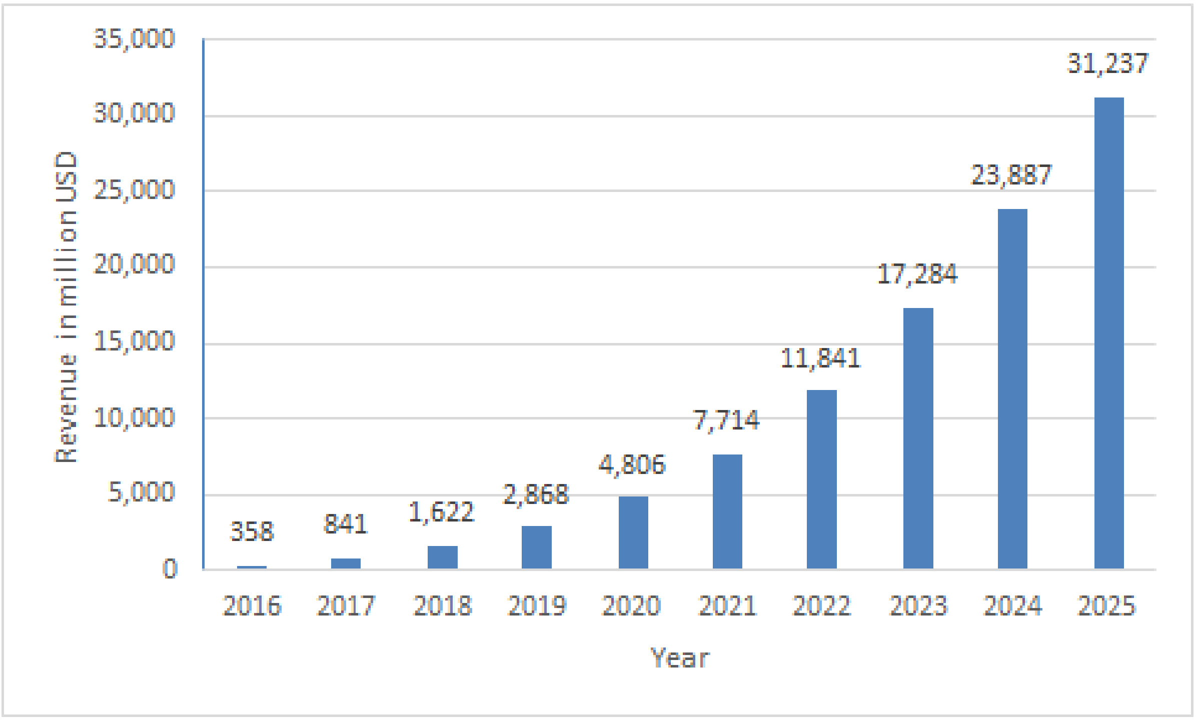


Figure 1.1: Revenues from the artificial intelligence for enterprise applications market worldwide, adapted from [31]

While the majority of the users are still the computer and software industries, the adoption of AI tools and methods is also being carried out in various other fields, as mentioned in a report by O'Reilly [65]. Examples of the potential application of AI in various aspects of our daily lives are listed in the following.

1. Healthcare: Several AI applications can perform as good as, or in some cases better than, clini-

cians in diagnostics [85]. In the future, AI applications may be used to improve the availability of diagnostic services and reduce their cost.

2. Business: Enterprise Cognitive Computing (ECC) is the use of AI to enhance business operation. This is achieved by using AI applications to perform repetitive tasks, improving efficiency [97].
3. Education: The usage of AI in education includes the automation of administrative tasks, the condensing of textbooks to useful exam preparation tools, and the development of an AI-based tutoring system [96].
4. Agriculture: The automatic detection of objects of interest, such as fruits, in orchards or farms using unmanned vehicles [69] and the monitoring of soil parameters [37] are two possible cases in which AI algorithms can be applied in agriculture, potentially helping farmers and agronomists with automation of labor-intensive tasks.
5. Network Security: The use of AI algorithms to observe internet traffic and detect unusual traffic to distinguish a security threat was demonstrated in [60].

The examples and numbers above show the growth and potential of AI applications. Many of algorithms used in these applications are based on information processing systems found in biological, mainly human, brains. These information systems are noted for their ability to perform tasks with remarkable efficiency in terms of energy.

The implementations of these algorithms on digital computers, such as GPUs (Graphical Processing Units), was proven to be possible [74]. The parallel structure of GPUs, originally developed to tackle the parallel nature of graphical applications, allow them to handle the computationally intensive nature of AI algorithms. However, these implementations still lack the energy efficiency of biological systems [70] upon which these algorithms were developed. The main cause of the energy problem with the implementation of AI algorithms on von Neumann computers (the most widely used architecture, including GPUs) is the so-called memory wall [80]. In von Neumann computers, the processing unit and the memory are separated, usually on different chips or different parts of a chip, and they are connected by a data bus. This causes the energy consumption of a memory access operation to be as much as 1,000 times higher than that of an arithmetic operation [48]. Since AI applications are also data-intensive, requiring a large number of read/write operations to the memory, the distance between the processor and the memory becomes a major bottleneck and cause of inefficiency in terms of energy [49]. Some of the applications of AI sofware take place in environments where energy for performing computations is limited (e.g. in embedded applications such as in [69]), worsening this problem.

To solve the problems above, in recent years, a new type of computer architecture has emerged that departs from the conventional von Neumann architecture. It specializes in data-intensive AI algorithms and aims to use the principles of biological neurons to replicate their energy efficiency. These new architectures are termed ”Neuromorphic processors”. Research on analog versions of neuromorphic processors was done as early as 1995 [28]. Currently, several prototype chips have already been published, such as Loihi[22], TrueNorth[71], and SpiNNaker[35]. Another chip, which is still in the developmental stage, is SENeCA [109]. SENeCA, named after the Roman philosopher, stands for Scalable Efficient Neuromorphic Computer Architecture. It is designed to be a flexible and scalable architecture that can accommodate future neural network architectures.

However, to accelerate the development of these neuromorphic processors, there needs to be a standardized process or suite to perform benchmarking [20]. A similar concept already exists in conventional CPU, such as Standard Performance Evaluation Corporation (SPEC), an American non-profit organization that aims to produce a standardized set of performance benchmarks for computers [91]. It is argued that one of the factors that drove the rapid development of CPUs was this benchmarking standard, and to truly unleash the potential of neuromorphic chips, a similar benchmarking suite or standard should be developed for neuromorphic architecture. Similarly, an effort to produce ”fair and useful benchmarks” for the field of machine learning is being made by MLPerf [67]. For neuromorphic computing to reach maturity as a field, a similarly standardized and systematic methodology or suite for benchmarking is surely needed.

1.2. Problem Statement

The goal of Neuromorphic Computing as a field is quite ambitious, namely, to decipher the secrets of the biological brain that allow the unparalleled efficiency and flexibility of brain-based computing. As the challenge is quite daunting, there is a need to focus on measuring quantitative metrics to prove real-world value by benchmarking, instead of open-ended exploration [20]. This might seem difficult, since the designs that have been published vary widely in their features and purpose [86]. Furthermore, there is no standardized language yet for neuromorphic processors at the time of writing, such as the C language that was used in SPECint benchmarks in the case of conventional CPUs.

In many fields, however, benchmarking has provided great value by allowing for a fair and useful comparison between architectures or algorithms. For example, in conventional von Neumann architectures, the SPECint [91] benchmarks were developed. For systems programming, Dhrystone was widely used [102]. More recently, the MLPerf benchmark suite was developed for machine learning algorithms [67]. A good benchmark serves to motivate researchers to solve a particular problem that can represent a wider class of useful problems, improving the real-world value of their field.

Benchmarks also allow designers to measure and make tradeoff decisions in their design. In particular, for a nascent field like neuromorphic computing, the impact of specific design choices might not be fully understood. Thus, it will be useful to perform a quantitative evaluation of the relative features, flexibility, and performance of existing platforms to help future designers understand the advantages and disadvantages of the design choices in those platforms and their impact on performance. This will lead to better designs in the future, helping to unleash the potential of neuromorphic computing to accelerate the applications of AI software to solve real-world problems.

As the goal of neuromorphic processors is to efficiently process neural networks, the focus of these hypothetical benchmarks should be on measuring performance and energy efficiency. Furthermore, a proper benchmarking suite would measure these metrics not only when running a full application but also when executing simple neural operations. This study aims to perform a systematic benchmarking process using one of the applications mentioned in [20] on SENeCA. The most important metrics to be measured, since the goal of SENeCA and neuromorphic processors as a whole, is to execute neural network operations in an energy-efficient manner, are energy-to solution and inference time. Other than that, peak and average power consumption, as well as chip area will also be measured.

1.3. State of the Art

This section will explore the published research on the topic of neuromorphic benchmarking. Most of the published architectures have some kind of demonstration section in which a certain algorithm is run on the architecture. The publication of Loihi [22] has a section which evaluates Loihi running a locally competitive spiking algorithm (LCA). Tianjic [26], meanwhile, was benchmarked with an unmanned bicycle controller with multiple ANN / SNNs. TrueNorth [71], had a framework built specifically for it to implement CNNs. Another design, MorphIC [33], was tested with the MNIST image classification database. The algorithms that were chosen to be tested on these architectures are mainly well suited to the architecture. While this is necessary to highlight the performance of these chips, the lack of a standard program or benchmarking methodology makes it difficult to perform a fair and objective comparison.

Several studies have been published that focused on developing a benchmarking suite. Ostrau, et al., developed a framework to develop benchmarks for neuromorphic processors named SNABSuite [76], based on the Cypress framework [93]. They used the framework to perform benchmarking with several programs, including a Sudoku solver and a simple program to measure the frequency of neuron firing on the four supported platforms [77]. The four supported platforms are BrainScaleS [82], Spikey [83], NEST [39], and SpiNNaker [35].They measured important metrics such as energy, accuracy, and time to solution.

Another framework for developing software for neuromorphic architectures, Nengo [11], was also used for benchmarking [10]. Nengo has several backend platforms to generate SNN-based code to run of various platform, and one backend, SpiNNaker, was run in this benchmarking study on the eponymous hardware, while four other backend platforms were run on more conventional hardware. Five different applications were used for benchmarking, all of them related to simulating large-scale brain models. The speed and accuracy while running these five applications were measured.

Other published studies focused more on using a application specific to other fields to demonstrate the capabilities of neuromorphic processors through benchmarking. A benchmark based on hand gesture recognition, for example, was performed by Ceolini, et. al. [17]. This study uses event-based camera input and signals from a sensor capable of measuring electrical signals from the forerarm muscle to detect hand gestures performed by humans. The neuromorphic chips used were Loihi [22] and a combination of ODIN [34] and MorphIC [33]. They were compared against a baseline implementation of more traditional machine learning based algorithm running on an NVidia Jetson hardware.

A benchmarking of neuromorphic processors using closed-loop applications, where the output of the processor is likely to affect its future input, was performed by Stewart, et. al., in 2015 [92]. The application is a relatively simple adaptive control program of a motor, where a neural network based algorithm is used by the program to improve its response in presence of disturbances that is randomized every time the benchmark is run. Instead of using a physical robot, they used a method known as Minimal Simulation [53] to introduce the variation in disturbances that is likely present in a physical system. They compared the performance of a neuromorphic processor (SpiNNaker) against a CPU and a GPU.

Instead of neural-network based applications, the capabilites of neuromorphic processors to perform more conventional, Message Parsing Interface (MPI) based parallel processing programs were tested in a benchmarking study by Urgese, et. al. [99], using a program for DNA sequence matching that is based on a text searching algorithm [61]. SpiNNaker was also used in this study, and it was compared against a server with two conventional CPUs.

Other than the publications mentioned above, there have also been several publications that aim to perform comparative studies on different platforms, both on neuromorphic architectures or other accelerators. In 2019, a study was conducted to compare the performance of Loihi with other more traditional architectures, on which this project is largely based [14]. The project used a keyword spotting algorithm based on a DNN architecture to measure the performance of Loihi compared to a CPU, GPU, and other accelerators such as NVidia Jetson and Movidius. Another similar study was conducted in 2020, pitting Loihi against SpiNNaker to compare their performance [107]. A more comprehensive discussion of these benchmarking studies is presented in Chapter 2.

Although evidently some progress has been made in the benchmarking of neuromorphic processors, the publications mentioned above vary in terms of metrics that were measured and the methodology used to measure them. Moreover, most of the comparative studies done focused more on comparing a neuromorphic architecture against a more conventional one, such as [99], [92], and [14]. Others that perform comparisons on two or more neuromorphic processors, such as [17] and [77] do not focus on going into detail to reveal which aspect or design choice of the benchmarked neuromorphic architecture gave the most impact in performance. As argued by Davies [20], one of the benefits of benchmarking is obtaining knowledge on the effects of design choices on performance, which is especially relevant in an emerging field such as neuromorphic computing. In general, we identified several areas in which improvements could be made:

- Since SENeCA is a new neuromorphic architecture, no benchmarking or comparative studies have been performed on it. Applying one of the benchmarks mentioned above on it will yield more insight, providing value to both the developers of SENeCA and future researchers.
- Most benchmarking studies mentioned above only have one/two workloads that represent real-world problems. While this is not an issue by itself, a real-world workload combined with a workload that benchmarks individual synaptic operations can provide an way of quantitatively comparing the fine-grained performance of different designs. This is similar to micro-benchmarks, which have been used to measure the fine-grained performance of GPUs [98] and network processors [9].

- No benchmarking publication thus far goes into the detail of measuring the power/energy consumption of individual components of the neuromorphic processor, instead they only the overall power/energy consumption. Knowing the power/energy consumption of individual components can reveal rooms for improvement in the design.

This study aims to perform benchmarking on SENeCA using an already existing benchmarking program that involves several iterations, with each iteration having a new optimization that involves a component in the architecture, to give better insight into the effect of the design choices present in the architecture on performance. Also, a secondary workload that benchmarks individual operations by SENeCA's neural processors will also be included. To do this, we will build upon the existing research by adapting the program used in [14] for SENeCA [109], instead of developing an entirely new benchmark. Regarding the metrics to be measured, we will base these upon [77], [17], and [14], which measure power consumption, time-to-solution, accuracy, and energy per inference, because they are more comprehensive than the measured metrics in [11], [92], and [99].

1.4. Contribution

The main goal of this thesis is to further improve the knowledge about neuromorphic architecture and neuromorphic benchmarking. The main contributions of this thesis are summarized in the following.

- **A new benchmarking methodology for neuromorphic processors**: As mentioned by Davies [20], there is a strong need for standardized benchmarking suites for neuromorphic processors, and that is where we hope this thesis can contribute. More specifically, the multiple iteration approach, with each iteration having a specific optimization, in the methodology used here can give insight into the design of individual components and the effect of these deisgn choices on the performance. Furthermore, the inclusion of a micro-benchmark workload to benchmark individual synaptic operations can give more insight into the fine-grained performance. Although we do not claim to have a fully developed benchmarking suite, we hope that the methodology used here can serve as a reference for future researchers.
- **Application of the new methodology on SENeCA:** SENeCA [109] is a neuromorphic architecture that is designed to be flexible enough to accommodate future neural network designs. As it will be made open source to researchers with an academic background, we hope that by performing benchmarking on it, the design choices of SENeCA and their impact on performance can be better understood, such that other designs can build upon it.
- **A new benchmarking implementation flow :** As the aim of neuromorphic processors is to run an application based on neural networks in an energy efficient manner, the power-aware implementation flow can be of reference for future developers.

To summarize, this thesis proposes a new benchmarking and application development flow for neuromorphic processors, and implements this flow on one design, SENeCA.

1.5. Thesis Organization

This thesis is divided into 7 chapters and an appendix. The organization of this thesis is summarized as follows:

- **Chapter 1:** this chapter will present the motivation, state-of-the-art analysis, problem statement, as well as the contribution of this research.
- **Chapter 2:** this chapter describes the necessary background information, mainly related to the field of neuromorphic processors. First, a brief history of artificial intelligence and computer architecture is presented, which leads to the motivation for neuromorphic computing. Then the basic

principles of neuromorphic computing are described, as well as several existing prototypes. Finally, an exploration of the literature related to the benchmarking of neuromorphic processors is presented.

- **Chapter 3:** this chapter describes in detail the architecture of the prototype neuromorphic processor that is the target of this research, SENeCA. First, an introduction to SENeCA is presented, along with its design principles and purpose. Then, a breakdown of SENeCA and its components is described, with a detailed explanation for each component. This chapter also includes a comparison of SENeCA with other existing prototype neuromorphic processors.
- **Chapter 4:** this chapter explains the design and implementation of the benchmarking software. It begins with a description of the neural network model and architecture upon which the software is based. Then, two sections will detail how this model is implemented in a PC and SENeCA, respectively. The section describing the implementation on SENeCA is divided into 2 versions of the software.
- **Chapter 5:** this chapter explains the experimental setup and methodology to measure the required performance metrics. Furthermore, this methodology will be implemented to the first 2 versions described in the previous chapter, and the results of those experiments will be presented. Furthermore, a section detailing an experiment to measure individual operations performed in SENeCA is also included.
- **Chapter 6:** this chapter describes the optimization process to obtain a faster and more energy-efficient software implementation. There are five versions in total, each obtained by analyzing the previous version to find room for improvement. For every version, an explanation of the optimization technique and implementation, performance measurement results, and power consumption measurement results are described. This chapter also includes a section dedicated to the measurement of accuracy of the SENeCA implementation.
- **Chapter 7:** in the final chapter, a summary and ideas about future work are described.

2
Neuromorphic Architectures and Benchmarking

This chapter introduces the necessary background information and dives more deeply into the literature currently available on neuromorphic architectures and benchmarking. First, Section 2.1 introduces Artificial Neural Networks (ANN). Section 2.2 will elaborate on why standard CPUs were limited for ANN operations. Section 2.3 will explain other architectures used for ANN operations before neuromorphic chips. Next, Section 2.4 will explain the design principles of neuromorphic architecture. Afterwards, Section 2.5 will dive into the available literature on neuromorphic chips. Finally, Section 2.6 will dive into the literature on bencmarking neuromorphic processors.

2.1. Artificial Neural Network

In the early days of artificial intelligence (AI), a field developed to realize man's dream of a machine that can think, problems that were difficult for humans but straightforward for computers were rapidly tackled. These problems could be described formally by mathematical rules without much difficulty. The real challenge came with problems that were difficult to describe but easy for humans to solve. These are the problems that we solve intuitively that feel automatic to us. These problems include recognizing faces from photographs and recognizing spoken words from an audio recording [40].

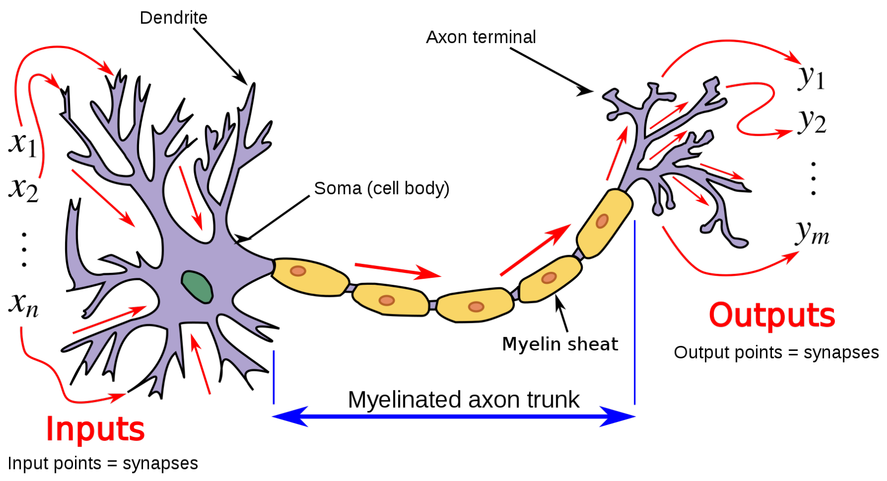


Figure 2.1: An illustration of a biological neuron [29]

The multilayer perceptron, another name for the Artificial Neural Network (ANN), is a solution for these more intuitive problems, as conventional algorithms had little success. This solution is based on the architecture of the human brain [40]. In general, ANNs consist of artificial neurons, which are based on the biological neuron cell found in human brains. Figure 2.1 depicts a biological neuron, also known as a nerve cell. Put simply, the neuron consists of the dendrites, the cell body (soma), the axon trunk, and the axon's terminals. The dendrites receive input signals from other neurons. The dendrites then transmits these signals to the soma, with ”adjustments”. These adjustments depend on the dendrite's ”weight” values, and depending on these values, dendrites can either amplify or diminish the signals sent to the soma. The soma then performs a summation of the signals from the dendrites. Every time the summation value in the soma exceeds a certain value, it emits a pulse signal and then sends it further to the axon. The axon's synaptic terminals, which are connected to other neuron's dendrites, pass the signal on to other neurons, repeating the process [73].

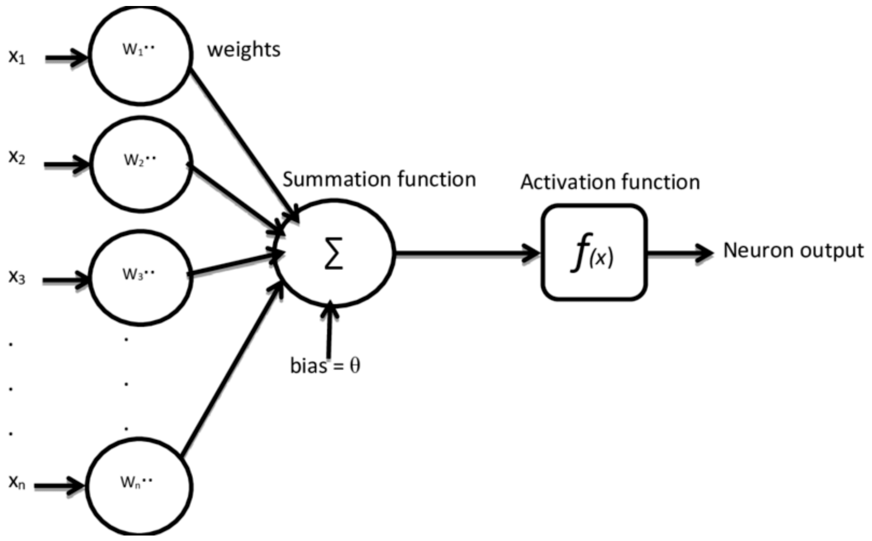


Figure 2.2: An illustration of an artificial neuron [106]

The artificial neuron is a mathematical model of the biological neuron [6]. Figure 2.2 depicts the model. In this analogy, the dendrites can be seen as the input of the artificial neuron. Each dendrite has a ”weight” value, and the input of that dendrite is multiplied by its weight value. This models the amplifying or diminishing action done by biological dendrites. The soma can be seen as the node, where the summation of the weighted input values from the dendrites takes place. After that, a bias term is added to the sum, and then it is passed through a non-linear activation function, producing an output. These are analogous to the threshold value of the soma. The output of the neuron, analogous the axon, sends the output value to other neurons [40].

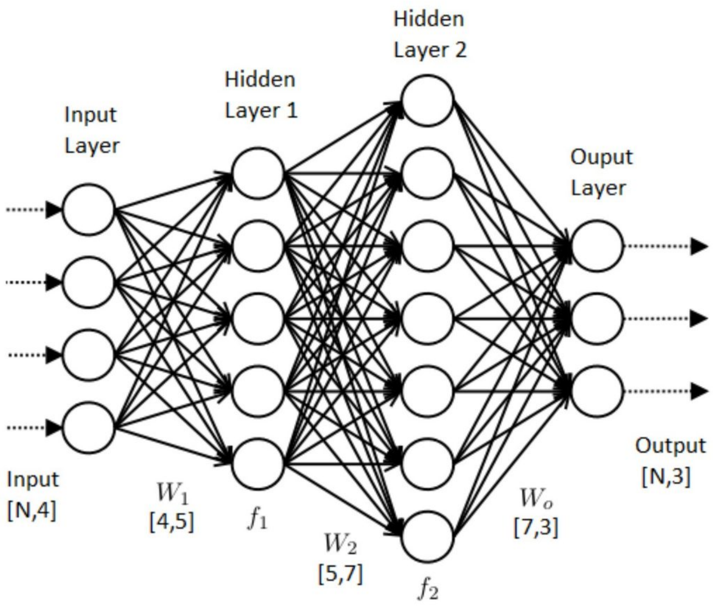


Figure 2.3: An illustration of a simple ANN [2]

An ANN is a network consisting of artificial neurons described above. Since the artificial neurons have a node and connections with other neurons, the network can be represented with a graph, with the neurons being the nodes and the connections being the edges. Figure 2.3 illustrates an example ANN. Each ball is a neuron, and each arrow represents a connection between the neurons. Here, the neurons are divided into four layers. Here, a neuron in a hidden layer receives inputs from all neurons in preceding layer, and sends its output to all neurons in the following layer. Since the data flows in a single direction, these type of ANNs are called feedforward networks. This is known as a Fully-connected, or dense, network [8]. The number of neurons and the layer configuration can vary based on the application.

As explained above, each neuron in a layer has connections to all neurons in the preceding layer. Also, each connection has a weight value assigned to it. In Figure 2.3, for example, each neuron in layer f1 (Hidden Layer 1) receives inputs from the four Input Layer neurons. Therefore, each neuron of layer f1 has four weight values. Since there are five neurons in layer f1, we can represent the 20 (four times five) weight values with a 4X5 matrix, named $W1$. Each layer has its own weight matrix, whose size depends on the number of neurons in that layer and the preceding layer. The weight matrices for the next layers are termed $W2$ and Wo. Also, each neuron has a bias term which is added to the sum of the inputs. The bias terms of neurons of a layer can be represented by a vector b, whose number of elements correspond to the number of neurons in the layer. Likewise, the inputs and outputs of a layer can also be represented by vectors, here termed x and a.

The relationship of the input vector, weight matrix, bias vector, and output vector of a layer L is defined by a function [40]. Typically, it takes the form of Equation 2.1.

$$Output_L = A_j = F(\sum_{i=0}^{n} W_{ij}.X_i + b_j) \tag{2.1}$$

F is the activation function, A_j is the output vector, X_i is the input vector, W_{ij} is the weight matrix and b_j is the bias vector of that layer, with $j = (0,..,m)$ and $i = (0,..,n)$ where m and n are the number of neurons of layer L and the preceding layer, respectively. In other words, to obtain the output, a weighted sum of all inputs is calculated. Then, a bias term is added to the weighted sum and an activation function is applied. Various activation functions are used, such as the Heaviside function (Equation 2.2) and the Rectified Linear Unit (ReLU) (Equation 2.3) [42].

$$F(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{2.2}$$

$$F(x) = max(0, x) \tag{2.3}$$

In essence, the objective of the neural network (NN) is to approximate a certain function. The multilayer feedforward ANNs are proven to be universal approximators [47]. An example of the application of ANN is described in [13], where images of handwritten digits are classified according to the digit in the image. In this case, the input is the pixel values of the images, while the output is a vector of probabilities for each number (0-9). To do this, the parameters of the neural network (weight values, bias values) are adjusted based on the learning process [40]. There are various forms of learning, generally divided into supervised and unsupervised learning. Unsupervised learning attempts to cluster unlabeled datasets by discovering hidden patterns in the data [45]. In supervised learning, the NN is presented with a labeled dataset, also known as the training data. During the learning process, the dataset is fed as input to the NN, and the output of the NN is compared against the correct output (the label), which produces an error value. This value is used to adjust the weights in such a way as to minimize the error value. A common method used the adjust the connection weights is backpropagation using stochastic gradient descent [84]. When this process is run repeatedly, the error rate may decrease. After the learning process in complete, the NN can be fed with actual data (also known as the test data) to evaluate its performance. This is known as the inference process [40].

From Equation 2.1, we can see that the majority of the arithmetic operations are done in the $\sum_{i=0}^{n} W_{ij}.X_i$ summation, which is basically a matrix multiplication. In ANNs that are designed to approximate complex functions, the number of neurons may be very large. For example, AlexNet, a neural network used to classify images of the ImageNet database, has over 650,000 neurons and 6 million parameters [63].

A subset of these parameters have to be accessed every time calculations take place, making it a very data-intensive application.

2.2. Limits of Traditional CPU

In computers that are based on the von Neumann architecture, the memory and the computing elements are separated from each other [80]. The computing element is a processor that executes sequences of machine instructions, such as arithmetic and logic operations. The memory is an element that can store the data. Typically, a Random Access Memory (RAM) is the type of memory used to store working data and machine instruction.

As argued in [48], the amount of energy required to read or write data to the RAM can be 1,000 times higher than the energy required to execute one computing operation. In addition, the maximum throughput at which data from the RAM can be transferred to the processor is usually limited by the width of the data bus (that connects the processor to the RAM), and much lower than the rate at which the processor can process the data. This results in latency and processor downtime while it waits for the data to arrive. This makes the bus that connects the processor and the RAM a significant bottleneck. This phenomenon is known as the Memory Wall. Many innovations in computer architecture have been developed to solve this bottleneck problem [80].

One solution to this problem is the introduction of a memory element closer to the processing unit. Modern processors usually have several layers of memory accessible to it, ranging from registers (small memory cells that are quickly accessible) to several layers of caches, to the RAM which is usually located on a separate chip [44]. Memory that is located close to the processor can be accessed quickly with little energy, but the size of the memory is limited. Accessing data located in the off-chip RAM, for example, requires a relatively large amount of time and energy compared to accessing data in the L1 cache. Figure 2.4 illustrates this point.

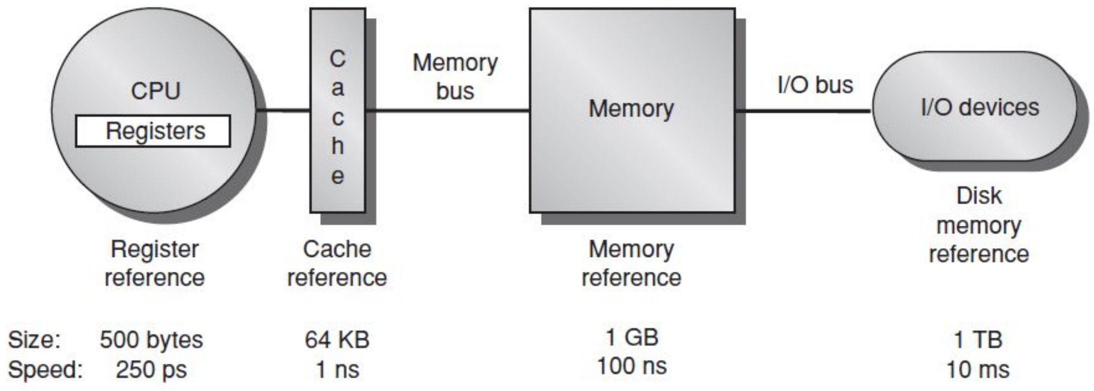


Figure 2.4: Memory hierarchy of the von Neumann architecture [44]

In architectures such as the one illustrated by 2.4, the weight matrix of ANNs described in Section 2.1 will most likely be stored in the RAM, since for most networks the weight matrix will be too large to store in the memory closer to the processor. Thus, the processor will often need to perform a read operation on the RAM to obtain the required weights or inputs (since the input vector is also likely to be stored in the RAM). As mentioned above, the data bus will limit the speed at which the operations can be executed. Furthermore, the number of weights increases exponentially depending on the number of neurons, inputs, and layers. This means that for larger networks, the time required to transfer the data will dominate the execution time [51].

Recent architectures, such as the AMD 3D V cache [104], attempt to solve this problem by introducing a larger cache. These caches, which can comprise around 40% of the chip area, can provide tens of megabytes of fast memory. Depending on the size of the network, this might allow the entire weight matrix of an ANN to be stored in the cache, allowing faster execution. Strategies to decrease the execution time for CPUs, such as speculative execution, branch prediction, and others mentioned in [80], are not applicable for ANNs, since the execution order is known beforehand. Other solutions depart from the traditional CPU architecture, such as the Graphics Processing Unit (GPU) and Tensor Processing Unit (TPU) architectures.

2.3. GPU and TPU

The GPU was initially developed to handle computer graphics [80]. It provides multiple streaming multiprocessors. These processors are smaller and slower than a typical CPU, but since there are several of them, calculations of multiple neurons can be executed simultaneously. Also, these streaming multiprocessors have large register files that can be used to store data from various contexts. If a single context currently has an operation that has a large latency, such as a memory read operation, the processor can execute instructions from other contexts instead. Regarding memory throughput, manufacturers such as NVIDIA began adding High Bandwidth Memory (HBM), bringing the memory throughput to 1.5 TB/s in the A100 architecture, released in 2020 [4].

A more recent architecture that is more specialized in neural network operations is the TPU, introduced by Google in 2015 [55]. This architecture attempts to solve the memory bandwidth problem by using systolic arrays in combination with software-controlled memory. At its heart lies a computational device that consists of 256x256 8-bit multiply-and-accumulate computational units. This is shown in Figure 2.5 as the Matrix Multiply Unit. Each unit has a memory that can store a weight value and is connected to four other units in a two-dimensional matrix. It performs two operations. First, it multiplies the input received from the unit above by the weight and adds that result to the number received from the unit to the left. Then it sends the input received from above to the unit below it without changing it and passes the sum obtained previously to the right. Essentially, it performs matrix multiplication in a pipeline, which is, as mentioned previously, the most time-consuming operation of an ANN. Also, this concept aims to minimize the number of operations that involve the memory, especially the storage and retrieval of intermediate results, since these are mostly stored in the memory of the MAC processors themselves.

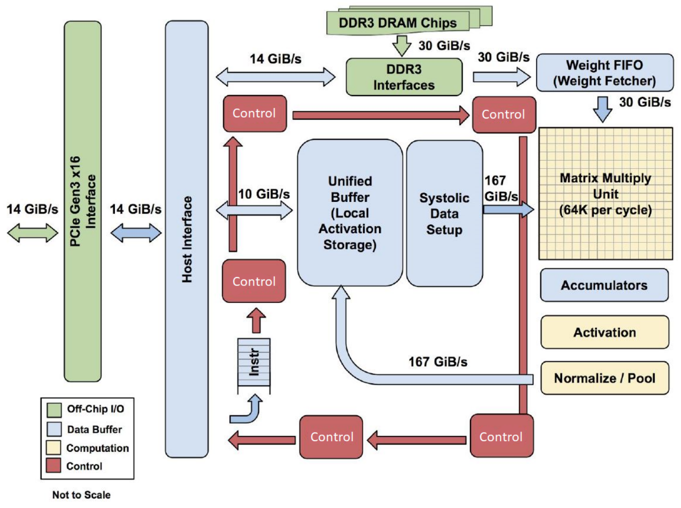


Figure 2.5: TPU Block Diagram [55]

In 2017, a performance test was published that compared the TPU architecture with other architectures mentioned above (CPU and GPU) [57]. The CPU and GPU were the Haswell (HSW) and K80 architectures, respectively. The study used six DNN applications run on three architectures. The study findings can be summarized in Figure 2.6. This uses the Roofline Performance model [103]. The performance of a specific architecture (in TeraOps/sec) is plotted against the operational intensity (in Ops/weight byte). Without enough operational intensity, the programs will be memory-bound, represented by the slanted part of the roofline. The flat part of the roofline represents the computation-bound area, where the processor speed becomes the bottleneck. As can be seen in the graph, the performance of the applications is quite close to the roofline for the TPU and lower for the CPU and GPU.

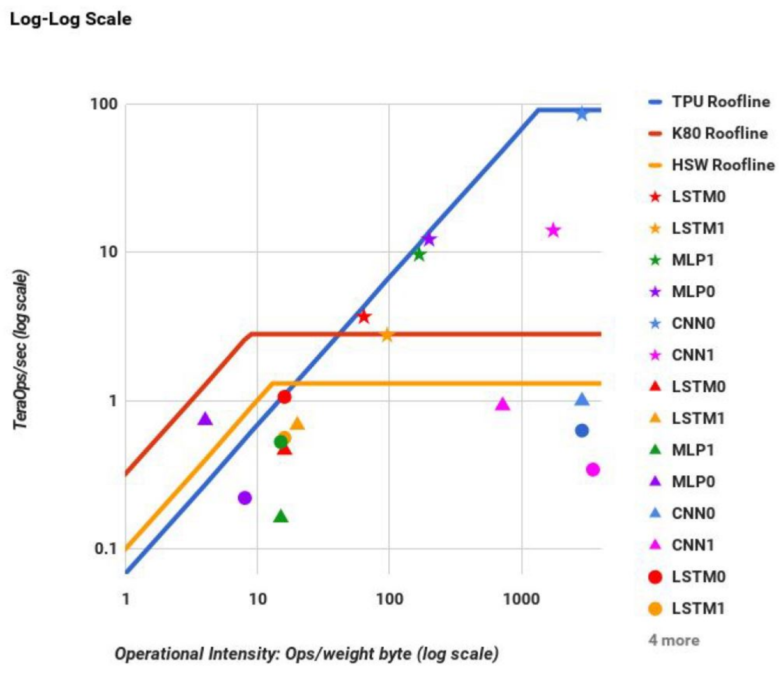


Figure 2.6: Roofline performance model of TPU (stars), GPU (triangles), and CPU (circles) [57]

The architecture of the TPU was shown to be more efficient in running DNN programs than the GPU and the CPU architectures [56]. Compared to the GPU, it was 15 times faster, while it was 29 times faster when the power consumptions of both are taken into account. Against the CPU, the numbers were 29 and 83, respectively. Several factors enable the TPU to have such a performance advantage:

1. The 2D Matrix Multiplication Unit of the TPU is more suited to matrix multiplication operations compared to the 1D multiplication units of the CPU and GPU.
2. The applications run on the TPU use 8-bit integers instead of 32-bit floating points.
3. The 2D arrangement of the processing units of the MMU allows for systolic arrays, reducing register access and energy consumption.
4. The TPU has only one thread, instead of the 13 threads of the GPU and 18 threads of the CPU, enabling it to save energy.

The TPU and GPU architectures have shown that they can run neural network programs faster and more energy-efficient compared to the traditional CPUs. However, these architectures are mainly designed for large-scale computing. For example, the Thermal Design Power (TDP) of the TPU in [55] is 75 W, while that of the K80 GPU architecture is 150 W. While that may be acceptable for computers in offices or datacenters, the power consumption may be too high for other situations, such as Edge AI or embedded applications. Thus, another type of architecture, termed ”Neuromorphic Architecture”, tries to fill this gap by focusing more on energy efficiency [1].

2.4. Principles of Neuromorphic Architectures

Information processing systems found in animals are very different from von Neumann architectures that are used in traditional computers [40]. Although these computers excel at performing operations such as arithmetic calculations, they are much less efficient in tackling problems whose input data are ill-conditioned and whose computation can be relatively specified. These problems include fields such as voice recognition, image processing, and others that have been influential in recent years. For tackling these kinds of problems, biological systems are orders of magnitude more efficient compared to traditional digital computers in terms of energy [70].

Despite the advances in architecture mentioned above, there are still gaps between the efficiency of animal brains and digital computers. Thus, the application of other design principles present in the brain structure can also possibly lead to further improvements. In recent years, this has led to the birth of neuromorphic computing.

The term neuromorphic engineering is a concept that was first developed by Carver Mead [70], and it was used to describe the use of VLSI (Very Large Scale Integration) systems containing electronic analog circuits to mimic neuro-biological architectures present in the nervous system [1]. More recently, the term neuromorphic architectures is generally used to describe analog, digital, and software systems that implement several models of neural systems [1]. It takes inspiration from the brain to develop energy-efficient circuits and systems for information processing. Several properties of neuromorphic architectures have been identified and explained by Ivanov et al., which appear to be useful in creating computer systems that can solve real-life problems [51]. Also, a taxonomy of neuromorphic architecture was presented by Bose, et al., in [15]. These principles differentiate neuromorphic architectures from the others.

1. Usage of non-von Neumann architecture, memory and computing are interspersed
2. Impulse nature of information transmission, low overhead for signal transmission
3. Sparsity of data streams, event-driven signal processing

An explanation of each principle is presented in the following.

2.4.1. Usage of Non-von Neumann Architecture

When performing neural network operations on a CPU, one core (or several cores, depending on the exact processor used) models a large number of neurons, sequentially switching context between them [51]. This creates a significant time and energy overhead to read/write the neuron state values or weights back and forth to the memory. A biological neuron, on the other hand, is simultaneously a device capable of storing its state and weights (by its membrane potential and strength of synaptic connections), and a computing device. This approach is free from the traditional von Neumann memory wall, rooted in physical separation of the memory and the processor. A possible implementation in silicon computers is to have each core of a neuromorphic processor model a single neuron only [50]. In digital implementations, it is possible to have several neurons modeled by a single core with limited context switching.

2.4.2. Low Overhead of Information Transmission

GPUs, while more efficient at performing neural network operations than a CPU, is still not optimally suited when energy consumption is factored in [16]. One reason is that in GPUs, the shared memory, which is a large (relative to the memory of the streaming multiprocessors) memory block able to be accessed by all the multiprocessors, is usually used for data transmission between neurons [74]. In biological systems, the information transmission occurs differently.

As explained in Section 2.1, a biological neuron generates electrical pulses that travel down its axon to communicate with other neurons. Sensory neurons, for example, are spiking neurons, and they change the temporal pattern of their electrical pulses depending on the external stimuli (light, sound, etc.) [1]. A sequence of pulses is also known as a spike train, and these spikes convey information based on their firing rate. Other neurons, such as specialized graded potential neurons, can communicate through spikes containing graded potentials [87]. These neurons have the advantage of higher information rates, because they are capable of encoding more states in a single spike than the spiking neurons. Both of these communication schemes are used in neuromorphic processors. For example, Loihi uses the rate coding approach [22], while SENeCA uses the graded potential spikes approach [109].

2.4.3. Sparsity of Data Streams

Studies of the brain show that only a small part of the brain, around 10%, is active at any time simultaneously [88]. This sparsity may be one of the reasons why the brain is much more energy efficient compared to a digital computer that runs a neural network program. It is very different from the execution of the inference phase of a classical ANN program, where every neuron is involved in calculations. Several factors explain this difference.

First, there are several cases in which subsequent inputs are quite similar to each other. An example is a Dynamic Vision Sensor (DVS), containing a computer vision program designed to detect an object that receives input from a stationary camera [90]. This allows traffic to be drastically decreased by transmitting only the differences between frames, instead of information from every pixel all the time. This is called temporal sparsity.

The second factor is the threshold value of the membrane potential. This threshold allows a neuron not to output any signal even though it has received input. This leads to spatial sparsity. In an ANN, this concept is implemented using biases that are added at the end of matrix multiplication operations and subsequent application of the activation function [40]. A neuron that does not fire would be similar to an artificial neuron that has an output of zero. In many CPU or GPU architectures, exploiting these fine-grained sparsities will not result in any performance benefit.

The third and last factor is the sparseness of the neural connection graph. As noted in [18], synaptic pruning occurs during the development of the human brain, resulting in the elimination of connections between neurons that are not needed. This results in a network with relatively few connections compared to the number of neurons, as opposed to a fully connected ANN. Each neuron has a rather limited number of connections (around 5000). This is called structural sparsity.

2.5. Examples of Current Neuromorphic Architectures

As the development of Neuromorphic Computers is still in its early stages, no consensus has been reached on the desired properties or universally agreed design principles [51]. Nevertheless, several existing projects will be considered in this section and the special features of each architecture will also be described. While there are other neuromorphic architectures not mentioned here, these architectures were chosen because of their similarity to SENeCA, the main target of this design. SENeCA, meanwhile, will be explained in Chapter 3.

2.5.1. TrueNorth

The TrueNorth project was created by IBM in 2014 and is the world's first neuromorphic chip used in industrial settings. It was developed under the auspices of the United States DARPA SyNAPSE (Systems of Neuromorphic Adaptive Plastic Scalable Electronics) program [71]. Its objective was a large reduction in synaptic neurons, around 10^6, with equally ambitious goals for architecture, hardware, and applications. These resulted in a multiprocessor system with up to 10^8 neurons.

TrueNorth has around 1 million neurons and 256 million synapses, distributed across 2,096 neurosynaptic cores. It was manufactured in Samsung's 28-nm low-power process and occupies $430mm^2$ of space. During typical use, it consumes around 100 mW of power, or something of that magnitude. Each neurosynaptic core includes its own local memory that stores all the necessary data, such as the weights and biases of the neurons. It also stores synaptic connection information in the memory, allowing it to break the von Neumann bottleneck by placing the memory close to the processing unit. A single core receives at most 256 inputs (axons) and produces at most 256 outputs (neurons). The architecture of a single core is shown in Figure 2.7.

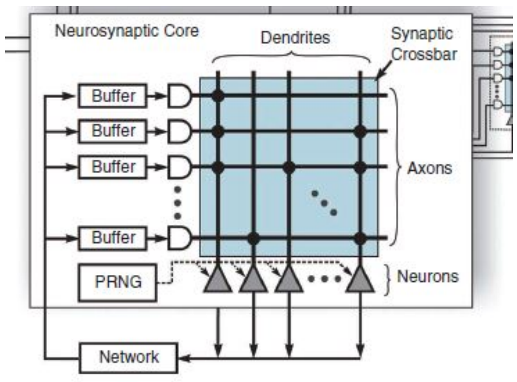


Figure 2.7: TrueNorth Architecture (single core) [23]

The axons are connected to any subset of neurons via the crossbar shown in Figure 2.7. The 256 x 256 programmable crossbar implements all-to-local connectivity within a single core. Each neuron accumulates weighted synaptic input, stores it in a high-precision variable called a membrane potential, and emits an output only if its potential crosses a certain threshold. The thresholds and weights are configured separately for each neuron. Similarly to the crossbar in each core, the connections between cores are also programmable, with each core having a spike router. This asynchronous router allows the formation of a distributed network to communicate spikes between cores. A digital data bus is used for communications between cores, and spikes are represented as Address Event Representation (AER). Each AER package contains the identifier of the sending neuron and the generation time. [23].

Of the basic mathematical operations, addition and subtraction are supported by the digital circuit of the cores, whereas multiplication and division are not. Furthermore, the weight of each synapse is coded by two bits, allowing only four types of weights. If excitatory and inhibitory synapses (positive and negative weights) are present, only two types are present. This means that learning algorithms cannot be performed, only inferences. The learning algorithm would have to be performed by another platform. In 2017, TrueNorth was shown to be capable of recognizing ten hand gestures with 96.5% accuracy while consuming only 0.18W of power [7].

2.5.2. Intel Loihi

In 2018, Intel released the first neuromorphic chip with accelerated on-device learning, called Loihi [22]. A single Loihi chip includes 128 neural cores, as well as three Pentium (x86) processors. It also has off-chip communication interfaces that allow expansion with another Loihi chip. An asynchronous network-on-chip (NoC) transports all communication between cores in the form of AER packets. Each neural core is capable of simulating up to 1,024 spiking neurons. It also has 128 KByte of Static RAM to store the states of the neurons. In total, all of the cores combined would be able to simulate approximately 128 thousand neurons and up to 128 million synapses. Each core calculates independently of its set of neurons, and any neuron that enters a firing state generates a spike message that the NoC distributes. After finishing its calculations, the core exchanges barrier messages with neighboring cores, giving notices to other cores that they are allowed to proceed with the calculation of the next timestep. This eliminates the need for a universal time reference (clock). It has also been proven that the Loihi mesh is deadlock-free.

On-chip learning is made possible by the fact that the synaptic weights are dynamically modifiable. They can be between one and nine bits long, in contrast to the two-bit weight of TrueNorth. In addition to weight, the state of each synapse is also described by a synaptic delay variable (up to six bits) and an extra variable of up to eight bits. When the core is configured, a learning formula is also specified, which determines how the weights are recalculated when the learning phase is done. This formula can only include addition and multiplication operations. Several Loihi-based neurocomputers have been created with various capacities. One of these is Pohoiki Springs, which includes 786 Loihi chips combined into 24 modules that are placed on a motherboard, simulating 100 million neurons [32].

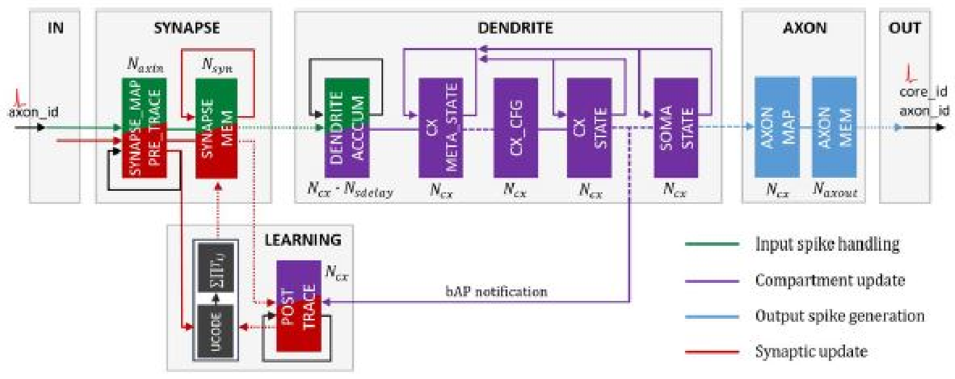


Figure 2.8: Loihi Core Top-Level Microarchitecture [22]

Figure 2.8 shows the internal architecture of a Loihi core. The SYNAPSE unit processes spikes from other cores. The DENDRITE unit updates the variables that represent the states of neurons. The AXON unit generates output spikes for all connected cores. The LEARNING unit updates the weights of the synapses using the aforementioned learning formula. The different colors represent the different operating modes available: input spike handling (green), neuron compartment update (purple), output spike generation (blue), and synaptic updates (red). Meanwhile, the blocks show the main memory blocks that hold the configuration, connectivity, and state information of the neurons in the core. The total memory capacity is 2 Mb. To deal with common bottlenecks, several degrees of parallelism and serialization are applied to the core's pipeline.

According to a study published in 2021 [21], more than 100 scientific groups from various countries are using Loihi in their research, as well as several groups focusing on applied problems. Some examples of programs that use Loihi include recognition of images and smells, data sequence processing, and the realization of a Proportional Integral Differential (PID) controller using a spiking neural network. Loihi's local learning capabilities were also used to solve several problems, such as robotic arm control [27].

2.5.3. SpiNNaker

The SpiNNaker (short for Spiking Neural Network Architecture) project [35] was started in 2011 at The University of Manchester. It was the first large-scale digital hardware platform developed exclusively for Spiking Neural Networks (SNN). The second generation of this project was undertaken with the cooperation of Dresden University of Technology in 2018 and is currently being developed as part of the European Human Brain Project [46].

Unlike the other architectures mentioned above, SpiNNaker is not a chip; instead, it is a massively parallel computer. The main part of the first generation is a custom-designed microcircuit that has 144 ARM m4 microprocessors along with 18 MByte of SRAM. These custom microprocessors have limited instruction sets compared to a typical x86 processor (for instance, not having a division instruction), in exchange for high performance and low power consumption. The second generation supports rate-based DNNs, plus several accelerators for numerical operations that were not supported before (such as exponentiation, logarithms, and random number generation). It also has dynamic power management that adjusts voltages and frequencies depending on the load of the task being performed.

The mainframe of SpiNNaker has several cabinets, each of which has 10 racks. Each rack contains 25 boards. Each board in turn has 56 chips. In total, the SpiNNaker neurocomputer has 106 processors, plus that of the control PC [68]. Similarly to other architectures, each processor operates independently, without a global synchronization mechanism. This leads to the need for AER packages for communication between processors, again similar to the aforementioned architectures. On the plus side, this gives the entire system more flexibility and scalability. The connections between the processors and the communication strategy (such as multicasting and nearest neighbor) can also be configured.

With the SpiNNaker neurocomputer, researchers can solve the problem of modeling the structure of a biological brain. A $1mm^2$ cortical column of a human brain was simulated in real time. This was demonstrated in [100]. That area of the cortical column contains around 77 thousand neurons and 285 million synapses, and it was simulated with a 0.1 ms time step. The best result of this simulation on a GPU was twice as slow as in real time, showing the power of this architecture to study the human brain. To simulate larger areas, the neurocomputer simply needs to add more chips due to its inherent scalability.

2.5.4. Tianjic

The Tianjic project was started in 2019 at Tsinghua University and developed the first chip that can work with traditional ANN and SNN, making it very versatile [81]. Furthermore, this versatility comes at the cost of only around 3% extra chip area. This is achieved due to the efficient reuse of components and circuits to perform calculations for different types of neural networks. One possibility that this architecture opens up is the combining of different types of neural network (ANN and SNN) into one larger system. Similarly to Loihi and TrueNorth, Tianjic has neural cores, of which there are 156 in one chip, plus 22 kilobytes of SRAM. Around 40 thousand neurons and 10 million synapses can be simulated. AER packets are also used on the Tianjic to represent the signals, and communication between the cores takes place on a digital bus. Tianjic is also easily scalable, as multiple chips can be combined in a 2D mesh to execute a larger neural network. However, as on-chip learning is not supported, the neural network must be pre-trained on another platform and then loaded to Tianjic, similar to TrueNorth. The network can then be run in inference mode. The performance of Tianjic compared to a GPU is described in [81], as follows:

1. SNNs run 22 times faster and 10,000 times more energy efficient.
2. LSTM runs 467 times more energy efficient.
3. MLPs run 35 times faster in terms of frame rate while consuming 723 times less energy.
4. CNNs run 101 times faster and consume 53 times less energy.

To demonstrate Tianjic's potential, an example using a hybrid ANN/CNN architecture was provided in [26]. One of the examples was a multi-modal unmanned bicycle that used an SNN for voice command recognition, a CNN for object detection and an MLP for balance control, as well as other networks performing other functions.

2.6. Benchmarking of Neuromorphic Processors

This section will explore the published research on the topic of neuromorphic benchmarking in more detail, particulary where an actual workload was tested on a neuromorphic architecture. Note that the labels here are not necessarily the titles of the publications by the authors, they are simply added to facilitate the reader.

2.6.1. Closed-loop Neuromorphic Benchmark

One of the first publications of a benchmarking was made in 2015 [92]. This study focused on what is called closed-loop benchmarks. These focus on environments where the output of the neuromorphic processor will likely influence its future input, such as interactive control of physical systems. This is in contrast to pattern identification tasks, in which the input is a fixed sequence and the hardware has to produce the correct output. In these closed-loop domains, neuromorphic processors will likely be used in the future, owing to its ability to run complex algorithms in low-power situations such as embedded devices. However, benchmarking closed-loop systems is more complex than a pattern identification, since the benchmark must specify a full system to be controlled or a software to simulate that system. A difficulty that comes with a simulation is that in robotics, simulations are much better-behaved than an actual physical system [54].

A methodology for creating closed-loop benchmark simulations was developed, based on a method known as minimal simulation [53]. Minimal simulation was developed to address the problems with developing algorithms for robotic control. Testing these algorithms with a physical robot is inefficient, while simulations would often not represent the behaviour of the physical robot. The solution was to introduce variability in the simulation itself, so the algorithm can be developed in such a way that it works across the whole range of that introduced variability. This method, which was previously not used outside of evolutionary robotics, was used here for neuromorphic benchmarks. The introduction of minimal simulation for benchmarking introduces variability to the benchmark, making it a more general benchmark which can by used by any researcher [92].

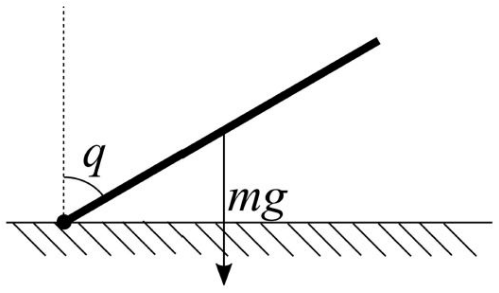


Figure 2.9: Depiction of the adaptive motor control task [92]

An example was demonstrated using an adaptive motor control task. Figure 2.9 shows a simplified diagram of the control task. To hold the joint at the angle q, a force must be applied by a controlling motor to counteract the effects of gravity (mg) and other disturbances. To determine the control signals necessary, the standard way is to use a Proportional Integral Derivative (PID) controller, which can counteract random disturbances. However, things become complicated if the desired angle is changed, say to q_d. The disturbance due to gravity depends on the desired angle q_d. In ideal conditions, this disturbance can be calculated, since the torque from gravity is $\tau = \frac{l}{2} mg \sin q$, where l is the length of the joint, m is its mass, and g is the gravitational acceleration. Then, by adding a compensating control signal that makes the controlling motor apply the extra torque τ, the desired angle q_d will be reached. However, this assumes a perfect distribution of weight in the joint, and ignores momentum, friction, and other forces. There is also no way to adjust this compensation.

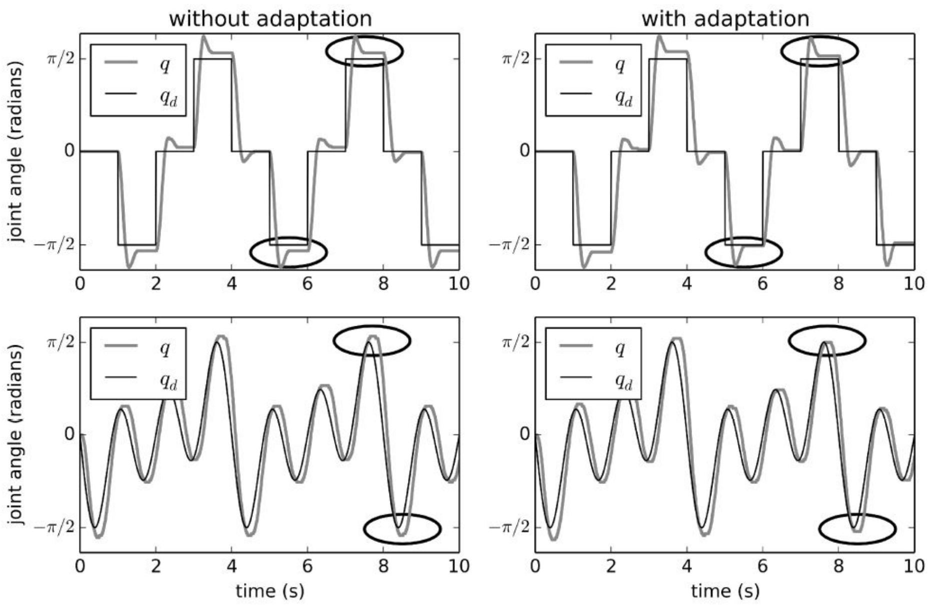


Figure 2.10: Adaptive control results [92]

Instead of a constant compensation, the minimal simulation used here introduces variations to the disturbance torque τ. The disturbance torque will be a function of q_d, but the function will change randomly every time the benchmark is run, and it is meant to simulate real-world situations more accurately. To compensate for this variation, the PID controller is replaced by a PD plus an adaptive controller based on a three-layer neural network. For each run of the benchmark, the neurons are trained to compensate for the disturbance generated during that run using a delta rule as the learning method. After a period of time, the neurons should be able to compensate better. To test the effectiveness of the adaptive control system, a test run was done, once without the adaptive system (only the PD) and once with the adaptive system. The results can be seen in Figure 2.10. In the figure, q_d is the desired angle, while q is the actual angle. q_d is constantly changed in the upper tests, and changed randomly in the lower tests. By observing the circled parts, we can see that without adaptation, q does not quite reach q_d when q_d is large. With adaptation, however, the control system adapts so that q becomes much closer to q_d after around 5s. This happens for both sets of the tests.

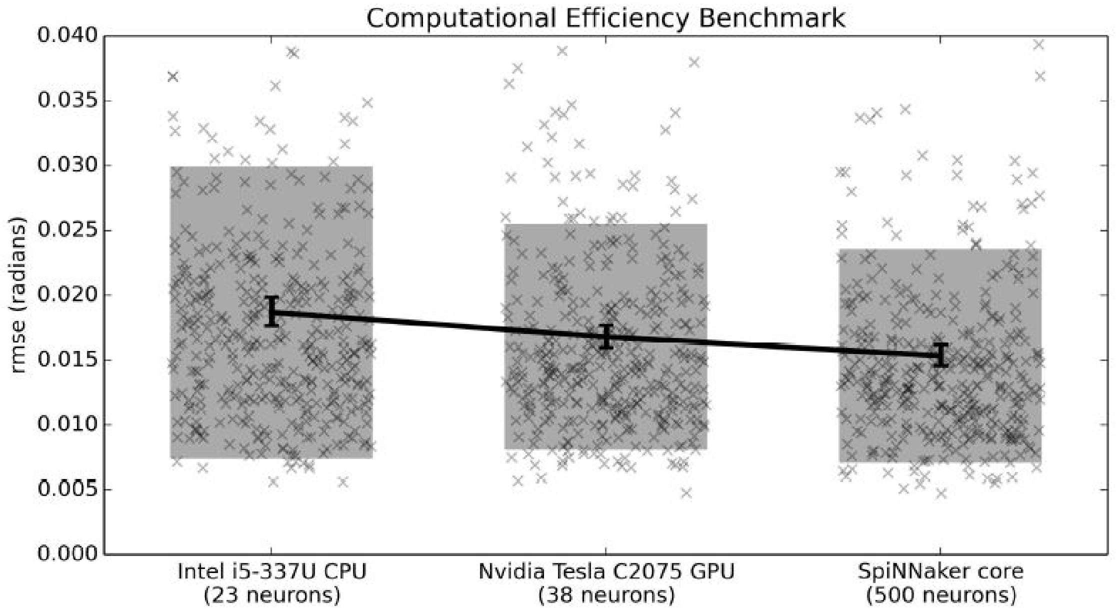


Figure 2.11: Comparisons of the adaptive control algorithm run on a CPU, GPU and Neuromorphic Processor, adjusting for power consumption [92]

The performance of a neuromorphic chip (SpiNNaker) was compared against that of a CPU (Intel i5-337U) and a GPU (Nvidia Tesla C2075). Figure 2.11 shows the comparisons of the three platforms running the adaptive control algorithm, with number of neurons adapted for each platform so that their power consumption is limited to 0.1 W. Each run (denoted by an x) uses a randomized q_d trajectory, and lasts 20s. The Y-axis shows the Root Mean Squared Error (RMSE) in radians (the final difference between q_d and q) during the last 10s of a single run (so that the neurons have time to learn). Random jitter is used on the x-axis to prevent overlap, and the shaded area is the mean RMSE of 400 runs (per platform), plus/minus standard deviation. We can see that the neuromorphic core (SpiNNaker) is not only able to have more neurons at the same power consumption level, but is also more accurate.

2.6.2. The Nengo Platform for Benchmarking

Nengo is a software package for designing and simulating large scale neural network models [11]. In 2015, it was demonstrated that Nengo can also be used to benchmark neuromorphic hardware to measure functional performance [10]. Nengo provides a high-level API that can express large-scale models in a platform-independent manner. Several simulators (backends) for Nengo have been developed, including some for neuromorphic hardware. In this study, five backends were used to run benchmarks. The five backend platforms used are listed in the following.

1. Reference: The basic backend deisgned to run on any general-purpose computer using NumPy [101], offering the most features.

2. Distilled: A backend designed as a learning/teaching tool, and as a template to build new backend. Similarly to the Reference backend, it is designed to run on any computer using Numpy.
3. OpenCL: A backend that uses the Open Computing Language (OpenCL) [94] to run Nengo models on more specialized hardware, such as GPUs and Field Programmable Gate Arrays (FPGAs).
4. Brainstorm Software: This backend was designed to run on a neuromorphic chip based partly on Neurogrid [12]. It aims to emulate the proposed hardware to test its applicability for neural models.
5. SpiNNaker Hardware: A backend designed to run on SpiNNaker [35]. This backend targets the physical neuromorphic hardware, and therefore is concerned with both accuracy and speed.

On each of the five backend platforms described above, four benchmarks that represent functioning aspects of large-scale brain models were performed. The four benchmarking programs are described in the following.

1. Communication Channel Chain: In this model, five ensembles of 100 LIF neurons each are connected in series. Each neuron computes the identity function, so the model as a whole attempts to communicate an input signal to the last ensemble in the chain.
2. Two-dimensional Product: In this model, the scalar product is computed from a two-dimensional ensemble of 100 LIF neurons. A space-filling curve (Hilbert curve) is used as the input signal. This model tests each backend's ability to compute non-linear functions.
3. Controlled Oscillator: This model uses a three-dimensional ensemble consisting of 600 LIF neurons that is recurrently connected such that the first two dimensions implement a cyclic attractor. The third dimension controls the direction and speed of the oscillation. An initial stimulus is provided as input. Each backend's ability to stably implement a dynamical system is tested.
4. Basal Ganglia Sequence: This last model implements a structure known as the basal ganglia using 4900 LIF neurons. They are organized in such that they iterate through a repeating set of actions. Additionally, a separate basal ganglia model in which some connections are pruned from the model is also tested.

For all models, the accuracy and speed of all five backend implementations were measured. Discussion of the results will be limited for brevity's sake here to the basal ganglia model, the largest model. Figure 2.12 depicts the accuracy results for the basal ganglia sequence model. Figure 2.12A depicts an example instance of the model run on the reference backend. The model quickly progresses from one action to the next, cycling back at the end of six items. The point at which the model switches from selecting one action to another is indicated with the dashed gray lines. For the original model (without pruned connections), depicted in Figure 2.12B, all backends perform similarly, except for the SpiNNaker backend. The first four backends have a transition time around 43 ms, but the SpiNNaker backend has a median transition time around 51 ms, while its interquartile range (denoted by the dark green rectangle) is also significantly larger than other backends. For the model with the pruned neurons (Figure 2.12C), however, the SpiNNaker backend has a similar level of performance to the other backends.

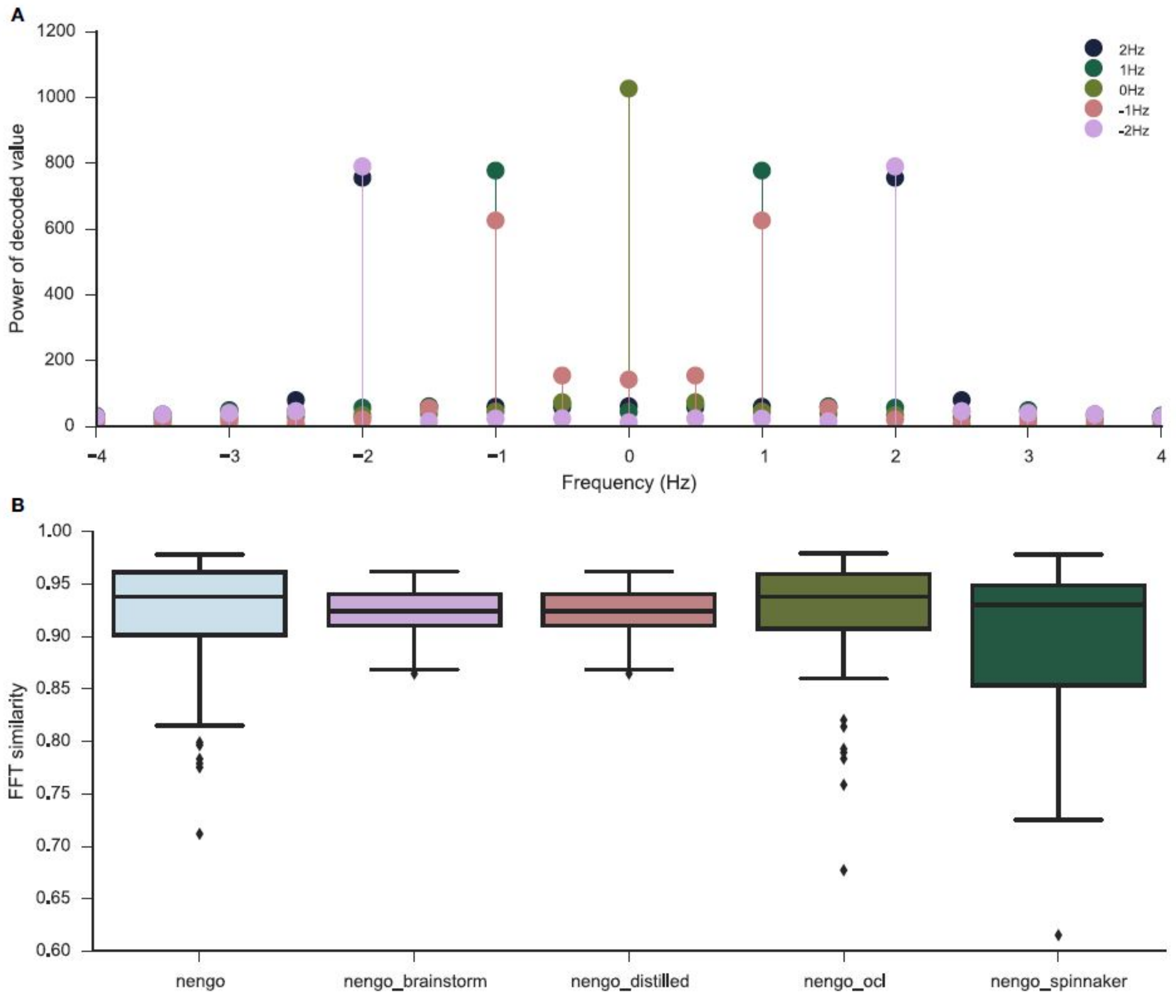


Figure 2.12: Results of the basal ganglia sequence model benchmarking accuracy [10]

Figure 2.13 depicts the run speed for each model on each backend. For the basal gangilia model, two graphs are presented, one without pruned connections and one with pruned connections. Each bar represents the mean run speed across 50 instances of each model. For the largest model, the basal ganglia model, SpiNNaker delivered the best performance, operating at around 1.2 times slower than real time.

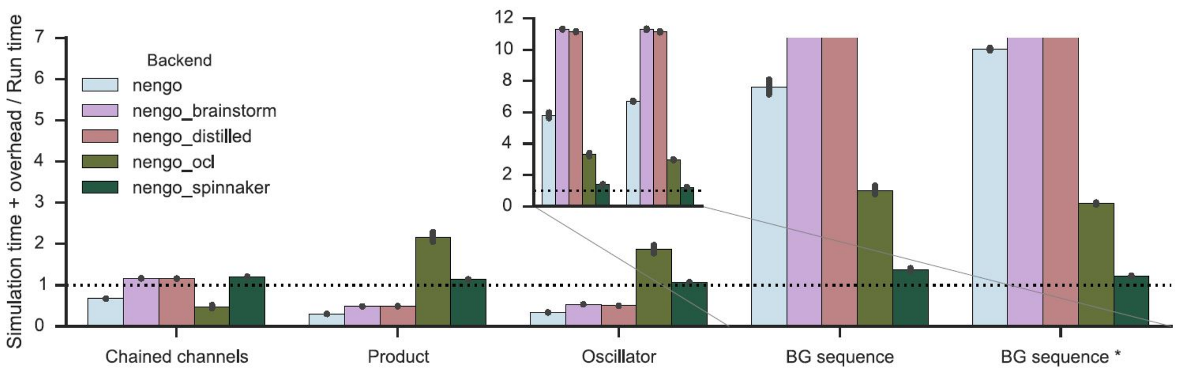


Figure 2.13: Run speed for each model on each backend [10]

2.6.3. Benchmarking SpiNNaker With DNA Sequence Matching Algorithm

A benchmarking study was published in 2019 by Urgese, et al., using an application for DNA sequence matching [99]. The neuromorphic platform used is SpiNNaker [35], and its performance is tested against that of a standard many-core CPU and GPU. Pattern matching, one of the most studied problem in computer science, has many real-world applications, one of them being DNA/RNA sequence matching [59]. In this study, an algorithm for DNA sequence matching known as Fast String Matching Method for Encoded DNA Sequences (FED), was used [61]. This algorithm assigns a unique 2-bit code to each of the four symbols composing the DNA alphabet (G,T,C,A). Then, in the pre-processing stage, all of the symbols in the search space is packed, with four elements packed into a single byte. Then a shift table is computed for every pattern to be matched. After that, the matching stage begins, where a byte-by-byte search is performed. The shift table is used to compute the number of positions allowed to be skipped if no match is found in the current search space.

The SpiNNaker chip, having 768 nodes, is pitted against a server having two Intel Silver Xeon 4114 processors, each having 10 cores and 20 threads. The text used for the sake of testing is the *Escherica coli* genome, which is about 4 million symbols long (around 1 MB of text), which is then split into a set of 4000 chunks, each 256 bytes long. In contrast to the other works on benchmarking presented here, this work does not test a neural network architecture, it uses a parallel application based on the Message Parsing Interface (MPI) framework, applied on the FED algorithm. The performance metric measured was how well the performance improved when the number of MPI workers is increased. Here, weak-scaling is used, so the ratio between the problem size and the number of MPI workers is kept constant [41].

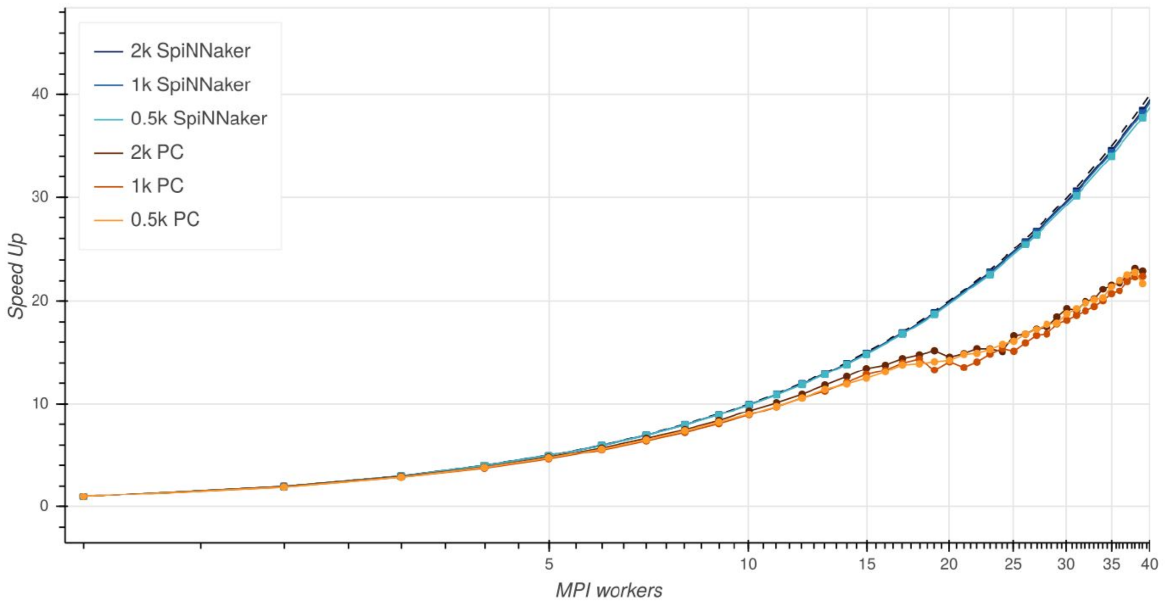


Figure 2.14: Comparison of speed-up for MPI-FED on a general-purpose CPU and SpiNNaker [99]

Figure 2.14 shows the weak-scaling speed-up of SpiNNaker and the PC. Tests were done for genomes of 500, 1000, and 2000 chunks. We can see how the massively parallel architecture of SpiNNaker influences the speedup. The high number of physical cores allows the speed to increase linearly, avoiding the discontinuities seen in the speed-up graph of the PC. The inflection point, 20 MPI workers, is when hyper-threading is activated, i.e., when the maximum number of physical cores on the Xeon processors is used.

2.6.4. SNABSuite

A more recent publication was from 2020, which used 4 neuromorphic simulators to run several benchmarking programs [77]. The software framework SNABSuite, which was introduced in an earlier work [76], was used. SNABSuite is a framework developed for ”black-box” benchmarking of neuromorphic processors, supporting benchmark networks are developed independently from the hardware on which they will be executed. These networks are presented as an abstract description, and are automatically configured to a platform specific implementation. It uses the Cypress framework, introduced in [93], to convert these network descriptions to the implementation. It is a C++ wrapper for the PyNN spiking neural network description language, supported by BrainScaleS [82], Spikey [83], NEST [39], and SpiNNaker [35] platforms. Hence, SNABSuite also supports these four platforms.

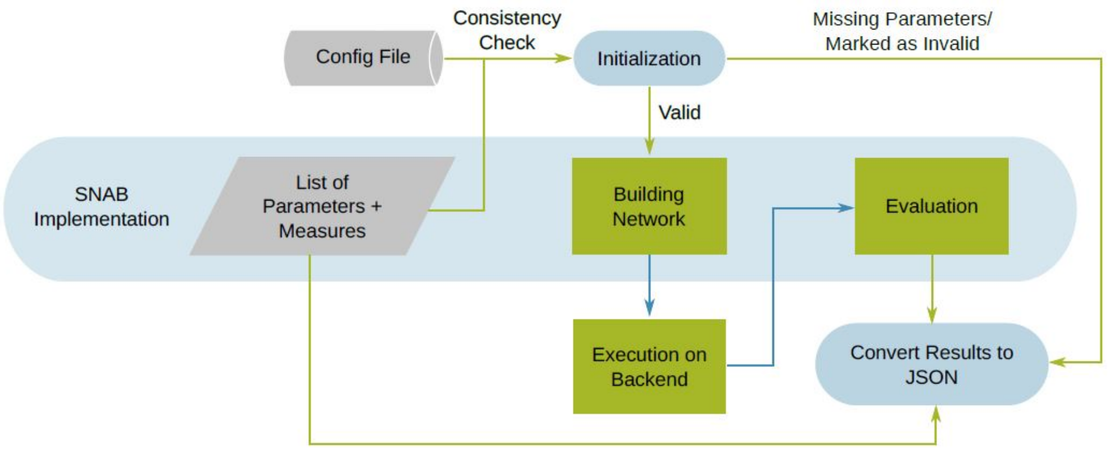


Figure 2.15: Flowchart of the SNABSuite framework [76]

Figure 2.15 depicts the flow used by the SNABSuite framework to perform benchmarking. SNABSuite only uses Spiking Neural Networks (SNNs) for performance estimation, and they are introduced to the framework via the ”Config File”. All benchmarks share a common interface, easing the introduction of new benchmarks. SNABSuite also allows the list of parameters of the backend platform and the measured indicators to be configured. This might be useful in several situations, for example for design space exploration. After an initialization phase that checks the configuration files for consistency, the Cypress framework begins the building of the network. The network is then executed on the backend platform, and the specific indicators are then measured. The measurement results are then converted to a JSON file.

Table 2.1: Results of benchmarking a network to measure maximum frequency of neurons in SNABSuite [77]

Platform	Avg. freq. (KHz)	Std. Dev. (KHz)	Max (KHz)	Min (KHz)
BrainScaleS	2.15	0.4	4.17	0.89
Spikey	2.8	0.13	2.86	2.5
NEST	5	0	5	5
SpiNNaker	1	0	1	1

Table 2.2: Results of benchmarking a network for solving Sudoku puzzles in SNABSuite [77]

Platform	Solved	Bio-time (ms)	SD (ms)	Real-time (s)	Power (W)	Energy (J)
BrainScaleS	86	3,241.90	4,573.10	0.000324	NA	0.0062
NEST	100	214.6	263.1	0.03	45	1.4
SpiNN-3	99	241.2	250	2.41	2.7	6.5
Spikey	75	3,745.80	6,041.10	0.000375	5.6	0.0021

In [77] the SNABSuite framework was used to execute several benchmarks on the four platforms supported. Among them was a low level test network to measure the output rates of a single neuron, and high-level test network to solve Sudoku puzzles. The results of the benchmarking process using the low-level network is presented in Table 2.1. The results of the benchmarking process for the high-level network is presented in Table 2.2. The benchmark criterion is the fraction of solved Sudoku puzzles for 100 puzzles and the average solving time. The solving time is separated into two, the bio-time, which is the neuron model internal time, and the elapsed wall-clock time to solution. The researchers also measured other metrics such as power consumption, and energy consumption.

2.6.5. Hand Gesture Recognition Benchmark

A benchmark based on hand gesture recognition was also published in the 2020 [17]. The input of this recognition system consists of two sensors, a Dynamic Vision Sensor (DVS) and an electromyography (EMG) armband sensor. The DVS is a neuromorphic camera built using the principles of visual processing in the biological retina, also known as an Active Pixel Sensor (APS) [64]. The DVS sends information that corresponds to changes in the illumination of the pixels of the camera, leaving out information from the static pixels. This information takes the form of a sparse and continuous train of events (also called spikes), with each event encoded in the Address Event Representation (AER) format [24]. The EMG armband, Myo by Thalmic Labs Inc., is a wearable device that records electric signals from the forearm muscles as they move and sends them. The data from these two sensors are then used to detect hand gestures.

Figure 2.16 shows an overview of the system. Figure A shows the setup of the DVS and the Myo armband. Figure B shows the data stream (spike train) from the DVS and the Myo armband. Figure C shows the neuromorphic systems that are the targets of benchmarking, which will be explained in more detail below. Finally, Figure D shows the five gestures to be recognized by the system in real time. In total, the dataset used covers 21 subjects doing three trials, with each trial having the subject do five repetitions of the five gestures. A recording of a gesture lasts 2s, which is then cut in chunks of 200ms each. Therefore, in total there are 15,750 chunks used. Since SNNs required spike trains as inputs, the outputs of the DVS can be used without modification, while the output of the EMG is converted using the delta-modulator ADC algorithm, based on a sigma-delta modulator circuit [19].

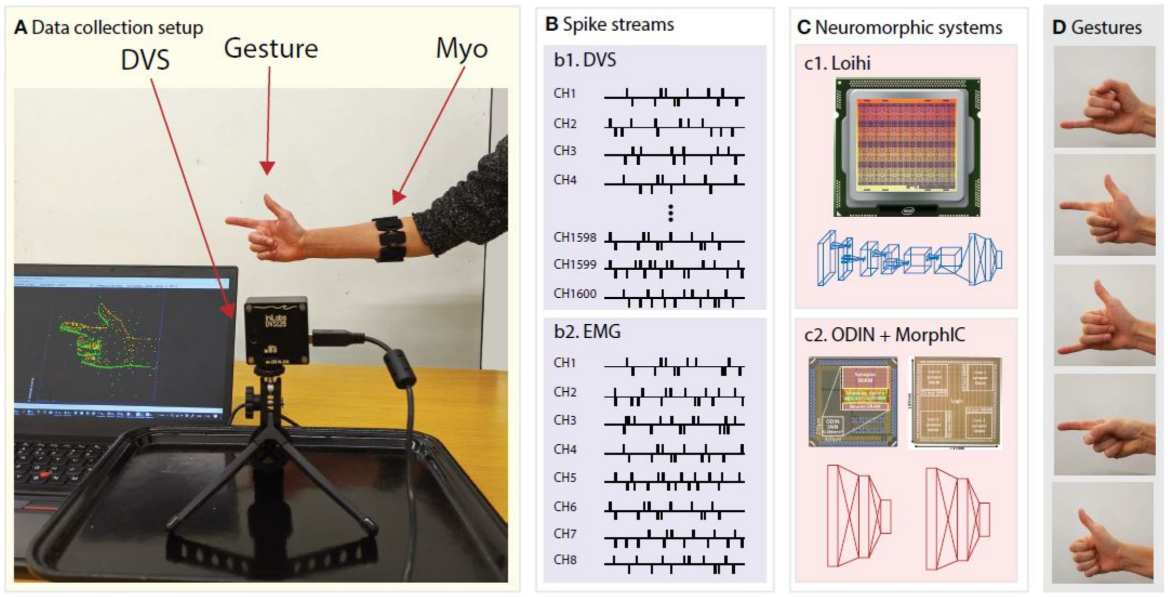


Figure 2.16: System overview of the hand gesture recognition benchmark [17]

This study used Loihi [22] and ODIN [34]+MorphIC [33] as the neuromorphic platforms to be benchmarked. ODIN is a neuromorphic processor that consists of a single neurosynaptic core with 256 neurons and 256^2 synapses, while MorphIC is a quad-core digital neuromorphic processor with 2k Leaky Integrate and Fire (LIF) neurons and more than 2M synapses. The ODIN+MorphIC setup uses an SNN architecture based on fully-connected Multilayer Perceptron (MLP) topologies. ODIN is used to process the output from the EMG sensor, while MorphIC is used to process data from the DVS. ODIN is then further used to perform sensor fusion of these two inputs and output the final result.

For Loihi, three separate networks were trained and used for this study. The framework SLAYER [89], which is a backpropagation framework used to evaluate the gradient of SNNs was used to train networks that were compatible with Loihi. The networks were trained in an offline fashion using a GPU. Three networks were used in total for Loihi, a spiking MLP for the EMG output, a spiking CNN for the DVS data, and a third network which used the penultimate layer neurons of the DVS and EMG networks. In addition, the MLP network used in MorphIC for processing the DVS data was also reused in Loihi. The details of the architecture of these networks is shown in Figure 2.17. Architecture A is the CNN architecture implemented on Loihi. Architecture B is the MLP architecture implemented on MorphIC to process the DVS output. Architecture C, the MLP network used to process the EMG output, is deployed on both Loihi (with configuration c1; the numbers indicate the number of neurons in a layer) and ODIN (with configuration c2).

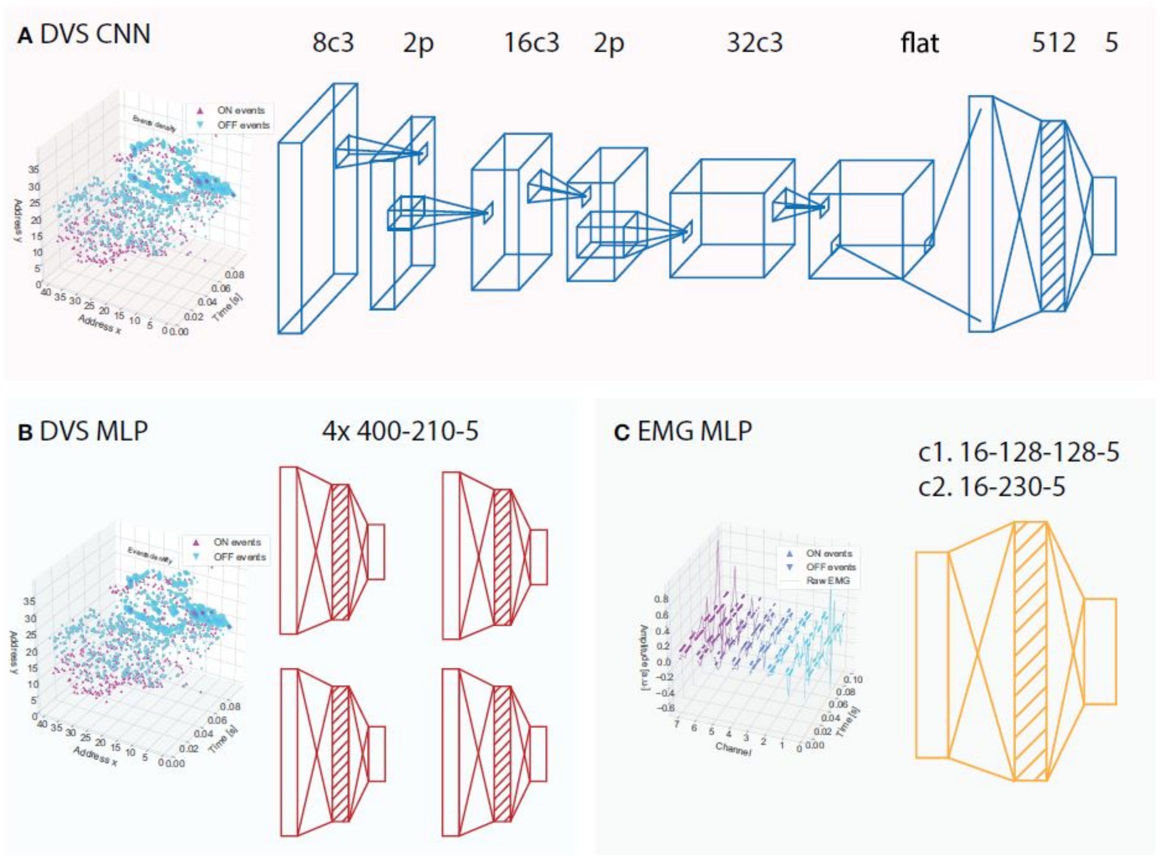


Figure 2.17: Architecture overview of the neural networks used in the hand gesture recognition benchmark [17]

For comparison, a Machine Learning (ML) based algorithm that uses frame-based inputs, i.e., traditionally sampled EMG signals and video frames, is used as a baseline against which the two fully neuromorphic architectures are compared. To have a fair comparison, the same neural network architectures were used. To extract the features in the sampled EMG signals, the Mean Absolute Value (MAV) and the Root Mean Square (RMS) are calculated over a windows of 40 samples (200 ms), and both are used as the input features of the ML-based neural network. For the video frame input, grayscale APS frames are used as input. Two baseline networks were used in total, with the description as follows. Note that all of the baseline networks are run on the NVIDIA Jetson Nano, an embedded system with a 128-core Maxwell GPU.

- Baseline ODIN + MorphIC : To process the APS frames, the 2-layer MLP architecture depicted in Figure 2.17B is used, while the 2-layer MLP architecture depicted in Figure 2.17C2 is used to process the EMG features. The fusion network is obtained as described previously.
- Baseline Loihi : To process the input from the APS, the same architecture depicted in Figure

2.17A is used for the APS baseline, receiving input from the APS instead of the DVS. For the EMG baseline, the same architecture depicted in Figure 2.17C1 is used, receiving MAV and RMS features instead. The fusion network is obtained by eliminating the classification (last) layer from the two networks, concatenating the two penultimate layers of the APS and EMG networks, and adding a common classification layer with five units.

The accuracy, energy consumption, and inference time were measured for each of the platforms. Furthermore, the Energy Delay Product (EDP) was also calculated. The overall results of the study in tabulated in Table 2.3. The spiking MLP architecture was also implemented on Loihi, but since the obtained accuracy is worse than the CNN architecture, the results for that implementation is left out of the table. By observing the values in the table, a few things can be inferred. First, the accuracy of the baseline models (marked (GPU)) is actually higher for the EMG-only setup. This is due to the fact that the baseline model uses the raw signal values from the EMG armband, while the neuromorphic implementations use an encoded input, losing some of the information. However, the final fused implementations (EMG+DVS) on the neuromorphic chips, outperform the GPU implementation, both for the CNN and the MLP architectures. Also, with regards to the inference time, the baseline GPU implementations outperform the neuromorphic chips in all (equivalent) situations. The EDP values, however, which are calculated by multiplying the energy cost with the inference time, are significantly better for the neuromorphic chips (between 30x and 600x more efficient).

Table 2.3: Results of benchmarking the gesture recognition system using different architectures and platforms [17]

System	Modality	Accuracy(%)	Energy(μJ)	Inf. time(ms)	EDP(μJ*s)
Spiking CNN (Loihi)	EMG	55.7±2.7	173.2±21.2	5.89±0.18	1±0.1
	DVS	92.1±1.2	815.3±115.9	6.64±0.14	5.4±0.8
	EMG+DVS	96±0.4	1104.5±58.8	7.75±0.07	8.6±0.5
CNN	EMG	68.1±2.8	(25.5±8.4)·10^3	3.8±0.1	97.3±4.4
	DVS	92.4±1.6	(31.7±7.4)·10^3	5.9±0.1	186.9±3.9
	EMG+DVS	95.4±1.7	(32.1±7.9)·10^3	6.9±0.05	221.1±4.1
Spiking MLP (O+M)	EMG	53.6±1.4	7.42±0.11	23.5±0.35	0.17±0.01
	DVS	85.1±4.1	57.2±6.8	17.3±2	1±0.24
	EMG+DVS	89.4±3	37.4±4.2	19.5±0.3	0.42±0.08
MLP (GPU)	EMG	67.2±3.6	(23.9±5.6)·10^3	2.8±0.08	67.2±2.9
	DVS	84.2±4.3	(30.2±7.5)·10^3	6.9±0.1	211.3±6.1
	EMG+DVS	88.1±4.1	(32.0±8.9)·10^3	7.9±0.05	253±3.9

2.6.6. Keyword Spotting Benchmark

In 2019, another more comprehensive study on benchmarking was published, which used a keyword spotting program to benchmark Loihi [22] against other, more traditional platforms [14]. Keyword spotting involves monitoring a real-time audio stream to detect some keyword of interest. As mentioned in [20], this particular problem may benefit from energy efficient implementations due to its potential deployment in embedded devices. Therefore, the performance, power consumption, and energy consumption of a neuromorphic processor (Loihi) was compared with other platforms for this program.

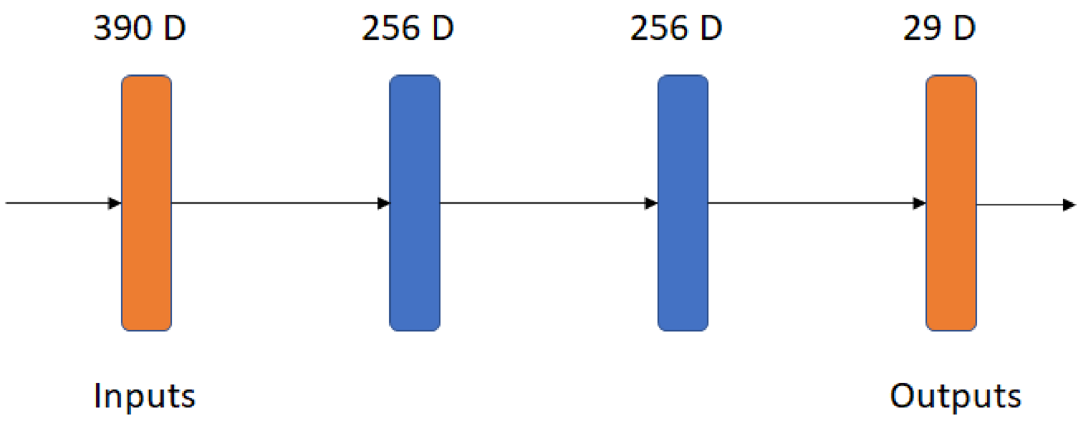


Figure 2.18: Architecture of the keyword spotting DNN [14]

In the study, a DNN architecture with three dense layers (and one input layer) is used to perform the classification task (depicted in Figure 2.18). The audio recordings are preprocessed by performing Fourier transforms on overlapping frames of the recording (each having a length of 10ms) to obtain the Mel-frequency Cepstral Coefficient (MFCC) features of that frame. These features, in the form of a 390-element vector will be fed to the DNN as input. The output of the network, a 29-element vector, represents the probabilities of sounds being included in the recording. By performing inference for multiple parts, a phrase can be obtained from the sequence of characters. The network was pre-trained, and the benchmarking only involved the inference process. On the baseline TensorFlow implementation, the network had an accuracy of around 92.7%. Also, since each input vector represents an audio frame of 10 ms, the processing rate must exceed 100 inferences per second to process data in real time.

Table 2.4: Overall benchmarking results using the keyword spotting program [14]

Hardware	Idle	Running	Dynamic	Inf/s	Energy/inf
GPU	14.97	37.83	22.86	770.39	0.0298
CPU	17.01	28.48	11.47	1813.63	0.0063
JETSON	2.64	4.98	2.34	419	0.0056
MOVIDIUS	0.21	0.647	0.437	300	0.0015
LOIHI	0.029	0.11	0.081	296	0.00027

The other platforms tested include a CPU and a GPU, as well as other neural network accelerators such as Movidius and NVidia Jetson. To obtain a more accurate measurement of the energy consumption, the idle power was measured for all of the platforms before running the program, and then the running power was measured. By taking the difference between the two power consumption values, the dynamic power can be calculated. The energy required per inference was calculated by multiplying the dynamic power by the time it took each platform to complete one inference. The results are presented in Table 2.4. By observing the ”Energy/inf” column, we can see that Loihi is indeed much more energy efficient compared to the other platforms. It outperforms its closest competitor, Movidius, by more than 5 times, while maintaining an inference speed that would allow it to process audio data in real time (more than 100 inferences per second).

To observe how the performance and the energy costs of Loihi and its closest competitor, Movidius, change with repsect to the workload, an additional set of experiments were performed. Instead of using the architecture depicted in Figure 2.18, the network was expanded with branches as depicted in Figure 2.19. Note that only the inference speed and energy cost are measured, since this expanded network cannot actually output meaningful results with respect to the classification task without retraining the parameters. The number of hidden layers is increased by 10, and the width of the network is also expanded. The parameter N is used in the experiments to set the network width and is varied from 0 to 10 with an interval of 2. As before, the dynamic power and inference speed of the two platforms are measured, and the energy cost per inference is calculated.

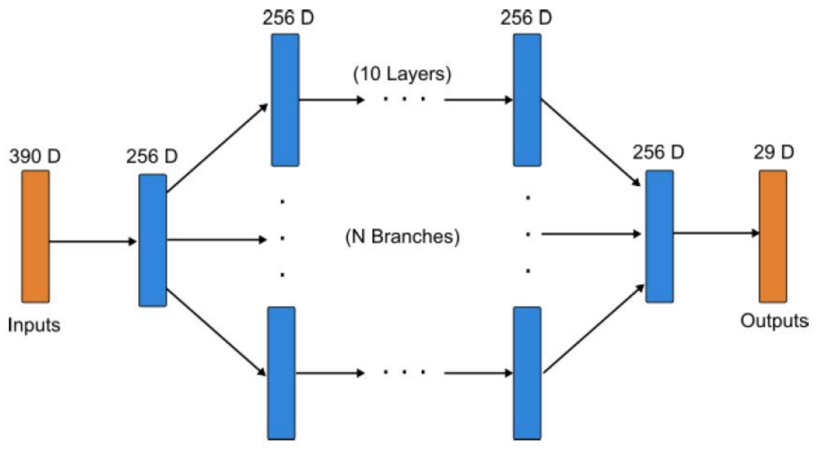


Figure 2.19: Architecture of the expanded DNN used for scaling analyses. Note that the size of the network is a function of a configurable parameter N [14]

The results of these additional experiments are shown in Figure 2.20. From the left, the three graphs show the average dynamic power, inference speed, and energy cost per inference of Movidius and Loihi as a function of the network size, here denoted by N. The dynamic power consumption of both devices, as expected, increases with the network size, but it rises much faster for Movidius than Loihi. Regarding the inference speed, it decreases for both as a function of network size, although, as is the case with the dynamic power, it decreases at a faster rate for Movidius. Furthermore, the dotted line indicates 100 inferences per second, the rate required to process audio data in real time. We can see that Loihi is able to process data in real-time for all values of N, while Movidiues fails for networks with an N value larger than 6. Finally, in the graph showing the inference cost, we can see that Loihi outperforms Movidius for every value of N, with the difference in performance being more pronounced as N increases.

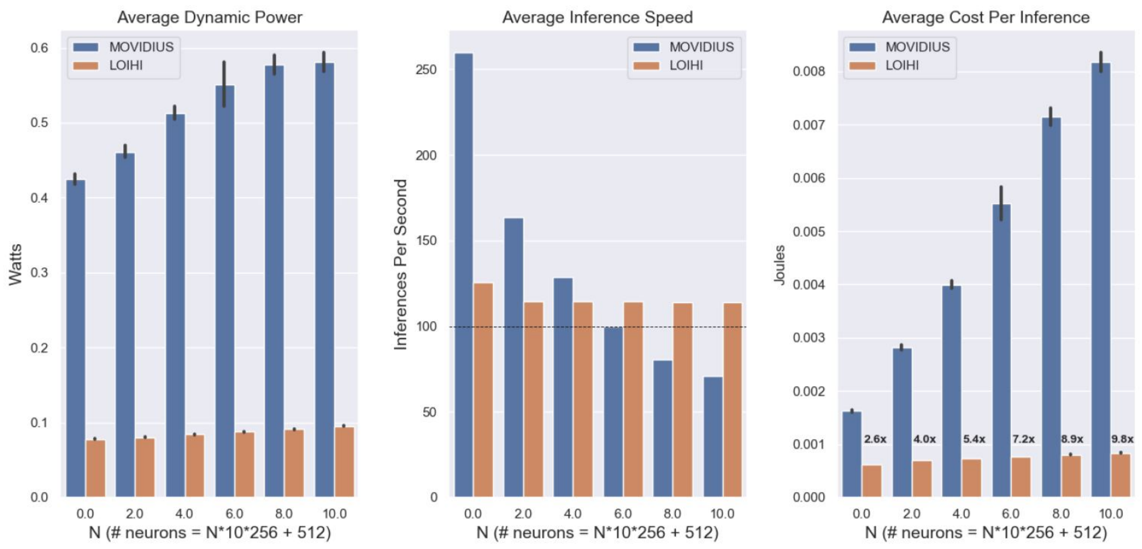


Figure 2.20: Architecture of the keyword spotting DNN [14]

While evidently some progress has been made in the benchmarking of neuromorphic processor, the publications mentioned above vary in terms of metrics that were measured and the methodology used to measure them. This study aims build upon the existing research by proposing a flow to perform benchmarking as well as software development that might provide deeper insight into how the design choices in SENeCA affect the performance on neuromorphic processors, specifically SENeCA [109]. To do this, instead of developing an entirely new benchmark, the keyword spotting benchmark described above will be used as baseline. This will be further explained in Chapter 4.

3 Architecture of SENeCA

As mentioned briefly in the previous chapter, SENeCA (Scalable Energy-efficient Neuromorphic Computer Architecture) is a RISC-V-based digital neuromorphic processor that was designed to accelerate Spiking Neural Networks (SNNs) [109]. Its main targets are extreme edge AI applications near or even inside sensors, where ultra-low power consumption and adaptivity features are a must. It is optimized to exploit both temporal and spatial sparsity in the computations and transfer of data occurring in neural network systems. It is digital IP that consists mainly of an interconnected Neuron Cluster Core network, with a RISC-V-based instruction set. Each core has processing elements called Neural Processing Elements and uses an event-based communication infrastructure. This chapter will go into more depth about the architecture of SENeCA, the platform for which this benchmark was designed. Section 3.1 gives a brief introduction to the architecture. Section 3.2 will dive into the detail of the design of SENeCA, including its components. Section 3.3 will give a brief comparison of SENeCA with other existing chips.

3.1. Introduction

State-of-the-art Deep Neural Network programs and similar architectures have exceeded the ability of the biological nervous system to complete tasks that were previously thought to be difficult for digital computers, such as image recognition. For example, on the ImageNet dataset, a very low error rate of about 3 % was achieved in 2016 [43]. However, most of the networks are designed without considering one of the main factors in biological evolution that led to biological brains, energy consumption. Natural selection pushed evolution toward energy-efficient structures and algorithms. The human brain is an excellent example of this phenomenon, even if it is still an energy-hungry organ in the human body (consuming around 20% of the total energy needs of humans).

At low levels, the elements of the biological fabric in human brains are not as fast and power-efficient as modern silicon technologies. If the operations per unit of energy are compared, for example, one joule of energy can power 10T synaptic operations in the brain [25], while it can also power 195T operations in modern ReRAM technology[105]. However, it is the combination of both hardware and algorithms that makes the brain so power efficient, as no current computing platform can come close to it. It is a perfect example of a hardware design that is optimized for the specific algorithm that it is running. The goal of neuromorphic processors is, as mentioned previously, to utilize the principles found in biological neural networks to process raw sensory data with the minimum amount of energy.

While research of the human brain is an active field of research, and many secrets of the computation that takes place are yet to be revealed, some principles have been identified. These principles of the co-design of algorithms and hardware can be used to design efficient platforms. Some of them are listed as follows [109]:

1. Spatiotemporal sparsity
2. Parallel processing
3. Infinite scalability
4. Low-precision parameters
5. Asynchronous - Non-deterministic execution
6. Adaptivity and fault tolerance

The various designs of neuromorphic platforms that are being developed right now, including the ones described in Chapter 1, are designed with these principles in mind to answer today's computing challenges. However, since the number of neurons and their connectivity in the brain is huge, it is not feasible or efficient to copy the architecture of the brain piece by piece. Furthermore, data can travel or be processed much faster in silicon than in biological fabric, as mentioned previously. Therefore, it makes sense to time-multiplex a single silicon neuron to emulate the actions of multiple biological neurons. Essentially, this means using one neuron to perform the actions of approximately one million neurons, since the silicon neuron can operate a million times faster. Additionally, the time-multiplexing of one neuron allows more efficient use of the chip area, leaving other resources free to perform other functions. Indeed, this is the approach used by many architectures.

However, significant overhead is required for efficient time-multiplexing. Namely, a complex controller to handle the time-multiplexed silicon neuron. Indeed, it might be said that the challenge of designing the controller might be more complex and difficult than designing the neuron itself, especially in a 2D silicon technology. This is because each neuron could be connected to thousands of other neurons. Improvements in manufacturing technology or silicon design, such as smaller technology nodes, novel materials, and 3D fabrications, can help overcome this difficulty. As for the circuit design, using a packet-switched Network-on-Chip (NoC) to do time-multiplexing of the synaptic connections can also be a helpful solution. In this concept, instead of implementing the connections between the neurons directly on silicon and having the spikes be single pulses, the spikes are encoded in a packet of data that contains other information, such as the destination of the spike, which the NoC then manages.

In SENeCA, a flexible and scalable neuromorphic architecture of IMEC, this problem is partially handled by sharing the complex controller between several of the silicon neurons. Also, the NoC concept is used, and the encoded packet of data also contains multiple spikes to reduce the overhead of sending a spike. The mentioned neuromorphic fundamental principles are taken into account, while flexibility over many applications and algorithms is maintained. The architecture is also compatible with event-based sensors and modern cameras. It also supports many models of neural networks and synaptic connectivities, while at the same time enabling several optimizations of the algorithm. In Figure 3.1, one instance of SENeCA is shown, which contains 64 cores. To give a more concrete understanding of SENeCA, in the next section, a comparison with other architectures mentioned in Section 1 will be given.

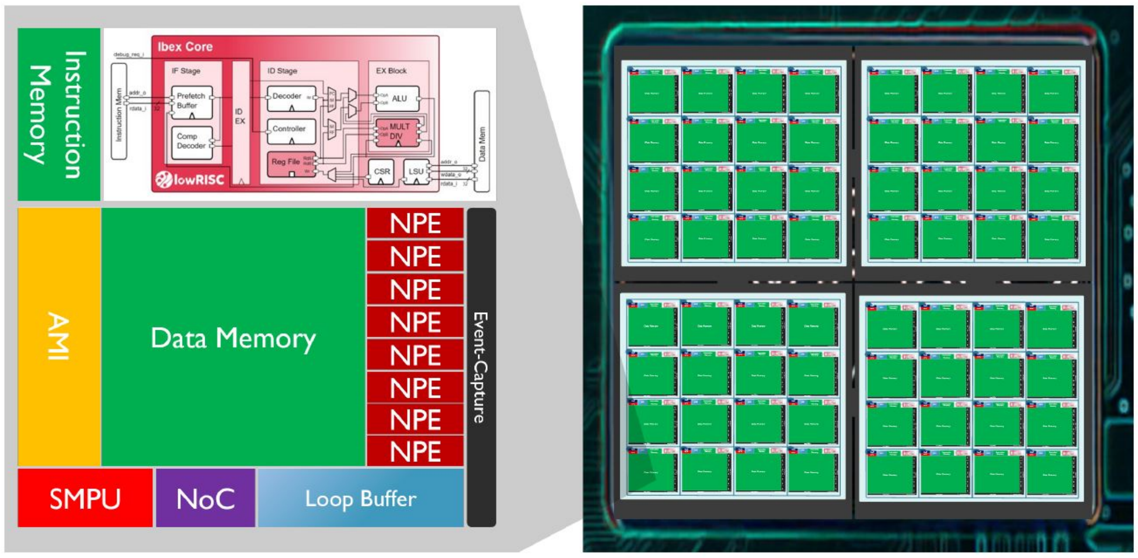


Figure 3.1: An example implementation of SENeCA, containing 64 Neuron Compute Clusters (cores) [109]

3.2. Detailed Architecture

In this section, the architecture of SENeCA will be explained in more detail, including its components. A core is called a Neuron Computing Cluster (NCC). Each core, in turn, contains a RISC-V core (Ibex), instruction memory, Axon Messaging Interface (AMI), Shared Memory Prefetch Unit (SMPU), and the main processing unit, called the Neuron Co-Processor (NCP). Each NCP, in turn, contains neural Processing Elements (NPE), the tightly coupled data memory, a loop buffer, and an Event Capture Unit (EVC)). Other than that, the cores also include a instruction MUX/arbiter that acts as the data bus. Both of the memory blocks (instruction and data) are based on conventional Static Random Access Memory (SRAM) blocks, with multi-port SRAM blocks for the data memory. The number of SENeCA cores is configurable to facilitate upscaling, so this instance is only an illustration. Configurable parameters will be set to specific values for this benchmark, and the values will be mentioned in Chapter 5.

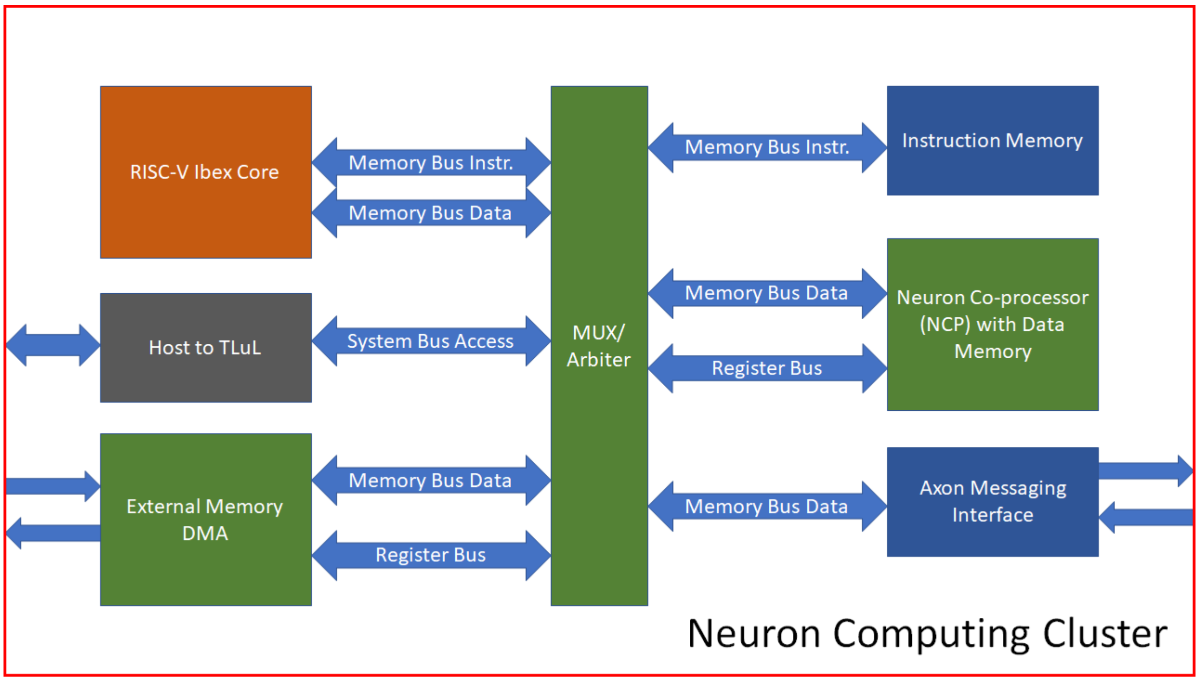


Figure 3.2: Illustration of a single SENeCA core (Neuron Computing Cluster)

3.2.1. RISC-V Core (Ibex)

Ibex is an open-source RISC-V CPU core originally developed by the PULP team and has since been contributed to lowRISC [5]. This small 32-bit processor with a 2-stage pipeline is used as a controller for each NCC. It uses the RV32IMC instruction set. IMC denotes the extensions added to the basic RISC-V core, which includes Integer (I), Multiplication and Division (M), and Compressed (C). The core is highly parameterizable and is well suited for embedded applications, including edge AI applications. The 2 pipeline stages are:

1. Instruction Fetch (IF): Fetches instructions from memory via a prefetch buffer, capable of fetching a maximum of 1 instruction per cycle
2. Instruction Decode and Execute (ID/EX): Decodes fetched instruction and immediately executes it, with reads/writes to the register all occurring in this stage.

Figure 3.3 shows this pipeline in more detail.

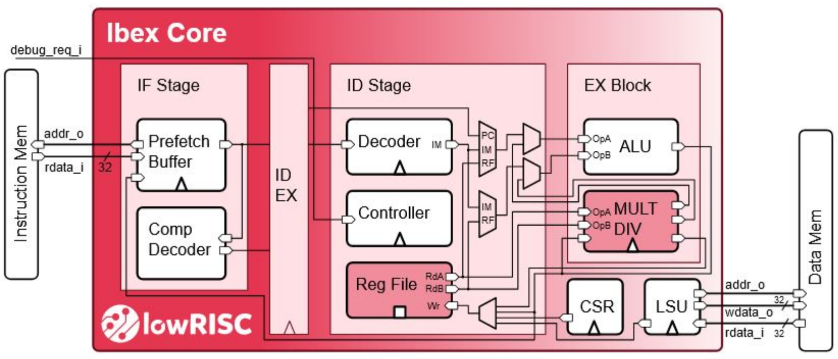


Figure 3.3: Block diagram showing the small parametrization of the Ibex core with a 2-stage pipeline, IF and ID/EX [5]

The IF stage of the pipeline can supply at most 1 instruction to the ID stage if the instruction memory can perform at the same rate. The fetched instructions are stored in a prefetch buffer for performance reasons, with the buffer filling linearly until it is full. The instructions are stored along with the program counter (PC) from which they came from.

The ID stage, on the other hand, controls the overall decode/execution process. This section contains the decoder to issue control signals based on decoded instructions, a controller that controls the overall execution of the processor, and the multiplexers required to choose the data being sent to the Arithmetic Logic Unit (ALU), and the data which will be written into the register at the end. A small state machine is used for multi-cycle instructions. If an instruction happens to be a multi-cycle instruction, the pipeline will stall until it has completed execution. This means that the maximum Instructions per Cycle (IPC) of Ibex is 1 if no multi-cycle instructions are used.

The ALU of Ibex is a purely combinational block that implements operations required in the RISC-V specification. These operations include the Integer Computational Instructions and Control Transfer Instructions, which cover arithmetic and comparison operations. Apart from that, the ALU is also used by other blocks, such as the Mult/Div block and Load/Store Unit (LSU) for address calculations and branch targets. Multiplication and division operations are handled by the MULT/DIV block. This is a state machine-driven block that may use the ALU in multiple cycles to execute multiplication/division algorithms. There are multiple algorithms possible for multiplication, while only 1 is available for division. In this instance, the Fast Multi-cycle Multiplier version is implemented, which is a reasonable trade-off between area and performance. It can complete multiplications in 3-4 cycles by having its own multiply-accumulate unit. Division, on the other hand, uses the long division algorithm, which takes 37 cycles to complete.

Other blocks included in Ibex are the Control and Status Register Block (CSR) and the Load-Store Unit (LSU). The CSR contains all of the control and status registers, such as the machine status registers, the interrupt enable register, etc. On the other hand, the LSU takes care of accessing the data memory. The load and store operations of 32-bit words, 16-bit half-words, and individual bytes are supported.

The Ibex core was chosen as the controller for SENeCA's cores, as it allows the optimization of energy/area by :

- Event-Compression
- Flexible memory allocation for the most optimal use of the limited on-chip memory
- Efficient use of the memory hierarchy
- Efficient reuse of parameters to reduce memory access in certain neural network architectures (CNN, for example)
- Deployment of online learning algorithms to improve accuracy and power efficiency

3.2.2. Axon Messaging Interface (AMI)

Since SENeCA lacks a central controller that oversees all cores, communication is done via a handshaking mechanism handled by the Axon Messaging Interface. Figure 3.4 depicts the AMI component. The AMI manages the incoming/outgoing events and messages to/from each core. Each core has 2 interfaces (with 2 FIFO buffers), one for sending and one for receiving. In this study, since a DNN is implemented, it is used to transmit neuron outputs from one layer to another. Two signals are used for this mechanism, the Data Valid (DV) and Ready signals. The DV signal is controlled by the sending core, while the Ready signal is controlled by the receiving core. When a core wants to send, the RISC-V controller puts data in the send FIFO buffer, and the AMI then sets the DV signal to 1. If the receiving buffer in the AMI of the receiving core is not full, then the AMI sets the Ready signal to 1, and data transfer occurs. The AMI of the receiving core then triggers an interrupt to the RISC-V processor. Although the length of the messages is configurable, this instance can send 32-bit flits.

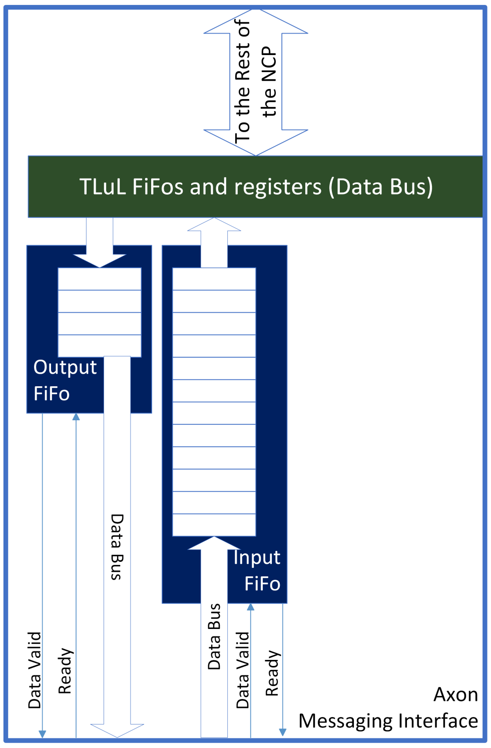


Figure 3.4: Illustration of the Axon Messaging Interface.

3.2.3. Shared Memory Pre-fetch Unit

To allow for the possibility of memory-intensive applications, shared memory is available as an optional feature of SENeCA. This shared memory can be used if the on-chip SRAM on the NCCs is not enough to store the data. This is a separate level of memory hierarchy shared between NCCs that allows the use of a different memory technology that can remove the limitations of SRAM technology. It can be added to the chip instead of more NCCs, for example, when cost or space limitations prevent the addition of more cores. The shared memory architecture is shown in Figure 3.5. However, it should be noted that, unlike GPU architectures, shared memory is not used to communicate between different cores.

To efficiently implement shared memory, the SMPU block was added. This is an optimized Direct Memory Access (DMA) that accelerates access to the shared memory by having a direct link to the arbiter of the memory. To reduce latency overhead, the SMPU fetches the required parameters to process a specific event from shared memory, while the event itself is still waiting in the AMI input event queue. With prefetching, the increase in the processing time of a single inference is only around 20% when the data (neuron states and weights) are stored in the shared memory as opposed to the local NCC SRAM, assuming enough memory is available.

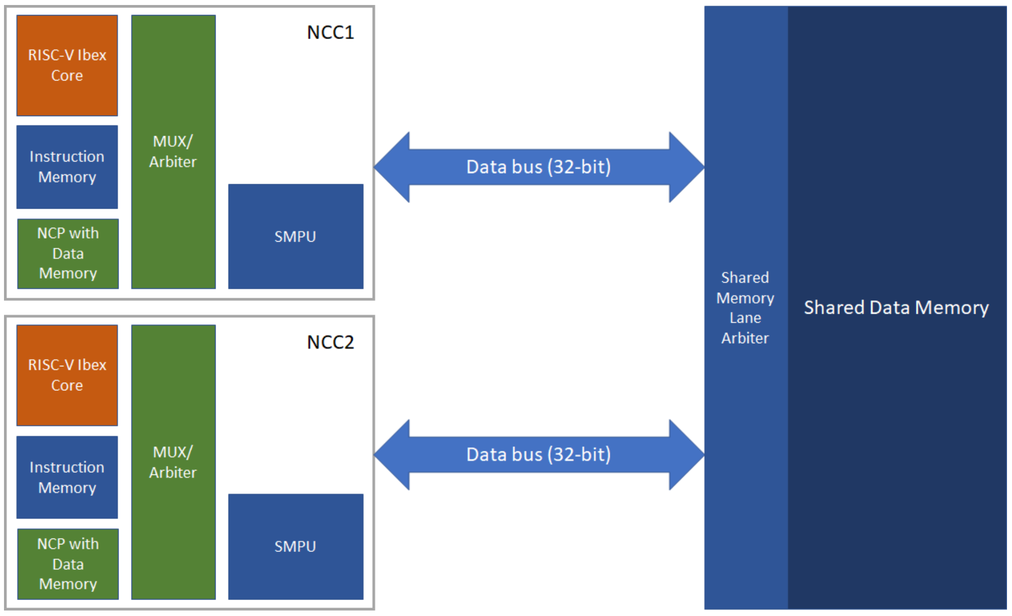


Figure 3.5: Illustration of shared memory, with the Shared Memory Pre-fetch Unit in every core

It should also be noted that the read/write command issued by the Ibex core of each NCC can only make the SMPU move data from the shared memory to its own SRAM or vice versa. A separate read/write command needs to be issued to read/write to the SRAM itself. In addition, the shared memory arbitrator uses a round-robin scheduling scheme, which allocates time slots to all NCCs of equal length.

3.2.4. Neuron Co-Processor (NCP)

The Neural Co-Processor (NCP) is the main element of SENeCA's core and performs the actual neural computations. The initial vision for the NCP is a programmable and flexible event-based accelerator. This is achieved by accelerating the most common neuromorphic operations to emulate silicon neurons instead of physically implementing different models. This approach allows the implementation of an evolutionary set of instructions that are accelerated by hardware as building blocks, which can then be used to make various neuron models. This approach was taken on the basis of the fact that the field of neuromorphic computing is continuously evolving. As a result, new neuron and synapse models are emerging rapidly, one after another. Having the building blocks of the most common operations in the NCP, SENeCA is well equipped to implement these new models. It is designed to be modular, scalable, and flexible.

The elements of the NCP are described below.

Neuron Processing Element (NPE)

The NCP includes an array of Neuron Processing Elements (NPEs), which are the silicon neurons of SENeCA. Essentially, an NPE is made of two parts: a small memory made of registers and a processing unit that can execute the most common neuromorphic instructions. Each NPE can execute one operation per cycle on average in a pipelined fashion. An array of NPEs constitutes what is commonly known as a single input multiple data (SIMD) architecture, since all of the NPEs in one array execute the same set of instructions. Figure 3.6 shows the block diagram of a single NCP. The NPEs are the 2 small blue rectangles containing the RF and ALU components on the top right. In the instance used for this benchmark, there are 8 NPEs in this array. On a side note, all individual NPEs can be turned off during idle times.

The instructions, as well as the addresses in memory of the data on which the instructions are to be performed, are stored in the micro-kernel, also shown in 3.6. The micro-kernel is part of the loop buffer. The instructions are written here by the Ibex core, which will also give a command to the NPEs to start executing instructions in the micro-kernel. The instructions are read/write memory operations or ALU operations. The Ibex core can also give commands to write data directly to the NPE registry files (marked RF in Figure 3.6), allowing data reuse in a cache-like fashion. The addresses of the data are also stored in the micro-kernel as pointers to an address in the data memory. The first NPE in the array will then use that pointer to perform a read/write operation, with the next NPE executing the same instruction on the next word. These read/write operations are essentially moving data from the register files to the data memory, and vice versa. The array of NPEs is connected through a wide port to the data memory, thus allowing each NPE to access the memory at the same time. This avoids bottlenecks due to data access.

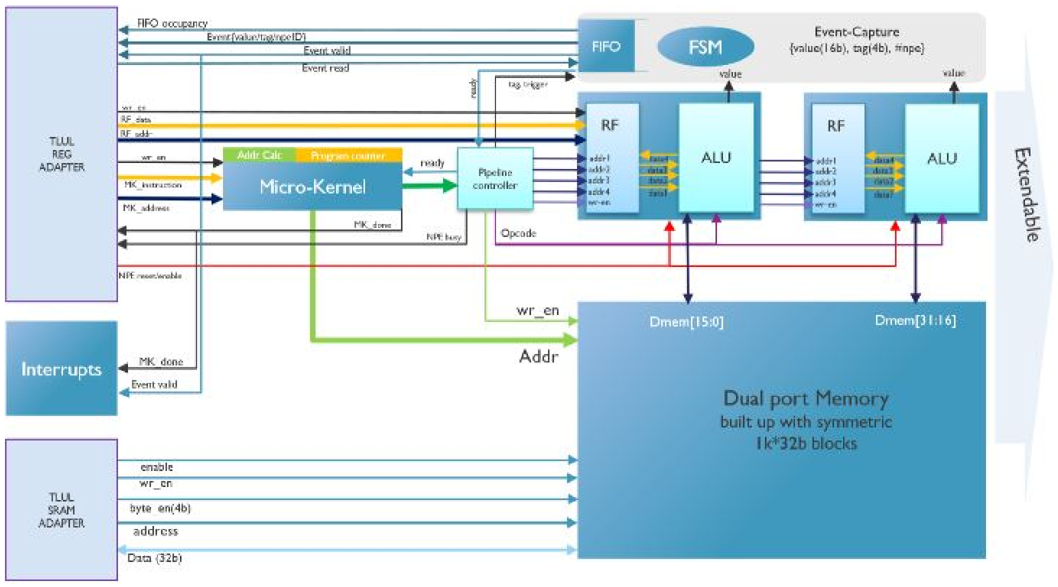


Figure 3.6: Block diagram of the NCP, with the blocks above the dual port memory containing an RF block and an ALU being individual NPEs

Regarding the format to be used for processing in the NPEs, BrainFloat16 (or BF16) was chosen. This format was originally proposed by Google for its TensorFlow platform to address the narrow dynamic range of the IEEE FP16[58]. Put simply, it is the first 16 truncated bits of a float32 number. This format has several advantages for custom hardware designs, listed below.

- Has the same range as IEEE 754 FP32 format, allowing fast conversion to and from FP32 (by truncating the last 16 mantissa bits)
- Has 8 bits of Sign+Mantissa, which allows for fast conversion to and from fixed-point 8-bit signed integers.

As an additional feature of SENeCA, a technique called Flexpoint [62] is also available to allow the sharing of a single exponent by several parameters. In this method, only the Mantissas of a group of parameters are stored in memory, in signed 8-bit, signed 4-bit, or unsigned 4-bit integer format. These parameters can, for example, be the weights of a layer of a neural network. Meanwhile, the exponent is shared, so only one instance is stored. A similar technique is used in TensorFlow, where the integer parameters are stored separately, with the scale factor shared[52].

All possible operations of the NPE ALU are shown in Table 3.1. All of the instructions are implemented with zero detection, allowing the instruction to be terminated early in the pipeline if an input is zero, therefore saving energy. These operations have operands 1,2,3 (op1, op2, op3) written in their comments, which indicates the registers that contain the desired operands if an R is denoted. Most operations are quite straightforward; for example, the addition operation performs an addition of the numbers contained in registers op1 and op2 and stores it in register op3. Logical operations (operations 7,8,9) produce a 0 if false, and a 1 if true. Furthermore, the ALU of the NPE is designed to complete all instructions in 4 pipelined stages. If there are no pipeline hazards between the instructions, then all of the instructions can be done subsequently (starting one instruction every clock cycle), and the results of one instruction can be accessed 4 clock cycles later.

Table 3.1: Available instructions for the ALU of an NPE. Operations 2 to 6 are arithmetic, 7 to 12 are comparisons, 14 to 17 are bitwise, while 18 and 19 are memory operations.

No	Operation	OpCode	Comment
1	No Operation	NOP	no operation
2	Addition	ADD	R[op3] = R[op1] + R[op2]
3	Subtraction	SUB	R[op3] = R[op1] - R[op2]
4	Multiplication	MUL	R[op3] = R[op1] * R[op2]
5	Division	DIV	R[op3] = R[op1] / R[op2]
6	Rounding	RND	R[op3] = Round(R[op1])
7	Greater than	GTH	R[op3] = (R[op1]>R[op2])
8	Greater than or equal	GEQ	R[op3] = (R[op1]>=R[op2])
9	Equal	EQL	R[op3] = (R[op1]==R[op2])
10	Maximum	MAX	R[op3] = max(R[op1],R[op2])
11	Minimum	MIN	R[op3] = min(R[op1],R[op2])
12	Absolute value	ABS	R[op3] = absolute(R[op1])
13	Integer conversion	I2F	R[op3] = FP(R[op1])
14	Bitwise AND	AND	R[op3] = R[op1] & R[op2]
15	Bitwise OR	ORR	R[op3] = R[op1] | R[op2]
16	Logical shift left	SHL	R[op3] = R[op1]<<R[op2]
17	Logical shift right	SHR	R[op3] = R[op1]>>R[op2]
18	Memory load	MLD	R[op3] = Dmem[address]
19	Memory store	MST	Dmem[address] = R[op1]
20	Event generation	EVC	Event generated when R[op1]!=0

Event-Capture Unit (EVC)
Since the NPEs are the representation of silicon neurons in SENeCA, they will produce spikes as output. These spikes are represented as events that are captured by the Event Capture Unit (EVC), which is present in every NCC and is connected to all neurons in it. This unit then converts these spikes to the form of Address Event Representation [108], which can carry an optional value field.

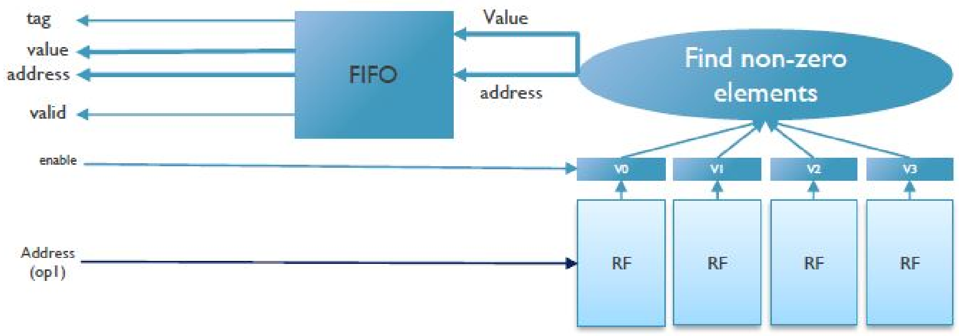


Figure 3.7: Block diagram of the Event Capture Unit

The EVC is illustrated in Figure 3.7. Each NPE can receive an instruction to emit an event (instruction number 20 in Table 3.1). If, at the time of execution of this instruction, the content of register op1 is not zero, then the EVC captures the value. That value and the ID of the source NPE (a simple unsigned integer) are sent to the ECU in the form of a message. In addition, a tag can also be included in the message. A starting value for this tag and an incremental value (to be added every time this instruction is executed) can be set by the user, allowing the actual neuron firing the event to be identified, since the NPE ID alone is not enough if time-multiplexing is used. This message then enters the FIFO buffer of the EVC. If this FIFO buffer is not empty, an interrupt is then issued to the controlling Ibex core, allowing it to respond to the neuron firing event. If multiple NPEs are activated in a single cycle, only one interrupt is issued, and the Ibex core can respond to the entire queue.

Loop Buffer
Anytime an event enters the Axon Messaging Interface, multiple neurons may need to be updated. The number of neurons can easily reach several hundred, and due to time multiplexing, each physical neuron must perform the same set of instructions repetitively in a loop. Involving the Ibex core to continuously run the NPE loops is very inefficient, as it needs to fetch the new instruction from the instruction memory every time. Furthermore, since the NPEs notify when they have finished an instruction set by way of interrupts, a lot of context switching will be needed, further decreasing efficiency. This problem is addressed by the mechanism called the Loop Buffer. SENeCA's loop buffer is a small memory made with registers (to save energy), and it is directly connected to the NPEs. These registers are used to store a local copy of the NPE instructions to be processed for a specific number of loops. This loop buffer is controlled by the Ibex core and triggers it for input events.

3.2.5. Network on Chip (NoC)

To connect the NCCs in a multicore version of SENeCA and deliver the spike events, a minimalistic mesh Network on Chip (NoC) is being developed at the time of writing with several important features. Among the most important are: 1) support of multicasting, and 2) support of variable-length packets.

Multicasting support is implemented by the use of source-based addressing. Compared to the destination-based addressing system used in many other neuromorphic architectures, the source neuron is specified in the packet containing the spike. The NoC router then redirects this message to its destination, based on a routing table stored in the router. This routing table specifies the connectivity of the neurons. Studies conducted by IMEC show that the network can be mapped with a relatively small routing table for small neurons. While this approach might not be sufficient for some applications (especially those requiring a large network), as some of the NCCs might receive an unwanted package, a filtering mechanism is implemented in the AMI. This filter will remove unwanted packages.

3.2.6. Synthesis Results

In this section, the area measurements are provided for a single-core (NCC). This information can be used to predict the area of an instance of SENeCA with multiple cores. Since static (leakage) power tends to be significant for neuromorphic architectures, a low-leakage FDX-22nm technology provided by GlobalFoundries was used to synthesize. Table 3.2 shows the resulting area of components of an NCC when synthesized using Cadence Genus. This instance has 8 NPEs, 2MB of data memory, and 128KB of instruction memory while using a target clock frequency of 500MHz.

Table 3.2: Area consumption of the main components of a single NCC.

Module	Area(kμm2)	Area (%)
AMI	12	2.2
Ibex core	23	4.2
Inst memory	28	5.1
NCP	38	6.9
Data memory	443	80.4
Total	551	100

3.3. Comparison of SENeCA with Other Architectures

Table 3.3 compares SENeCA with other neuromorphic processors.

Table 3.3: Area of SENeCA compared to other comparable neuromorphic chips.

Architecture	Area(mm2)/Core	Memory(Mb)/Core	Technology
Loihi [22]	0.41	2	Intel 14nm
Loihi2 [75]	0.21	1.5	Intel 4nm
TrueNorth [3]	0.1	0.1	Samsung 28nm
µBrain [95]	1.42	0.15	TSMC 40nm
SENeCA [109]	0.55	2.1	GF 22nm
SpiNNaker [78]	101.64	0.032	UMC 130nm

3.3.1. With SpiNNaker

SpiNNaker is perhaps the architecture that has the highest level of similarity to SENeCA [79]. SpiNNaker contains several ARM cores as processing units which are connected through an advanced star-type multicasting network. The network is packet-switched, asynchronous, and contains only one router. The second version of SpiNNaker added several arithmetic processing units for acceleration, as well as a new power management system in the GF22nm technology node of Global Foundries [68]. On the other hand, SENeCA has simpler processing units, which are smaller (based on the open-source RISC-V processor) and are not used for event processing. Also, in contrast to the multicasting network of SpiNNaker, SENeCA has a low-overhead mesh-type multicasting Network on Chip (NoC) for sparse event-based computation, which has reduced functionality compared to SpiNNaker's network, along with optimized accelerators. Also, the purpose of both architectures is different, with SpiNNaker being geared more for brain research, as opposed to SENeCA's focus on having both hardware and software for innovations in EdgeAI applications.

3.3.2. With IBM TrueNorth

Compared to the first commercial neuromorphic chip, IBM TrueNorth [3], it can be said that the TrueNorth neuron update is more energy efficient. This is because TrueNorth's core emulates exactly 256 neurons. Each neuron has 256 input synapses, organized in a crossbar architecture to determine connectivity. A single output axon is also connected to 256 other neurons in another core. Also, TrueNorth uses a plain mesh packet-switched network, which is unicast, as opposed to SENeCA's NoC. Another similar architecture, μbrain [95] goes further in the optimization of the processing core, and allows for application-specific IP with ultra-low power consumption.

3.3.3. With Intel Loihi

Compared to TrueNorth, Intel Loihi's cores are less flexible [22]. The interconnect, meanwhile, is a simple uni-cast packet-switched mesh. Also, one defining feature of Loihi is its ability to do on-chip learning with a bio-inspired learning algorithm, as opposed to other architectures, including SENeCA. The cost of this flexibility and added learning feature is the higher energy of a single neuron update (around 80 pJ). Loihi 2 [75] takes this a step further and packs more neurons and synapses in a die, using a better technology node (Intel4). Additionally, it features programmable neurons with microcode, a feature also available in SENeCA. Loihi 2 also supports a specific kind of bio-inspired learning algorithm, similar to its predecessor.

4 Software Architecture and Implementation

To measure the capabilities of SENeCA and perform a meaningful and fair comparison with other similar neuromorphic architectures, a test program must run on the different architectures and measurements must be made on the metrics of interest. This is known in the field of computer architecture as benchmarking. However, as Davies pointed out [20], a widely adopted standard benchmarking suite for neuromorphic processors is yet to be developed. As such, it is considered of interest to the SENeCA developers to perform benchmarking with an application that will typically be deployed on it in the future. As SENeCA is still in the development phase at the time of writing, the only possible way to perform benchmarking is through simulation. In this section, the application that is used to benchmark SENeCA is described, as well as the details of the implementation of that application on SENeCA. In particular, we explain how the unique components of SENeCA are leveraged to have an implementation that is as efficient as possible.

4.1. Keyword Spotting

In his article on benchmarks for neuromorphic processors, Davies [20] mentions several programs that could be included in a future benchmark. The programs mentioned include programs that are similar to real-world applications and programs that can represent algorithmic primitives that have little standalone value. The metrics that would be of interest include time-to-solution (latency), energy-to-solution, throughput, and accuracy. The first application mentioned in the hypothetical ”SpikeMark” suite is a program to classify spoken keywords, by using a specified pre-trained deep neural network. This application will be used as the main program for this study. The application in question is based on the study by Blouw et al. in [14].

There are a few reasons why this application in particular was chosen, listed below.

- A spoken keyword classification program can benefit greatly from reduced power consumption since it will likely be implemented on handheld and/or embedded devices. As such, it would be a typical application for SENeCA.
- The network used for classification is relatively simple, greatly reducing implementation time since SENeCA, especially its Software Development Kit (SDK), is still in development. Furthermore, the implementation of the NoCs (mentioned in Chapter 3) that allow the multicasting of data is still ongoing at the time of writing. This makes the cores only able to send a single designated core, precluding the use of more complicated networks such as the one used in [17]. Furthermore, the lack of a physical chip prevents the implementation of a system such as the one described in [92].
- As mentioned in Chapter 3, the SDK of SENeCA is not yet finished, preventing the integration into benchmarking suites such as SNABSuite [77] and Nengo [11] presented in Chapter 2. Therefore, only a single workload was used in this study, while benchmarking with other workloads is planned at the time of writing.
- As access to other neuromorphic hardware was not available at the time of writing, actual benchmarking was not possible with several platforms. However, since the program has already been run on several platforms, including Intel Loihi [14], the data provided in the study will be used instead as comparisons for SENeCA instead.
- The source code to implement the program on the platforms mentioned in the study is publicly available, allowing easy replication on SENeCA.

Keyword spotting revolves around monitoring an audio stream to detect a keyword of interest, such as the popular ”Hey Siri”, or ”Ok Google”. This particular task is useful for neuromorphic processors because of its low latency requirements to process audio in real time, its potential benefit from improvements in energy efficiency, and its numerous applications. In [14], the program was run on several different platforms as follows:

1. CPU : Intel Xeon E5-2630.
2. GPU : Nvidia Quadro K4000.
3. Jetson TX1: Supercomputer on a module, built around the Nvidia Maxwell™ architecture, has 256 CUDA cores.
4. Movidius NCS (Neural Compute Stick): is powered by the Intel® Movidius™ Myriad™ 2 vision processing unit (VPU).
5. Intel Loihi (Wolf Mountain Board) : explained in Chapter 1.

In this study, SENeCA will be added to the list.

The application takes in an audio waveform corresponding to an utterance and then predicts a sequence of characters that is in the waveform. Then it is determined whether the utterance contains the keyword of interest. To find the features in the waveform, it is necessary to pre-process the data. This preprocessing involves a Fourier transformation and a discrete cosine transformation on an overlapping series of windows to compute the Mel-frequency clepstral coefficients (MFCC). MFCCs are coefficients that collectively make up the Mel-frequency cepstrum (MFC), a representation of the short-term power spectrum of a sound. MFCCs are commonly used as features in speech recognition systems [36]. The windows of adjacent MFCC features are then combined into frames of 390 dimensions each, and each frame is fed into the network. Each frame corresponds to a stride of 10 ms, which means that, for real-time processing, at least 10 frames need to be processed per second.

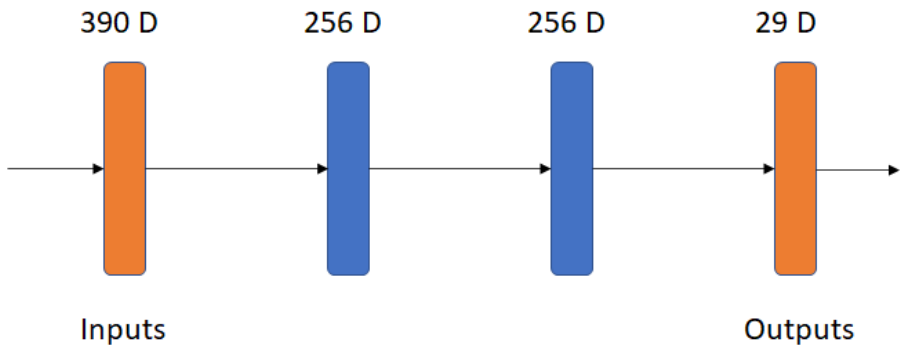


Figure 4.1: Network topology for the keyword spotter DNN. All layers are fully connected (FC). Adapted from [14]

The DNN for the keyword spotter has 3 layers, all of which are fully connected. The input is the frame of 390 dimensions, while the first two layers are 256-dimensional hidden layers. The third and final layer is used to predict a 29-dimensional output vector, which corresponds to a probability distribution over alphabetical characters. Each layer has its own pre-trained set of weights and biases. Furthermore, every layer performs the ReLU activation function for its output. Figure 4.1 illustrates this network. After processing one frame, the next one is fed into the input, and the prediction vectors are collapsed by merging repeated characters and stripping out special characters and spaces.

4.2. Reproduction and Simplification of DNN Model

To implement the neural network described above, all of the operations involved in the process have to be broken down into simple arithmetic calculations, since SENeCA does not have dedicated APIs to simplify the implementation of layers on it at the time of writing. To have a closer look at the operations that occur in the layers, visualization was done using TensorBoard [66]. This tool allows for a deeper look inside the structures that TensorFlow provides.

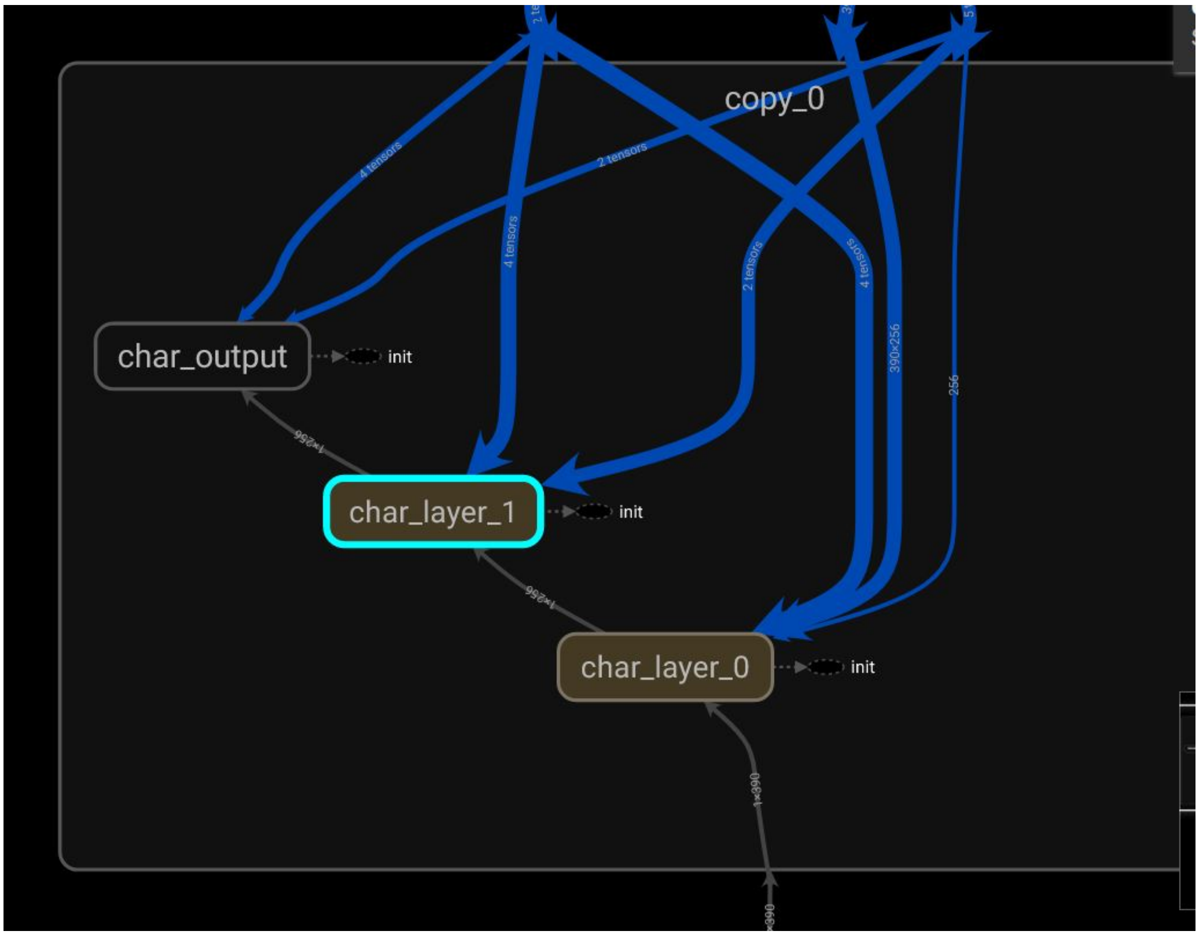


Figure 4.2: TensorBoard visualization of the main network. The rounded rectangles represent the layers, while the arrows represent the flow of data.

Figure 4.2 shows the visualization of the network. The rounded rectangles represent the layers, the gray arrows represent the flow of data between the layers, and the blue arrows represent the initializations of the layer parameters. As mentioned before, layer 1 receives a vector of 390 data points, while layers 2 and output receive a vector of 256 data points from the previous layers, shown in the figure as the gray arrows. Since the network is pre-trained, the weights and biases of each layer are initialized by TensorFlow without training, which simply means that they are loaded from memory. The sizes of these matrices to be stored in memory are shown in Tables 4.1 and 4.2, for the weights and biases, respectively. As the original implementation has the data stored in IEEE 754 floating point format, while the implementation in SENeCA is in the BrainFloat16 format, both sizes are shown.

Table 4.1: Dimensions and sizes in memory of the weight matrices of the 3 layers. Sizes are denoted in KBytes.

Layer	Weight matrix dimension	Datapoints	Size (float32)	Size (bfloat16)
Layer 1	390 x 256	99840	399.36	199.68
Layer 2	256 x 256	65536	262.144	131.072
Output	256 x 29	7424	29.696	14.848

Table 4.2: Dimensions and sizes in memory of bias vectors of the 3 layers. Sizes are denoted in KBytes.

Layer	Bias matrix dimensions	Datapoints	Size (float32)	Size (bffloat16)
Layer 1	256 x 1	256	1.024	0.512
Layer 2	256 x 1	256	1.024	0.512
Output	29 x 1	29	0.116	0.058

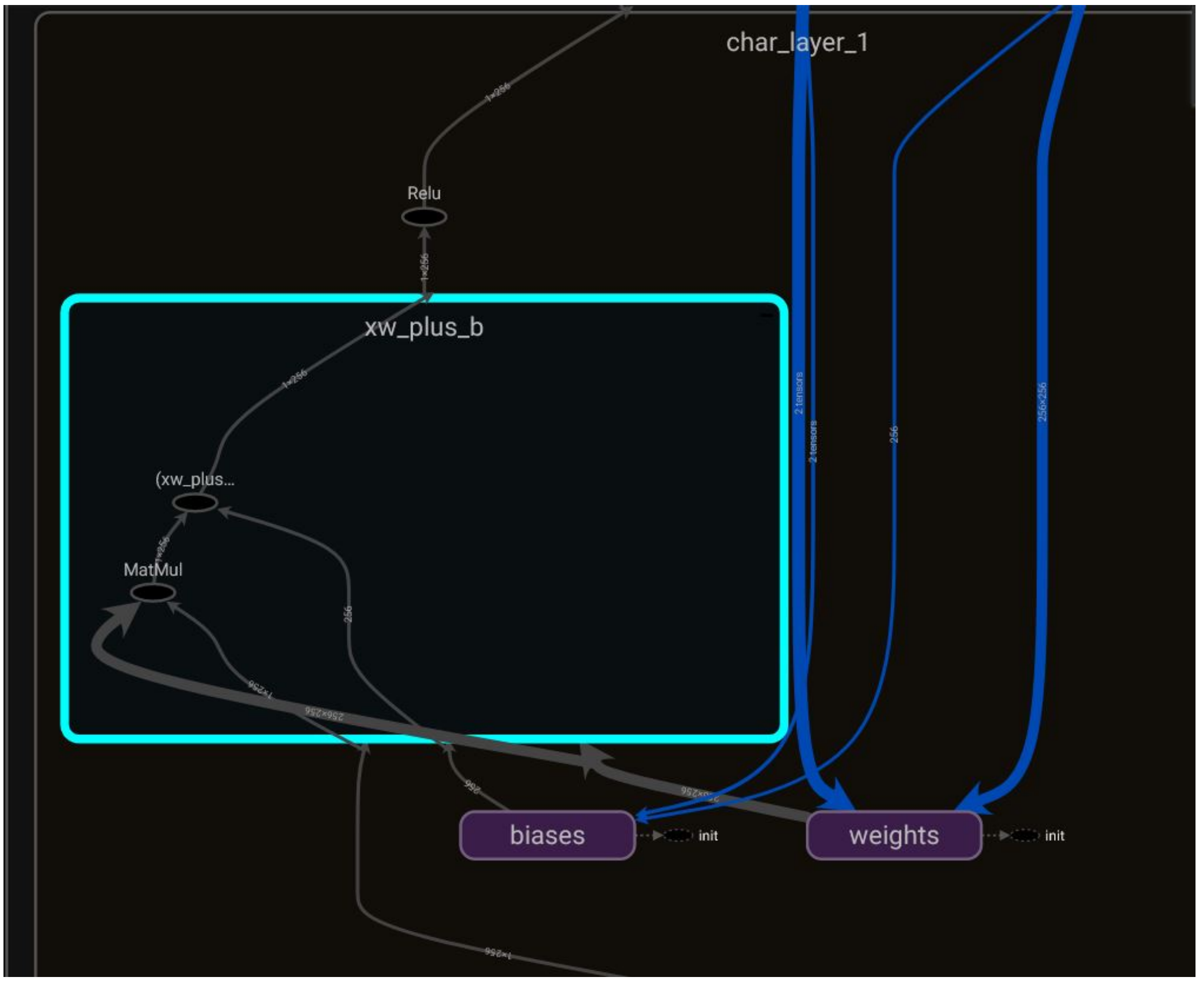


Figure 4.3: TensorBoard visualization of the second layer. The two main operations taking place, xw_plus_b and Relu, are shown here.

In Figure 4.3, the internal structure of Layer 2 is shown. As the other 2 layers are more or less similar because they only differ in the sizes of the matrices, this layer will be used to explain the operations for all of the layers. The highlighted rounded rectangle, labeled xw_plus_b, is the main multiply and accumulate operation and also includes the bias addition at the end. The MatMul operation, the lower left ellipse, is the matrix multiplication operation that multiplies the input vector with the weight matrix. As can be seen in the figure, the ellipse receives an input of 256x1, which is the input vector received from layer 1, and a 256x256 matrix, which is the weight matrix. The output of this operation then goes to the next ellipse, labeled xw_plus_b as well. This operation adds the bias vector, here of size 256x1, to the resulting vector of the MatMul operation. The result of this bias operation goes out of the highlighted rounded rectangle and goes into the ellipse labeled Relu. This applies the activation function known as Rectified Linear Unit (ReLU) to the results, essentially converting all negative values to zeros while keeping the positive ones. The resulting 256-element vector is the final output of layer 2, which goes to the output layer (not seen in the figure).

This program developed in Python will be used as a baseline to compare the accuracy of the SENeCA implementation developed in the next section, as we unfortunately do not have access to other platforms, including Loihi, at the time of writing. The test data provided with the source code of [14] includes 192 matrices, each representing an audio recording of different lengths. The implementations on SENeCA will be mostly run with one input vector, namely the first frame of the first audio recording.

4.3. Implementation on SENeCA

The implementation of the keyword spotting neural network on SENeCA will be explained in this section. As this application is one of the first applications of an end-to-end program on this platform, the implementation was done in several iterations. Each iteration will utilize a different optimization technique, utilizing the components in SENeCA's NCC to accelerate the neural network operations. The performance of each iteration will also be individually analyzed. Therefore, the impact of each optimization on performance can be discerned, both in terms of execution time and power consumption. In this chapter, the discussion will be limited to the first two iterations. These will serve as the baseline program for further optimizations.

4.3.1. RISC-V Only Implementation

This implementation is the most basic, as it only uses the RISC-V Ibex core to perform all of the operations required, without any help from the accelerators. As the RISC-V processor only has a single core, all operations are done sequentially. Also, since the core is configured for 32-bit words, the operations will mainly involve floating point formats for the weights, biases, and input data. This version was also used to understand how to program SENeCA.

Initialization and Main Function

The source code for the main loop of the program is shown in Listing 4.1. The operations involved in the neural network calculations do not take place in the main function; instead, they are implemented in the interrupt handler. Therefore, the main loop consists mainly of initializations and the main infinite loop. Initializations include the reset of neurons (line 7) and the enabling of interrupts (lines 5 and 13). Other than that, a timer function prints out statements every 5ms to indicate the passage of time.

The variables and constants used for the calculations are also declared here. This source code is for the first layer, hence the dimensions of 390 x 256. The variable named neuron_states is used to store the temporary values of the neurons as their values are updated. The counter variable is there to keep track of which input is being processed at the moment. The weights and biases variables, on the other hand, are initialized in a different file and are omitted due to their size; here they are declared with the "extern" keyword. The union is used to process the conversion between 32-bit data (used by the AMI) and float32.

Listing 4.1: Source code for the main loop and initializations

```
1
2  #define NUM_ROWS     256
3  #define NUM_COLS     390
4
5  typedef union {
6      uint32_t i;
7      float f;
8  } u;
9
10
11 static float neuron_states[NUM_ROWS];
12 static int16_t counter = 0;
13 extern float weights[NUM_COLS][NUM_ROWS];
14 extern float biases[NUM_ROWS];
15
16 int main(int argc, char **argv) {
17
18     // enable global interrupt
19     csr_write_mstatus(0x8);
20
21     //Reset output neurons
22     reset_neurons();
23
24     // Enable periodic timer interrupt
25     timer_enable(0x5000, f);
26
27     // Enable AMI Interrupt
```

```
    ami_enable();

    uint64_t last_elapsed_time = get_elapsed_time();

    while (1) {
        uint64_t cur_time = get_elapsed_time();
        if (cur_time != last_elapsed_time) {
            last_elapsed_time = cur_time;
            if (last_elapsed_time & 1) {
                puts("Tick!");
            } else {
                puts("Tock!");
            }
        }
        WFI();
    }
    WFI();
    return 0; /* Return from main will throw exception! */
}
```

Axon Messaging Interface (AMI) Interrupt Handler
As mentioned in Chapter 3, the Axon Messaging Interface (AMI) handles communication between the NCCs (cores) of SENeCA, as well as with the outside world. In this benchmark, since it is run on a simulation, the input data is fed via the AMI to the first layer which runs on core 1. This is done by the testbench software, which reads the input data from a file and sends the input through the AMI. In this implementation, the input data are stored in fp32 format, and the width of the AMI bus is 32 bits, so only one datapoint can be sent at a time. Each time a data point is sent to a specific core, an interrupt is triggered in that core's Ibex processor.

Listing 4.2 shows the source code for the AMI interrupt handler that is called if a datapoint is received. The ami_msg_recv() function reads the register that contains the datapoint received through the AMI and returns it as a 32-bit variable. The union u is used to convert the 32-bit variable to float32, and depending on the value of the datapoint, the program proceeds to update the neuron states with benchmark_update, or execute the final operations with benchmark_post(). The input denotes an end-of-stream event if it is all ones (0xFFFFFFFF) or if its float value is more than 60000.

Listing 4.2: Source code for the AMI interrupt handler

```
ATTR_INTR void ami_handler(void) {

    u union_float;
    uint32_t event = ami_msg_recv();
    union_float.i =event;

    //Respond to interrupt depending on the value of an event
    if ((event == EVENT_EOS)|| (union_float.f > 60000.0)) {
        //Execute final calculations
        benchmark_post();

    } else {

        //Multiply the event with the weight vector and update the neuron values
        benchmark_update(event);
    }
}
```

Matrix Multiplication
Implementing matrix multiplication is quite straightforward since it is done sequentially. Every time there is an interrupt from the AMI that is not an EOS event, this function is called. The 32-bit event is converted to fp32 format, which is then used to update the neuron values. This is done by multiplying the incoming event with the corresponding row in the weight matrix and adding the resulting vector to the vector that holds the current neuron values. Since only one core is used, a for loop is used to iterate through the neurons, updating all of them sequentially. The source code is shown in Listing 4.3.

Listing 4.3: Source code for matrix multiplication

```c

void benchmark_update(uint32_t ev) {
    //Union to convert a 32-bit variable to floating point
    u union_float;
    union_float.i = ev;

    //Iterate over all neurons and update them
    for (int i = 0; i < NUM_ROWS; i++){
        //Multiply and accumulate (MAC) operation
        neuron_states[i] += weights[counter][i] * union_float.f;
    }
    counter++;
}
```

Bias Addition, Activation Function, and Transmission

Listing 4.4 shows the source code for the final phase of the calculations (bias addition and activation function), and the process of sending the neuron states to the next layer via the AMI.

Listing 4.4: Source code for final part of the calculations

```c

void benchmark_post(void) {

    //Variables to send the values and store the FIFO occupancy
    uint16_t low_var;
    uint16_t high_var;
    uint32_t ami_irq;

    // Add biases to the neuron states
    for (int i = 0; i < NUM_ROWS; i++) {
        neuron_states[i] += biases[i];
    }

     //Implementation of rectified linear unit function
    for (int i = 0; i < NUM_ROWS; i++) {
        if(neuron_states[i] < 0.0){
            neuron_states[i] = 0.0;
        }
    }

    //Iterate through all the neurons and send their values
    for (int i = 0; i < NUM_ROWS; i++){
        //Use a union to convert float to 32-bit data
        u union_float;
        union_float.f = neuron_states[i];

        //Split the data into two parts to send
        low_var = (uint16_t)union_float.i;
        high_var = (uint16_t)(union_float.i >> 16);

        //Send the lower 16-bit data
        event_t ev = create_event(0, low_var);
        ami_msg_send(ev.r);

        //Send the upper 16-bit data
        event_t ev2 = create_event(0, high_var);
        ami_msg_send(ev2.r);

        //Read the FIFO buffer occupancy to make sure it does not overflow
        ami_irq = ami_buffer_read();
    }

    //Send the End of Stream (EOS) event
    event_t ev = create_event(0, EVENT_EOS);
    ami_msg_send(ev.r);

}
```

This process is divided into 3 main loops, each iterating over the neurons. As this code is for the second layer, there are 256 iterations for each loop. The first loop adds the bias values to the neuron values which are obtained using the MAC operations. The second loop implements the Rectified Linear Function (ReLU) function, converting all negative values to 0 while keeping all positive values. The final loop sends the neuron values to the next layer through the AMI. While the AMI is configured to send at most 32-bit data, at the time of development of this baseline software, sending was configured to 16-bit address and 16-bit data, so each value has to be divided into two. These two parts are the 16-bit upper and lower bits, which are sent separately. After sending two halves, the FIFO occupancy rate is read to ensure that the buffer does not overflow due to the difference in processing time between sending and receiving. At the end of the loop, an End of Stream (EOS) event is sent, signaling the end of the data stream and letting the next layer know that it can start processing the data.

4.3.2. Baseline NPE Implementation

The second version of this benchmark utilizes the SENeCA NPEs to accelerate neural network operations. As the RISC-V-only implementation does not involve any component of SENeCA that makes it a neuromorphic processor, this version is the true baseline to be used for the optimizations that will be explored in later chapters. As explained in Chapter 3, the SENeCA NPEs have a common kernel that functions as an instruction memory for them. The RISC-V core can write instructions to this kernel, making the NPEs all perform this same set of instructions in a Single Instruction Multiple Data (SIMD) fashion. There are two sets of instructions used mainly in this implementation to execute the neural network calculations, the MAC and Bias sets. From the available instructions listed in Table 3.1, a Multiply and Accumulate program was written, and the instructions are listed in Table 4.3. While instruction No. 1 is not technically an instruction done by the NPE, as it is the RISC-V core that writes the input data to the register file directly, it is included for completion's sake. The operations involved are quite straightforward, with the required operands being loaded first, multiplication and addition being performed, and finally the result is written back into the memory.

Table 4.3: Program to perform multiply and accumulate operations by the NPE. Five NPE registers are used out of the 16 available.

No	Instruction	Operand 1	Operand 2	Target
1	Write to REG	input	N/A	eval_reg
2	MLD	ptr to weight	N/A	weight_reg
3	MLD	ptr to state	N/A	state_reg
4	MUL	eval_reg	weight_reg	mul_res_reg
5	ADD	mul_res_reg	state_reg	new_state_reg
6	MST	new_state_reg	N/A	ptr to state

Table 4.4, meanwhile, lists the operations required to perform the bias addition, as well as the application of the ReLU function. Similarly to Table 4.3, instruction 1 is included even though it is not executed by the NPE. Operations 2 and 3 load the required variables into the NPE registers. The bias addition is performed by Instruction 4. The ReLU function is implemented by Instructions 5 and 6. Since conditionals are not supported by the NPEs instruction set, a comparison is performed to find out if the neuron state is greater than zero, which results in a 1 if true and 0 otherwise. The neuron state is then multiplied by the result, resulting in the same result as the application of ReLU with a conditional.

Table 4.4: Program to perform bias addition and activation function by the NPE. Six NPE registers of the 16 available are used.

No	Instruction	Operand 1	Operand 2	Target
1	Write to REG	0	N/A	zero_reg
2	MLD	ptr to bias	N/A	bias_reg
3	MLD	ptr to state	N/A	state_reg
4	ADD	bias_reg	state_reg	add_res_reg
5	GTH	add_res_reg	zero_reg	comp_res_reg
6	MUL	new_state_reg	comp_res_reg	mul_res_reg
7	MST	mul_res_reg	N/A	ptr to state

Initialization and Main Function

Similarly to the previous version, the main loop shown in Listing 4.5 consists only of a timer function that prints statements to indicate the passage of time and that the simulation has not stopped. However, since this version uses NPEs, several commands are added to activate the components related to the NPE. Lines 14 and 15 enable the interrupt from the NPEs, although they are not used in this version. Lines 18 and 19 are reset and enable all available NCPs (including the NPEs).

Listing 4.5: Source code of main function and variable initializations

```
 1
 2
 3 #define DRAM_START  0x10000
 4 #define NUM_ROWS    256
 5 #define NUM_COLS    390
 6
 7
 8 int main(int argc, char **argv) {
 9
10     puts("Neuron CoPro test, layer 1");
11
12     // enable general interrupt
13     csr_write_mstatus(0x8);
14
15     // enable interrupts of neurons
16     csr_set_bits_mie(BIT(18));    // spikes_handler: read_msg();
17     csr_set_bits_mie(BIT(19));    // neuron_proc_done_handler;
18
19     init_hw();
20     reset_all_ncp();
21     enable_all_ncp();
22
23     ami_enable();
24     INIT_REG_FILE();
25
26     uint64_t last_elapsed_time = get_elapsed_time();
27
28     while (1) {
29         uint64_t cur_time = get_elapsed_time();
30
31         if (cur_time != last_elapsed_time) {
32             last_elapsed_time = cur_time;
33
34             if (last_elapsed_time & 1) {
35                 puts("Tick!");
36             } else {
37                 puts("Tock!");
38             }
39         }
40         WFI();
41     }
42     return 0;
43 }
```

AMI Interrupt Handler

This version's handler of the AMI interrupt is identical to Listing 4.2, hence here it is omitted for brevity's sake.

NPE Loop

Listing 4.6 is the main loop that performs the neural network operations; therefore, all statements that involve NPEs are included here. For clarity, biasesV_g, weightV_g, and stateV_g are the arrays that contain the biases, weights, and neuron states, respectively. Meanwhile, beginning_of_weightsV and beginning_of_statesV are pointers that point to the beginning of the array to be processed in that specific loop.

The first loop (lines 10 to 13) converts the biases that are initialized as floating-point numbers to the bfloat16 format. This inefficiency will be optimized in later versions, since the biases can simply be declared and initialized as bfloat16 instead of float32. This eliminates the need for conversion. The main loop for matrix multiplication starts at line 10. It iterates through every input, of which there are 390, since this example is from the first layer. If the input value is zero, it is skipped to decrease execution time. The function KERNEL_SENECA writes the input to the NPE register and writes the required commands. This function will be explained more thoroughly in the next subsection. The inner loop iterates through neurons as every neuron needs to be updated for each input, but because there are 8 NPEs, the number of iterations is divided by 8. For each iteration, the pointer to the beginning of the states and weights to be processed is adjusted. The function DO_LOOP then commands the NPEs to execute the instructions that have been written by KERNEL_SENECA.

The second loop (starting at line 20) takes care of the bias addition and activation function. This loop iterates through all neurons, which takes fewer iterations than the previous version due to the NPEs. Again, the number of iterations is the number of neurons divided by 8. The KERNEL_SENECA_B function writes the required instructions to the kernel, while DO_LOOP_B executes them.

Listing 4.6: Source code of NPE Loop of Version 2

```
1
2
3  void run_kernel_seneca()
4
5  {
6      uint32_t last_miram_1stHalf_addr_used = 0x00;
7      uint32_t last_miram_2ndHalf_addr_used = 0x07;
8      for (uint32_t k = 0; k < NUM_ROWS; k++){
9
10         //Convert biases to bfloat16
11         FLOAT2BF(&biases[k], &biasesV_g[k]);
12     };
13
14     for (uint32_t  i = 0; i < NUM_COLS; i++)
15     {
16         if(input[i] =! 0){
17             KERNEL_SENECA(input[i]);
18             for(uint32_t k = 0; k < (NUM_ROWS/8); k++){
19                 beginning_of_weightsV = (void*)&weightV_g[(i*256)+(k*8)];
20                 beginning_of_statesV = (void*)&stateV_g[k*8];
21                 DO_LOOP(beginning_of_weightsV, beginning_of_statesV);
22             };
23         }
24     }
25     for (uint32_t  i = 0; i < (NUM_ROWS/8); i++)
26
27         {
28             beginning_of_statesV = (void*)&stateV_g[i*8];
29             KERNEL_SENECA_B();
30             beginning_of_biasesV = (void*)&biasesV_g[i*8];
31             DO_LOOP_B(beginning_of_biasesV, beginning_of_statesV);
32         }
33
34 }
```

Matrix Multiplication Instruction Set

Listing 4.7 shows the source code for the loop that executes the matrix multiplications. As the operations are done by the NPEs, the code shown here contains commands to the RISC-V core to write instructions to the kernel. The whole process is divided into two functions since there are commands that change only every time a process starts for a new input (every iteration of the loop starting at line 10 of Listing 4.6), and there are commands that change for every iteration of the loop starting at line 20 of Listing 4.6. This division prevents commands from being overwritten unnecessarily.

Before going into the explanation of the commands to write to the kernel, an explanation about the variables last_miram_1stHalf_addr_used and last_miram_2ndHalf_addr_used needs to be made. These two variables will be referred to as the miram_address variables. As explained in Chapter 3, the commands to store and load from memory need two registers, one to store the address and one to store the command itself. To organize the commands and addresses, the kernel is divided into two sections, one for the addresses (1st section) and one section (2nd section) for the commands. The variables above allow the user to write a command (and an address, if necessary) to empty registers, with the variable last_miram_1stHalf_addr_used for addresses, and last_miram_2ndHalf_addr_used for commands. The corresponding variable is then incremented, allowing the next command to be written in the next empty register. In Listing 4.6, last_miram_1stHalf_addr_used was initialized as 0x00, while last_miram_2ndHalf_addr_used was initialized as 0x07. This means that the range for addresses starts at 0x00 and ends at 0x06, while commands can be written from 0x07 onward.

The function KERNEL_SENECA writes all of the commands listed in Table 4.3, except the ones related to memory. The first step taken is to declare a register for the input to be processed, eval_reg. Then the input is converted to bfloat16 and loaded into the register. The next step is to load the weight into the memory register (line 10). However, since the weight value is specific to each neuron, it is not written here. Instead, space is made for the MLD command, both for the address and the instruction, by incrementing the two miram_address variables. A weight register is also declared. The same process is done for the neuron state (line 14). The rest of the function writes the commands that are the same for all neurons within one input iteration. First, the register that holds the multiplication result (mul_res_reg) is declared. Then, the command to multiply the input with the weight and store the result to the mul_res_reg is written. Second, the register that holds the new neuron value new_state_reg is declared, and the addition command is written that adds the multiplication result to the old neuron state. Finally, the command is written to write the new neuron state back to the memory.

The DO_LOOP function, meanwhile, adds the commands to read the weight and neuron state values. It does that in the following way. First, the last_miram variables are reset. Then, the registers to store the data to be read and the space to store the address from which the data are to be read are declared. Then the commands to read the data are written (34 and 37). After this step, all necessary commands and addresses should be written in the instruction memory. To tell the NPEs which instructions they have to execute, the required parameters are written in a specific way, done by the statement on line 39. This includes the start, end, and number of repetitions. The first instruction is the first space in the section designated for instructions. The last instruction of this loop is stored in the variable last_miram_2ndHalf_addr_used. Before writing the loop configuration, this variable is incremented by 3, the number of instructions written in KERNEL_SENECA. As the loop buffer (explained in Chapter 3) is not used, the number of repetitions is set to 1.

Listing 4.7: Source code of matrix multiplication instruction set

```
1
2  void KERNEL_SENECA(float datapoint)
3
4  {
5
6          uint32_t eval_reg = 0u;
7          float ev_val_F = datapoint;
8          uint16_t ev_val_BF = 0;
9          FLOAT2BF(&ev_val_F, &ev_val_BF);
10     write_to_regfile(eval_reg, ev_val_BF);
11
12     uint32_t weight_reg = eval_reg + 1;
13     uint32_t weight_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
14     ++last_miram_2ndHalf_addr_used;
15
16     uint32_t state_reg = weight_reg + 1;
17     uint32_t state_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
18     ++last_miram_2ndHalf_addr_used;
19
20     uint32_t mul_res_reg = state_reg + 1;
21     append_to_miram_command(MUL, eval_reg, weight_reg, mul_res_reg);
22     uint32_t new_state_reg = mul_res_reg + 1;
23     append_to_miram_command(ADD, mul_res_reg, state_reg, new_state_reg);
24     append_to_miram_write_reg_to_mem(state_adr_addr_ptr, new_state_reg, 0x01u);
25 }
26
27 void DO_LOOP(void * beginning_of_weightsV, void* beginning_of_statesV)
28 {
29
```

```
    last_miram_1stHalf_addr_used = 0x0;
    last_miram_2ndHalf_addr_used = (MIRAM_1ST_HALF_LAST_INDEX);
    uint32_t eval_reg = 0u;

    uint32_t weight_reg = eval_reg + 1;
    uint32_t weight_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr, beginning_of_weightsV,
        weight_reg, 0x01u);

    uint32_t state_reg = weight_reg + 1;
    uint32_t state_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(state_adr_addr_ptr, beginning_of_statesV, state_reg,
         0x00u);
    last_miram_2ndHalf_addr_used += 3u;

    uint32_t loopcfg = LOOP_CONFIG_INST(1u, last_miram_2ndHalf_addr_used,
        MIRAM_1ST_HALF_LAST_INDEX+0x01u);
    WriteLoopConfig(loopcfg);

    WFI();
    }
```

Bias and Activation Function Instruction Set

The bias and activation function loop is implemented by the source code shown in Listing 4.8. Similarly to the loop for matrix multiplication, the statements listed are mainly to write the correct commands to the NPE's instruction memory to allow the NPEs to perform bias addition and activation function application properly. To avoid overwriting the instructions already written for the matrix multiplication loop, the last_miram variables are not initially set to 0x00. Instead, they are initialized to the next empty spot after the last instruction of the MM loop (0x04 and 0x13, respectively). Similarly to the MM loop, the statements are divided into two loops, which will be explained in the following.

The KERNEL_SENECA_B function writes the instructions that do the calculations (as opposed to memory access). At the beginning, the last_miram variables are initialized as mentioned above. After that, the value of 0 is written to the specific register, to be used in a subsequent operation. Then, to make space for the instruction to read the neuron state and the bias value from the memory, both last_miram variables are incremented twice. The registers to store the data are also declared. Subsequently, the commands to execute the calculations are written to the instruction memory. First, the ADD operation to add the bias to the neuron state, as well as the register to store the result. Then, to implement the ReLU function without a conditional statement, a GTH (greater than) operation and a MUL operation are carried out, as explained in Table 4.4. The registers to store the comparison result and the multiplication are also declared. Finally, the command to write the result back to the memory is added.

The DO_LOOP_B function meanwhile, writes the read commands to load the bias and neuron state from the memory. The structure is very similar to Listing 4.7's DO_LOOP. The register to store the variables to be read and the space to store the addresses are declared. Subsequently, the commands to read are written into the space reserved by KERNEL_SENECA_B. last_miram_2ndHalf_addr_used is updated to take into account the instructions written by KERNEL_SENECA_B. Finally, the loop configuration is written, with the first instruction being 0x14 (start of the instructions of the bias and activation set), the last instruction being stored by last_miram_2ndHalf_addr_used, and the repetition number being 1.

Listing 4.8: Source code of bias and activation instruction set

```

void KERNEL_SENECA_B()
{
    last_miram_1stHalf_addr_used = 0x4;
    last_miram_2ndHalf_addr_used = 0x13;

    uint32_t zero_reg = 0u;
    uint16_t ev_val_BF = 0;
    write_to_regfile(zero_reg, ev_val_BF);

    uint32_t temp_reg = zero_reg + 1;
    uint32_t temp_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    uint32_t bias_reg = temp_reg + 1;
    uint32_t bias_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    uint32_t add_res_reg = bias_reg + 1;
    append_to_miram_command(ADD, bias_reg, temp_reg, add_res_reg);
    uint32_t comp_res_reg = add_res_reg + 1;
    append_to_miram_command(GTH, add_res_reg, zero_reg, comp_res_reg);
    uint32_t mul_res_reg = comp_res_reg + 1;
    append_to_miram_command(MUL, add_res_reg, comp_res_reg, mul_res_reg);
    append_to_miram_write_reg_to_mem(bias_adr_addr_ptr, mul_res_reg, 0x01u);
}

void DO_LOOP_B(void * beginning_of_biasesV, void* beginning_of_statesV)
{
    last_miram_1stHalf_addr_used = 0x4;
    last_miram_2ndHalf_addr_used = 0x13;

    uint32_t zero_reg = 0u;
    uint32_t temp_reg = zero_reg + 1;
    uint32_t temp_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(temp_adr_addr_ptr, beginning_of_biasesV, temp_reg, 0
        x01u);

    uint32_t bias_reg = temp_reg + 1;
    uint32_t bias_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(bias_adr_addr_ptr, beginning_of_statesV, bias_reg, 0
        x00u);

    last_miram_2ndHalf_addr_used += 4u;
    uint32_t loopcfg = LOOP_CONFIG_INST(1u, last_miram_2ndHalf_addr_used, 0x13+0x01u);
    WriteLoopConfig(loopcfg);

    WFI();

}
```

Transmission Loop

This version's handler of the transmission loop is identical to Listing 4.4, hence here it is omitted for brevity's sake.

5

Methodology and Preliminary Results

After the architecture and implementation of the benchmarking software have been explained in Chapter 4, in this chapter the methodology to obtain the performance indicators of SENeCA will be discussed. Furthermore, the first two versions of the benchmarking software were also described. The results obtained by applying the methodology with the aforementioned versions of the software will also be presented. As these results do not properly show the capabilities of SENeCA, since most of the accelerators were not used, they are not meant to be taken as final, definitive results. Instead, they should be taken as baseline results and as a way of understanding the methodology more clearly. Furthermore, a section will explain a test to estimate the power consumption of individual NPE operations listed in Chapter 3, providing a basis for the estimation of the power consumption of programs prior to implementation.

5.1. Experiment Setup and Flow

This section will explain the setup used for this experiment. Unless otherwise stated, this setup will be used throughout all of the experiments run here. As mentioned previously, SENeCA does not yet exist as a physical chip, so all of the experiments were performed on simulators. The simulation runs on the Cadence© Xcelium Logic Simulator, a high-performance simulation platform for HDL-based IP and SoC verification. The power consumption analysis is performed by Cadence© Joules. This fast power estimation software delivers time-based RTL power analysis along with system-level runtimes and capacity, while still providing accurate estimations of wires and gates. The accuracy level is within 15% of signoff power. The synthesis of the SENeCA RTL is performed using the Cadence© Genus software. Meanwhile, the compilation of the C source file of the benchmarking program is done by GCC, with RISC-V configured as its target. An overview of the main software used in this research is listed in Table 5.1.

Table 5.1: An overview of the software used in this study.

Sofware	Version	Function
Joules	20.11	Performs power consumption analysis
Xcelium	18.03	Simulates RTL and C program
Genus	19.11	Synthesizes RTL design
GCC	4.8.5	Compiles C code

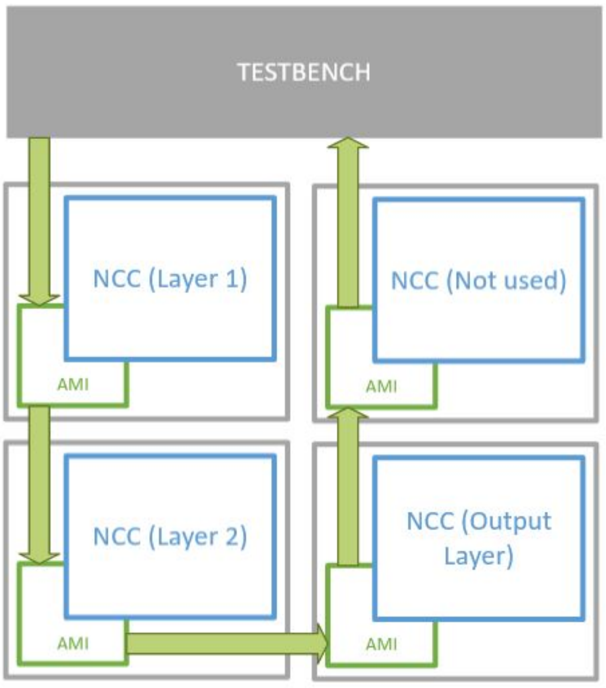


Figure 5.1: SENeCA instantiation for this study.

Meanwhile, the hardware configuration is depicted in Figure 5.1. The testbench sends the initial input values to NCC 1, which is handling layer 1. It performs the calculations, then sends the results via its AMI to NCC 2, etc. As mentioned, each core handles the data and calculations of 1 layer, but since there are 4 cores, the final core is not used and merely relays the outputs of core 3 to the testbench (bypass mode). The testbench then prints the values on the computer screen, and the results can be verified. Each core is configured to be able to send only directly to the next core and receive only from the previous core.

With the above software, experiments are done with the flow shown in the figures below.

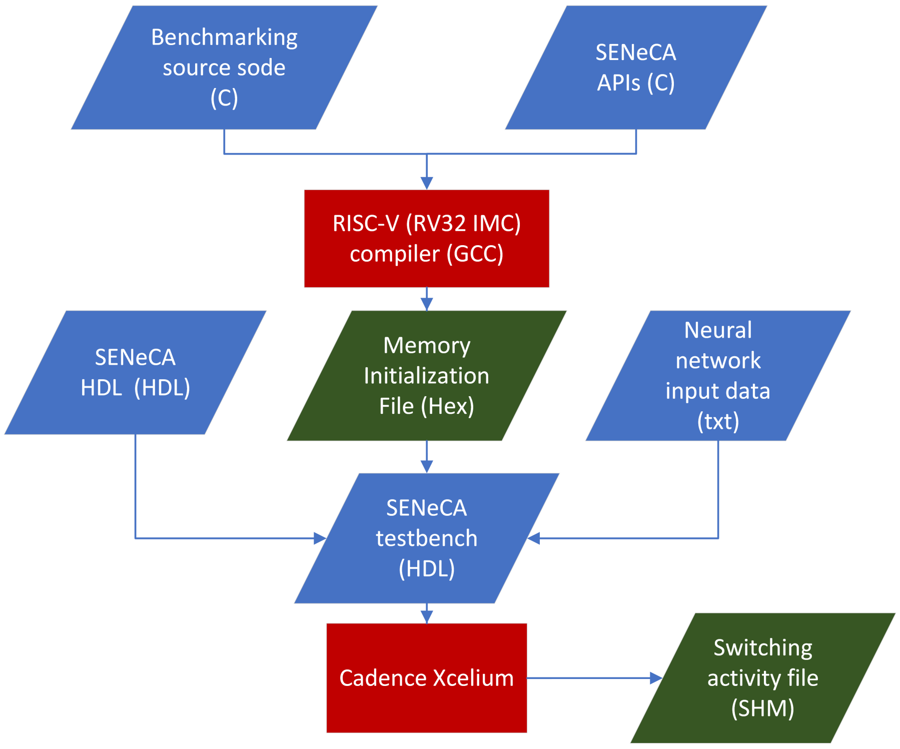

Benchmarking source sode (C)
SENeCA APIs (C)
RISC-V (RV32 IMC) compiler (GCC)
SENeCA HDL (HDL)
Memory Initialization File (Hex)
Neural network input data (txt)
SENeCA testbench (HDL)
Cadence Xcelium
Switching activity file (SHM)

Figure 5.2: Experiment flow for performance measurements.

Figure 5.2 shows the flow used to perform the simulation and the performance measurements. First, the source code for the benchmarking program is written in C, with the help of existing SENeCA APIs. They are then compiled using a GCC-based compiler for RISC-V processors, producing the memory initialization files. A testbench file is used to instantiate the SENeCA chip as the DUT, and it will also load the memory initialization files afterwards to SENeCA's RAM. After loading them, the testbench will also take in a file that contains the input values of the neural network and send them to SENeCA as input. Then, the simulator, Xcelium, takes in all the required files and runs the simulation. As it runs, it outputs a switching activity file in SHM format. SHM is a proprietary format of Cadence©, used to store simulation waveforms. To be clear, the contribution of this thesis is limited to the source code for benchmarking (in C) and the testbench, while the other software and source files were made available by IMEC.

The flow of the experiment to perform the analysis of power consumption and obtain the estimates is depicted in Figure 5.3. As this flow uses the results of the flow depicted in Figure 5.2, this flow is always carried out after the simulation is completed. The main software used in this flow is Joules, the power estimation software. It reads the switching activity file obtained earlier, as well as the SENeCA HDL. To obtain the power measurement numbers, synthesis of the SENeCA HDL needs to be performed. This is done by Genus, which is called automatically from within the Joules software. However, to do this, it needs the technology library. The technology node used here is the GF22-FDX of Global Foundries. After obtaining the netlist, Joules will perform the power estimation, whose results are average power estimations in the form of text reports and a graph showing the power consumption over time. This graph is in the SHM format, similar to the output of the previous flow.

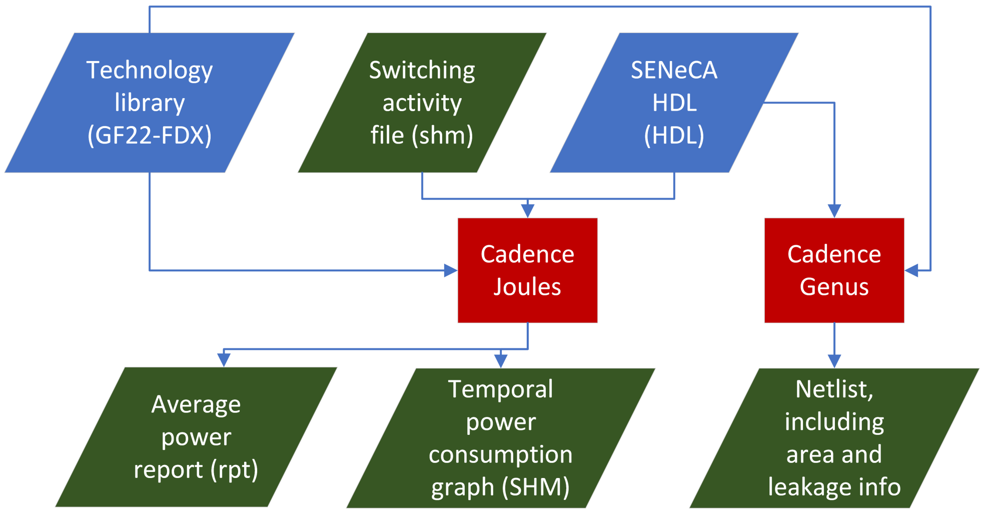

Technology library (GF22-FDX)
Switching activity file (shm)
SENeCA HDL (HDL)
Cadence Joules
Cadence Genus
Average power report (rpt)
Temporal power consumption graph (SHM)
Netlist, including area and leakage info

Figure 5.3: Experiment flow for power consumption measurements.

As mentioned above, SENeCA is meant to be a configurable and scalable design. The configurations of the SENeCA instance used as a target in this benchmarking process are summarized in Table 5.2.

Table 5.2: Parameters of the SENeCA instance used.

Parameter	Value
Number of NPEs per NCP	8
Number of NCCs (cores)	4 (1 in bypass mode)
Clock	100 MHz
RAM size	256 x 32 Kbit blocks
Opcode width	5
Number of registers in NPE	16
Number of registers in Loop buffer	32
EVC buffer size	128

Figure 5.4 shows the distribution of the chip area for a single NCC core using the parameters above.

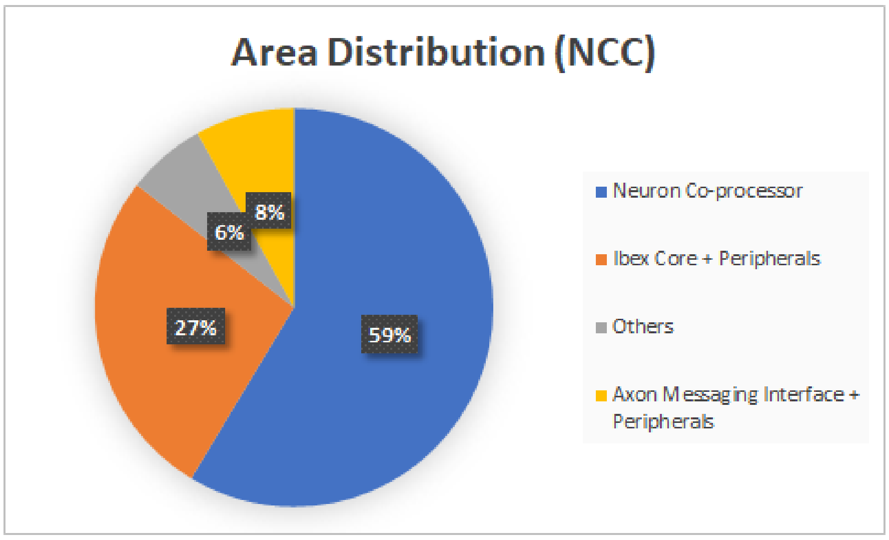
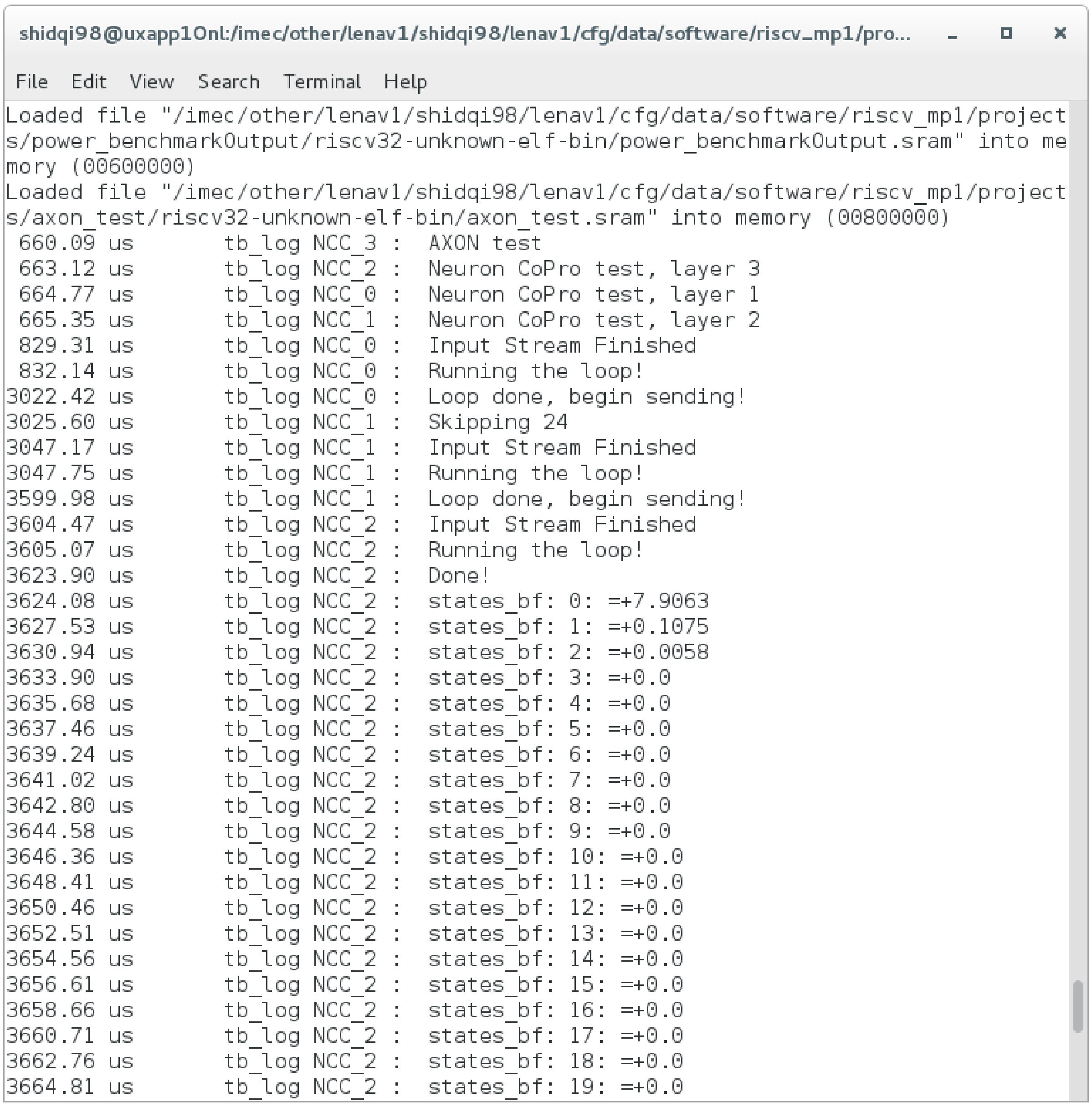


Figure 5.4: Breakdown of .

5.2. Execution Time Measurements

To measure the time it takes for SENeCA to execute the neural network described in Chapter 4, a simulation is run using the SENeCA RTL together with the compiled benchmarking program. A testbench file written in SystemVerilog is used, which configures the SENeCA chip to run the neural network by uploading the compiled binary files of the benchmarking program to each of the core's RAM. The simulation is then run. In the study on which this benchmark is based, the audio files span several seconds[14]. Initially, the simulations will be run with only one frame as input (corresponding to 10ms of audio). Communication with the user is allowed by configuring SENeCA to print text messages on the screen. Xcelium also shows a timestamp (in simulation time) of the print statements, allowing the user to know with reasonable accuracy when a specific part of the computation has been done by placing the print statements after the said part has been completed in the code.


shidqi98@uxapp10nl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/pro...
File Edit View Search Terminal Help

```
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/project
s/power_benchmarkOutput/riscv32-unknown-elf-bin/power_benchmarkOutput.sram" into me
mory (00600000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/project
s/axon_test/riscv32-unknown-elf-bin/axon_test.sram" into memory (00800000)
 660.09 us       tb_log NCC_3 :  AXON test
 663.12 us       tb_log NCC_2 :  Neuron CoPro test, layer 3
 664.77 us       tb_log NCC_0 :  Neuron CoPro test, layer 1
 665.35 us       tb_log NCC_1 :  Neuron CoPro test, layer 2
 829.31 us       tb_log NCC_0 :  Input Stream Finished
 832.14 us       tb_log NCC_0 :  Running the loop!
3022.42 us       tb_log NCC_0 :  Loop done, begin sending!
3025.60 us       tb_log NCC_1 :  Skipping 24
3047.17 us       tb_log NCC_1 :  Input Stream Finished
3047.75 us       tb_log NCC_1 :  Running the loop!
3599.98 us       tb_log NCC_1 :  Loop done, begin sending!
3604.47 us       tb_log NCC_2 :  Input Stream Finished
3605.07 us       tb_log NCC_2 :  Running the loop!
3623.90 us       tb_log NCC_2 :  Done!
3624.08 us       tb_log NCC_2 :  states_bf: 0: =+7.9063
3627.53 us       tb_log NCC_2 :  states_bf: 1: =+0.1075
3630.94 us       tb_log NCC_2 :  states_bf: 2: =+0.0058
3633.90 us       tb_log NCC_2 :  states_bf: 3: =+0.0
3635.68 us       tb_log NCC_2 :  states_bf: 4: =+0.0
3637.46 us       tb_log NCC_2 :  states_bf: 5: =+0.0
3639.24 us       tb_log NCC_2 :  states_bf: 6: =+0.0
3641.02 us       tb_log NCC_2 :  states_bf: 7: =+0.0
3642.80 us       tb_log NCC_2 :  states_bf: 8: =+0.0
3644.58 us       tb_log NCC_2 :  states_bf: 9: =+0.0
3646.36 us       tb_log NCC_2 :  states_bf: 10: =+0.0
3648.41 us       tb_log NCC_2 :  states_bf: 11: =+0.0
3650.46 us       tb_log NCC_2 :  states_bf: 12: =+0.0
3652.51 us       tb_log NCC_2 :  states_bf: 13: =+0.0
3654.56 us       tb_log NCC_2 :  states_bf: 14: =+0.0
3656.61 us       tb_log NCC_2 :  states_bf: 15: =+0.0
3658.66 us       tb_log NCC_2 :  states_bf: 16: =+0.0
3660.71 us       tb_log NCC_2 :  states_bf: 17: =+0.0
3662.76 us       tb_log NCC_2 :  states_bf: 18: =+0.0
3664.81 us       tb_log NCC_2 :  states_bf: 19: =+0.0
```

Figure 5.5: Example of a simulation run. The timestamps are displayed on the left, while the text messages indicate a specific task completed.

An example run is shown in Figure 5.5. These are basically messages printed by the SENeCA while it is running the neural network program. At the top, it can be seen that the first message, which is printed out at the start of the main function, is printed at around 660 us. This is because the simulation time takes into account the time it takes for the testbench program to load the compiled binary files into the RAM of the SENeCA cores. This time is quite long because of the size of the weights and biases, and the fact that the uploading to the memory cannot occur in parallel. However, since this time is not part of the actual computation time, it will be omitted and will not count towards the execution time. Therefore, the computation is considered to begin with the first message from one of the cores, since the print statement is put first before any other commands.

As explained in Chapter 4, there are 3 layers of neurons and the computations of each layer are executed by a separate core. The testbench software is written so that when a message is printed, the source core of that message can be identified. In Figure 5.5, the sender can be identified by the name NCC_X, where X is a number from 0 to 3. In this implementation, NCC 0 executes layer 1, NCC 1 executes layer 2, while NCC 2 executes the output layer. NCC 3 is not involved in the calculations. Also explained in Chapter 4 is that neural network operations can be divided into two major parts, the sending of the data between cores and the calculations taking place in the cores themselves. Using the timestamps of the printed statements, such as those in Figure 5.5, the transmission time and the execution time for each core can be determined. For example, it can be seen that at the 832.14 us timestamp, NCC 0 starts to execute the calculations of layer 1, and at 3022.42 us, it can be seen that NCC 0 has finished the calculations. By subtracting these times, the processing time of the calculations of layer 1 can be determined.

Since there are 3 layers involved in the benchmarking program, the total execution time will be divided into 3 parts, one for each layer. Furthermore, to know in more detail how SENeCA spends the execution time, each layer will also be divided into two: the time it takes to receive the input values from the previous layer and the time it takes to actually process the data. In this way, bottlenecks that slow the process can be identified and optimized. In summary, there will be 3 timespans measured for the neural network operations (henceforth known as Loop 1, Loop 2, and Loop 3) and 3 timespans measured for the sending/receiving between cores (henceforth known as Sending 1, Sending 2, and Sending 3). Therefore, the milestones at which the printed statements would be placed are the starts and ends of the six parts. Since the end of one part happens at the same time as the start of the next, in total there would be 7 points to be measured.

5.3. Power Measurements

As SENeCA is still in the development stage without any finished physical chips, power measurements are also made in the simulation. Joules allows power estimations to be done in 2 modes, average and time-based.

5.3.1. Average Power Estimation

Table 5.3: An example of an average power report generated by Cadence Joules©. All power values are denoted in Watts.

Instance	Cells	Pct_cells	Leakage	Internal	Switching	Total	Lvl
SENeCA Top Level	483404	100.00%	1.12E-02	3.78E-02	1.03E-03	5.00E-02	0
Core 1	120920	25.01%	2.80E-03	9.79E-03	3.70E-04	1.30E-02	1
NCP core 1	70219	14.53%	2.78E-03	8.86E-03	1.85E-04	1.18E-02	2
Instruction RAM	6	0.00%	3.61E-06	5.62E-04	4.15E-07	5.66E-04	2
Ibex Core	33199	6.87%	5.37E-06	2.66E-04	1.65E-04	4.37E-04	2
AMInterface	9750	2.02%	2.26E-06	4.89E-05	4.83E-08	5.12E-05	2
Mux/Arbiter	4413	0.91%	6.39E-07	1.70E-05	1.20E-05	2.96E-05	2
IRAM Adapter	448	0.09%	1.00E-07	1.21E-05	6.94E-06	1.92E-05	2
SMPU	1619	0.33%	3.40E-07	1.41E-05	7.42E-07	1.52E-05	2
RV Timer	1240	0.26%	1.82E-07	6.22E-06	0.00E+00	6.40E-06	2
NCC Configuration	18	0.00%	3.88E-09	7.46E-07	0.00E+00	7.50E-07	2
Debugger	7	0.00%	6.50E-10	1.13E-09	6.99E-09	8.77E-09	2

Average power estimation, as its name suggests, calculates the average power for a specific timeframe. It needs a stimulus file to know what happens in a specific part of the circuit at what time, which is obtained from the simulation done in the execution time measurement process. By adjusting which parts of the stimulus file are read by Joules, the timeframe that will be analyzed can be adjusted so that it covers only the relevant parts of the simulation, i.e. the sending/receiving of data between the cores and the execution of the neural network operations. In other words, unnecessary parts, such as the initial loading of data into the memory, are omitted. An example of a report produced by the average power estimation can be seen in Table 5.3. This specific report covers four cores, with one core being expanded to include its parts (NCC 0). The cells and pct_cells columns show the size of a specific part. The leakage, internal, switching columns indicate the power consumption of that part. Leakage power, also known as static power, is the power loss due to leakage currents. The internal power is the power consumed by the charging and discharging of the internal and gate capacitances. Switching power, meanwhile, is the power lost due to the instantaneous short-circuit connection during the switching of transistors. The combined internal power and switching power make up what is known as dynamic power.

5.3.2. Time-based Power Estimation

Time-based power estimation produces a power versus time graph to allow a more detailed analysis of power consumption. It can be particularly useful if peak power consumption or how power consumption varies over time is important. An example graph is shown in Figure 5.6. Each graph in that figure represents the power consumption of a specific NCP core, and the names of the cores are indicated on the left-hand side. The actual execution of the program starts at around the 2400 us mark, with the time before that being the initialization and loading of data to the memory. The power consumption patterns of the three cores show the three cores working one after the other in a sequential fashion. This allows the peak power consumption during execution to be determined. Furthermore, it also allows one to calculate the difference in power consumption between the execution time and the idle time. Similarly to the report produced by the average power estimation, graphs representing the leakage, internal, and switching power can also be shown.

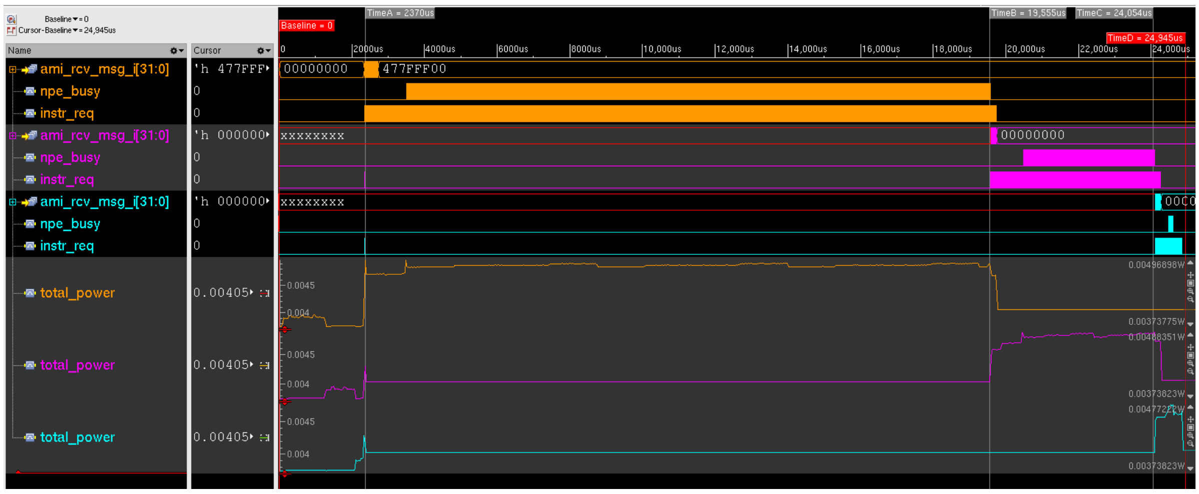


Figure 5.6: Example of a graph produced by time-based power estimation.

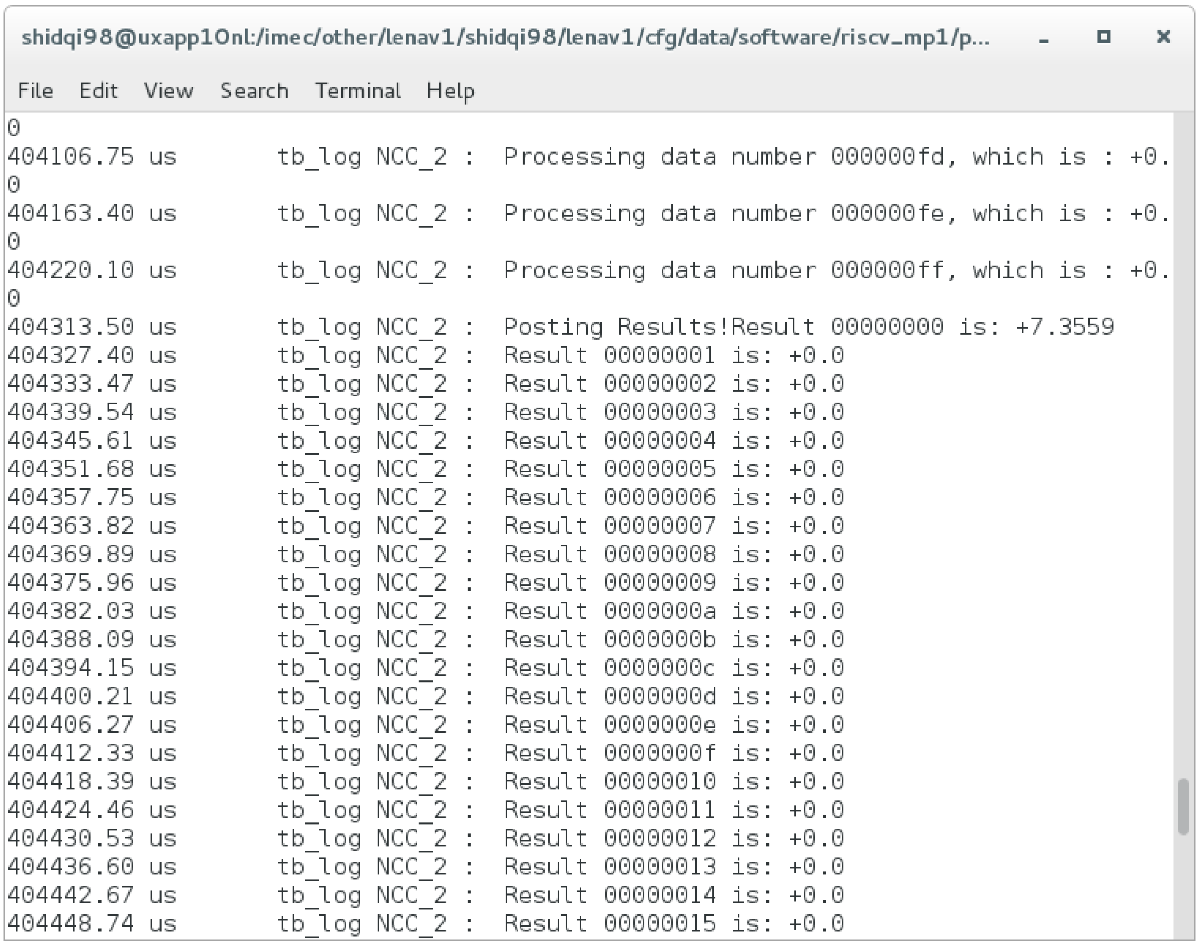
To obtain a deeper and more thorough analysis, both methods are used. The average mode allows us to obtain precise values, which are used in the end to compare SENeCA with other platforms discussed in [14]. The numbers can also be used to calculate how much power is consumed by a specific part as a percentage of the total power. However, the time-based mode is useful for obtaining peak power data and for determining how power is consumed over time. This method is also useful to determine the power consumption in the idle state. One might think that the power consumed in the idle state is simply the leakage power, but since switching activities still occur even in the idle state, this is most likely not the case.

5.4. Results for Version 1 : RISC-V Implementation

The methodology explained above will be applied to version 1 of the benchmarking program, which is the version in which the RISC-V core executes all neural network operations without the help of the NPEs or other components (as explained in Chapter 4). First, we will explore the execution time measurements, then we will explore the power measurement results.

5.4.1. Execution Time Measurements

As mentioned previously in this chapter, a simulation by Xcelium© is used to simulate SENeCA running the benchmark software. The print statements in the code would cause the text to be printed on the terminal used to run the simulation. At specific points in the program, the print statements are placed to mark a certain milestone being reached. The timestamps at which these print statements are executed will be used to calculate the execution time.


shidqi98@uxapp10nl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/p...
File Edit View Search Terminal Help

```
0
404106.75 us       tb_log NCC_2 :  Processing data number 000000fd, which is : +0.
0
404163.40 us       tb_log NCC_2 :  Processing data number 000000fe, which is : +0.
0
404220.10 us       tb_log NCC_2 :  Processing data number 000000ff, which is : +0.
0
404313.50 us       tb_log NCC_2 :  Posting Results!Result 00000000 is: +7.3559
404327.40 us       tb_log NCC_2 :  Result 00000001 is: +0.0
404333.47 us       tb_log NCC_2 :  Result 00000002 is: +0.0
404339.54 us       tb_log NCC_2 :  Result 00000003 is: +0.0
404345.61 us       tb_log NCC_2 :  Result 00000004 is: +0.0
404351.68 us       tb_log NCC_2 :  Result 00000005 is: +0.0
404357.75 us       tb_log NCC_2 :  Result 00000006 is: +0.0
404363.82 us       tb_log NCC_2 :  Result 00000007 is: +0.0
404369.89 us       tb_log NCC_2 :  Result 00000008 is: +0.0
404375.96 us       tb_log NCC_2 :  Result 00000009 is: +0.0
404382.03 us       tb_log NCC_2 :  Result 0000000a is: +0.0
404388.09 us       tb_log NCC_2 :  Result 0000000b is: +0.0
404394.15 us       tb_log NCC_2 :  Result 0000000c is: +0.0
404400.21 us       tb_log NCC_2 :  Result 0000000d is: +0.0
404406.27 us       tb_log NCC_2 :  Result 0000000e is: +0.0
404412.33 us       tb_log NCC_2 :  Result 0000000f is: +0.0
404418.39 us       tb_log NCC_2 :  Result 00000010 is: +0.0
404424.46 us       tb_log NCC_2 :  Result 00000011 is: +0.0
404430.53 us       tb_log NCC_2 :  Result 00000012 is: +0.0
404436.60 us       tb_log NCC_2 :  Result 00000013 is: +0.0
404442.67 us       tb_log NCC_2 :  Result 00000014 is: +0.0
404448.74 us       tb_log NCC_2 :  Result 00000015 is: +0.0
```

Figure 5.7: Output messages obtained from SENeCA while running the simulation with version 1.

Figure 5.7 shows the last part of the terminal that runs the simulation. Since the simulation takes a very long time for version 1, not all parts of the simulation can be shown here. However, by taking the timestamps from the terminal and subtracting the timestamps of adjacent milestones from each other, the execution times for the six parts of the benchmarking program mentioned in the methodology section above can be calculated. The results are shown in Table 5.4. The Timestamps column shows the seven milestones that are measured and their respective timestamps. The Execution Time column shows the execution times of the 6 parts mentioned previously, calculated from the timestamps measured. The total execution time for the Loop (neural network operations) and Sending parts are also calculated, placed under each respective column. Finally, the grand total is calculated, which represents the total time of inference for one frame.

Table 5.4: Execution time measurements for version 1. All times are in microseconds (μs).

Sequential					
Timestamps		Execution Time			
1467	Start	Loop 1	18620	Sending 1	38524
39991	Input stream 1 finished	Loop 2	28557	Sending 2	1621
58611	Loop 1 finished	Loop 3	2617	Sending 3	1419
60232	Input stream 2 finished	Total	49794	Total	41564
88789	Loop 2 finished	Grand total			91358
90208	Input stream 3 finished				
92825	Loop 3 finished				

From the table, some things are clear. First, the grand total, which represents the time it takes for one inference, is around 0.37s, which is far too great to process the frames in real time (which requires at least one inference per 10ms). Furthermore, since layer 1 receives the input vector with no zero values, it takes the majority of the execution time, around 66%. However, keep in mind that this version uses floating point (fp32) variables, while the next versions all use brainfloat (bf16).

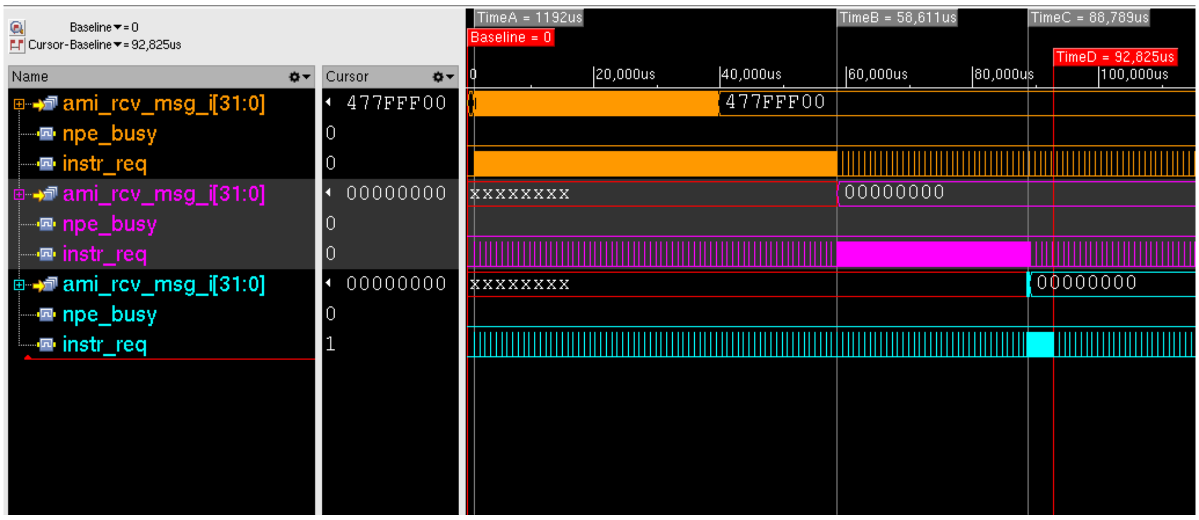


Figure 5.8: Waveforms of the signals produced by SENeCA when simulated using version 1 of the benchmark.

Figure 5.8 shows the simulated waveforms of SENeCA while running version 1 of the benchmark. The column on the left shows the signal names, divided into 3 colors. These colors represent the cores to which the signals belong (orange for core 1, turquoise for core 2, and purple for core 3). The signals here are chosen to represent a specific part of SENeCA that is active. The signal instr_req comes from the RISC-V core and is switching if it requests instructions, signalling that it is active. Ami_rcv_msg signals that the axon messaging interface is active, showing that the core is receiving messages from the previous core. Npe_busy signals that the NPEs are being utilized. Note that all of the npe_busy signals stay 0, since none of the NPEs are used.

5.4.2. Power Measurements

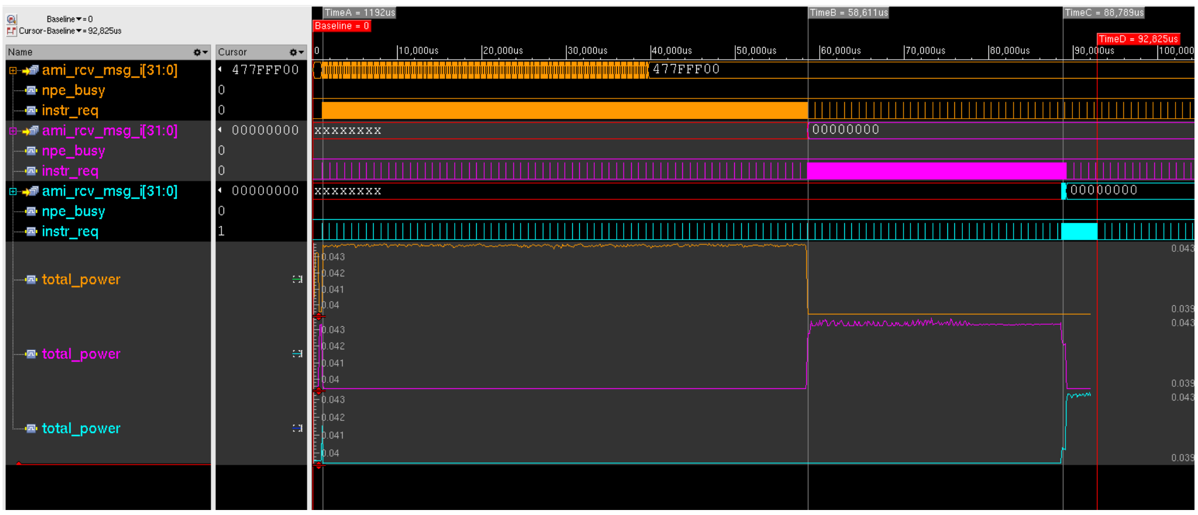


Figure 5.9: Time-based power graph of version 2 together with several signals.

After performing the time-based power estimation, the graph showing the power consumption of the cores (the three lowest signals) was added to the previous simulation waveform, forming the graph shown in Figure 5.9. The green graph shows the power consumption of core 1, the turquoise graph shows that of core 2, and the dark blue graph shows the power consumption of core 3. Also, this graph has several cursors, placed at the moments when one core starts/stops and the next one takes over. The cursor labeled TimeA, for example, is placed at the time when core 1 stops execution after calculating the last neuron value, and core 2 begins to receive the datastream. While the power graphs provide some insight into how the power consumption of the cores reflect their activity, such as the power graph of core 1 peaking when it is active and drops around the time core 2 begins execution (marked by the cursor labeled TimeA), it is difficult to obtain precise numbers to calculate the total energy. As previously explained, this is where the average power estimation comes into play.

The average power estimation can be done over the entire time period that SENeCA is active. However, to obtain a more detailed picture of the power consumption of the cores, we decided to divide the period into three parts, each corresponding to a core. The time between cursors ”Baseline” and ”TimeA”, is the time during which core 1 is active, henceforth referred to as Active 1. Between cursors ”TimeA” and ”TimeB”, core 2 is active, and that time will be referred to as Active 2. The active time of core 3, meanwhile, is the time between cursors ”TimeB” and ”TimeC”. It will be referred to as Active 3. Average mode power analysis will be done on these separate parts.

Table 5.5: Average power consumption of version 1, during the active time of core 1 (Active 1). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_0	25.00%	2.80E-03	3.74E-02	3.44E-03	4.37E-02
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 5.6: Average power consumption of version 1, during the active time of core 2 (Active 2). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_1	25.00%	2.80E-03	3.72E-02	3.23E-03	4.33E-02
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 5.7: Average power consumption of version 1, during the active time of core 3 (Active 3). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_2	25.00%	2.80E-03	3.73E-02	3.23E-03	4.33E-02
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Tables 5.5, 5.6, 5.7 show the results of the average power estimation. Estimating the average power allows us to predict the power consumption of all components of SENeCA during a specific timeframe of the simulation. To know the power consumption values for each core when they are active and idle, the simulation is divided into 3 timeframes. One timeframe represents the time at which one specific core is active.

To obtain a more definitive number that represents the performance of SENeCA, the power consumption values and execution times will be used to calculate the energy consumed per inference. The total energy is obtained by the following equation:

$$W_{total} = W_{t1} + W_{t2} + W_{t3} = P_{t1}T_{t1} + P_{t2}T_{t2} + P_{t2}T_{t2} \tag{5.1}$$

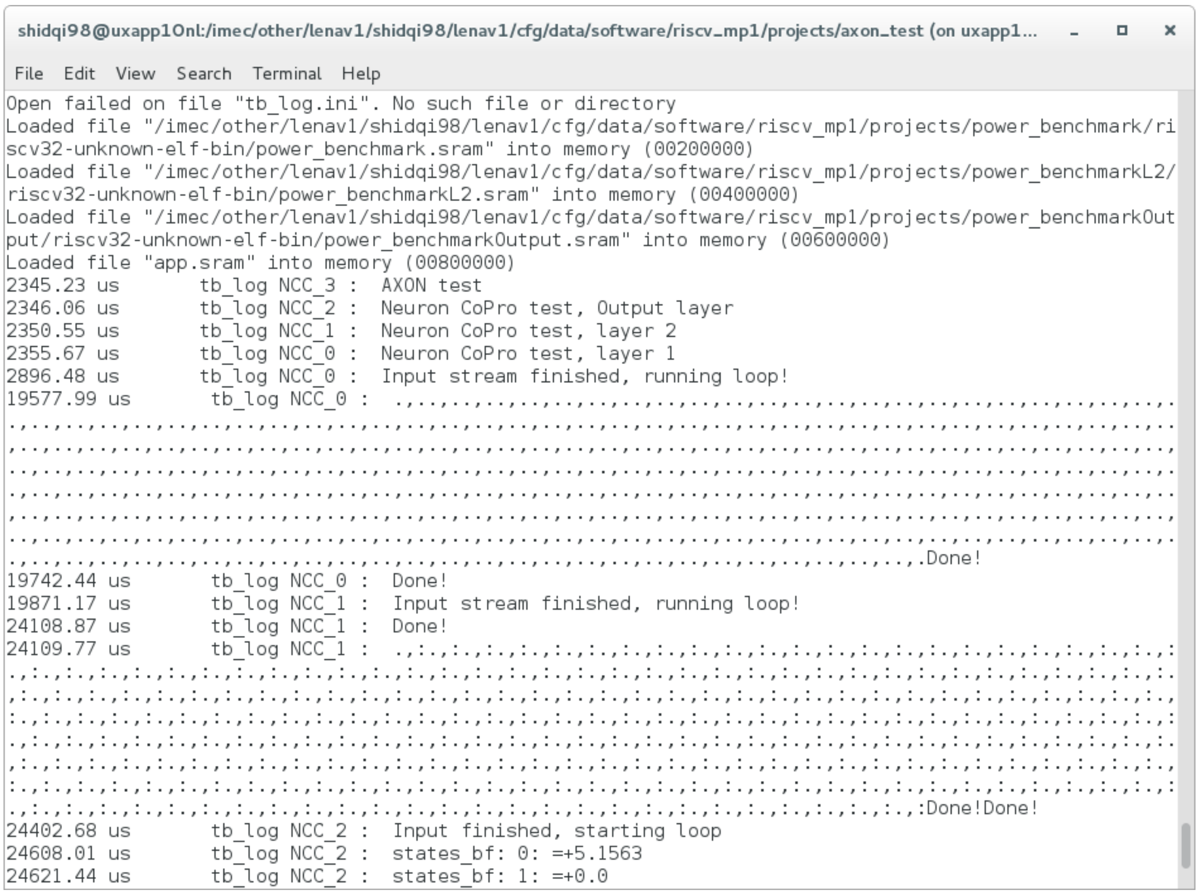
P_{tX} refers to the average power consumption during the time period of Active X, where X is 1, 2, or 3. These numbers are obtained by summing up the ”Total” column in Tables 5.9, 5.10, and 5.11. The same applies to T_{tX}, except that it refers to the duration of that period of time. The duration is obtained by adding the times shown in Table 5.8, for example, the total time of Active 1 is Loop 1 plus Sending 1. By using these numbers to calculate the total energy, we get the number of 22.11 mJ.

5.5. Results for Version 2 : Baseline NPE Implementation

Version 2, as mentioned previously, is the version that uses the NPEs in the most basic way. This will be used as a baseline and the results obtained from running this version will be analyzed to determine the methods in which optimization is possible, with regard to both timing and power.

5.5.1. Execution Time Measurements

As mentioned above, print statements placed in the source code allow us to observe when a certain part of the program has been completed. In this version, since the simulation takes less time compared to version1, all the printed statements of the simulation can be shown. Figure 5.10 shows the screenshot of the simulation.


shidqi98@uxapp1Onl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/axon_test (on uxapp1...
File Edit View Search Terminal Help

```
Open failed on file "tb_log.ini". No such file or directory
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_benchmark/ri
scv32-unknown-elf-bin/power_benchmark.sram" into memory (00200000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_benchmarkL2/
riscv32-unknown-elf-bin/power_benchmarkL2.sram" into memory (00400000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_benchmarkOut
put/riscv32-unknown-elf-bin/power_benchmarkOutput.sram" into memory (00600000)
Loaded file "app.sram" into memory (00800000)
2345.23 us       tb_log NCC_3 :  AXON test
2346.06 us       tb_log NCC_2 :  Neuron CoPro test, Output layer
2350.55 us       tb_log NCC_1 :  Neuron CoPro test, layer 2
2355.67 us       tb_log NCC_0 :  Neuron CoPro test, layer 1
2896.48 us       tb_log NCC_0 :  Input stream finished, running loop!
19577.99 us       tb_log NCC_0 :  .,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.Done!
19742.44 us       tb_log NCC_0 :  Done!
19871.17 us       tb_log NCC_1 :  Input stream finished, running loop!
24108.87 us       tb_log NCC_1 :  Done!
24109.77 us       tb_log NCC_1 :  .,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:
.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.
,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,
:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:
.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.
,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,
:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:
.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:.,:Done!Done!
24402.68 us       tb_log NCC_2 :  Input finished, starting loop
24608.01 us       tb_log NCC_2 :  states_bf: 0: =+5.1563
24621.44 us       tb_log NCC_2 :  states_bf: 1: =+0.0
```

Figure 5.10: Output messages obtained from SENeCA while running the simulation with version 2.

By extracting the timestamps that correspond to a certain milestone being reached, the execution time of each of the parts, as well as the total execution time, can be calculated, similar to what we did for version 1. Table 5.8 shows the result of these calculations.

Table 5.8: Execution time measurements for version 2. All times are in microseconds (μs).

Version 2					
Timestamps		Execution Time			
2355	Start	Loop 1	16681	Sending 1	541
2896	Input stream 1 finished	Loop 2	4238	Sending 2	294
19577	Loop 1 finished	Loop 3	206	Sending 3	293
19871	Input stream 2 finished	Total	21125	Total	1128
24109	Loop 2 finished	Grand total			22253
24402	Input stream 3 finished				
24608	Loop 3 finished				

Observing Table 5.8, a few things are clear. First, since the total time required to process one is 22253 μs, or around 22 ms, this version is still not fast enough to process frames in real time. Since the frames represent 10 ms of audio, the processing time must also be less than 10 ms. Furthermore, by looking at the execution times of the individual parts, it is also clear that the time required to perform the neural network calculations is much larger than the sending time. Finally, Loop 1, the execution time of neural network operations that take place in layer 1, takes up the vast majority of the total time. Therefore, it is quite clear that optimizations are required, and a prime target is the source code that executes the neural network operations, with the most significant improvement expected in Loop 1.

However, this version is still a significant improvement from version 1 that did not use the NPEs at all. To be precise, the speedup is around 16 times. Theoretically, since the NPEs process 8 data values at once, the speedup should be limited to 8 times, but since version 1 uses the fp32 number format while version 2 uses bf16 numbers to comply with the NPE requirements, further speedup was made possible at the cost of precision. Other factors that could explain this extra speedup is that the RISC-V processor does not natively support the fp32 number format, so an external library had to be used.

5.5.2. Power Measurements

To measure the power consumption of Version 2, a time-based power analysis was performed. The resulting graph is shown in Figure 5.11. On the left-hand side are the names of the signals, color-coded by their cores. Orange stands for core 0, purple for core 1, and turquoise for core 2. These color codes will be used throughout this document. The three signals are active during a specific part of the program. The signal named ami_rcv_msg_i is active during the receiving period by the corresponding core, npe_busy is active during the neural network operations, while instr_req is the signal emitted by the RISC-V processor of that core, indicating that it is active when the signal fluctuates. The four cursors are placed at the starts and ends of each core's active time.

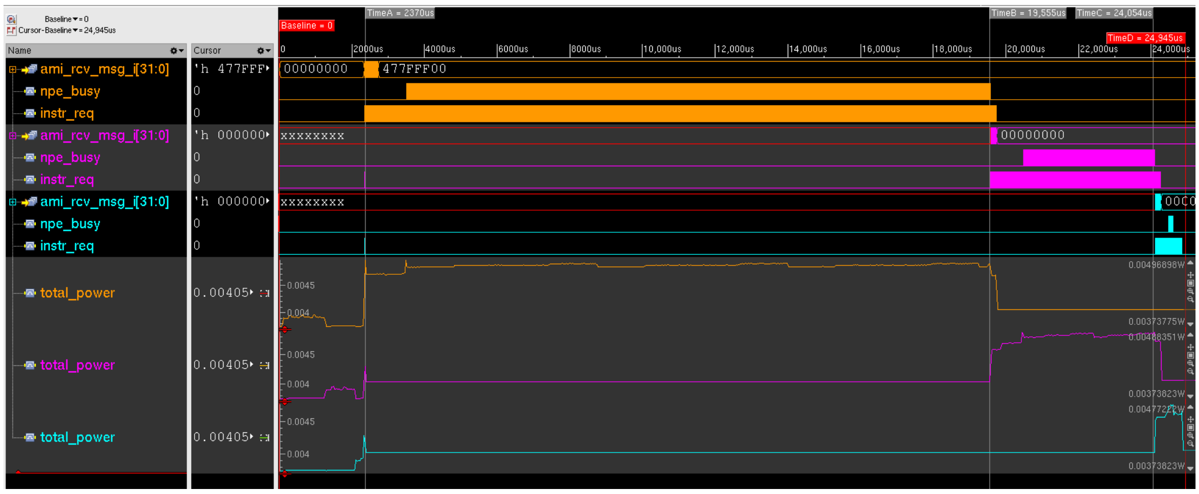
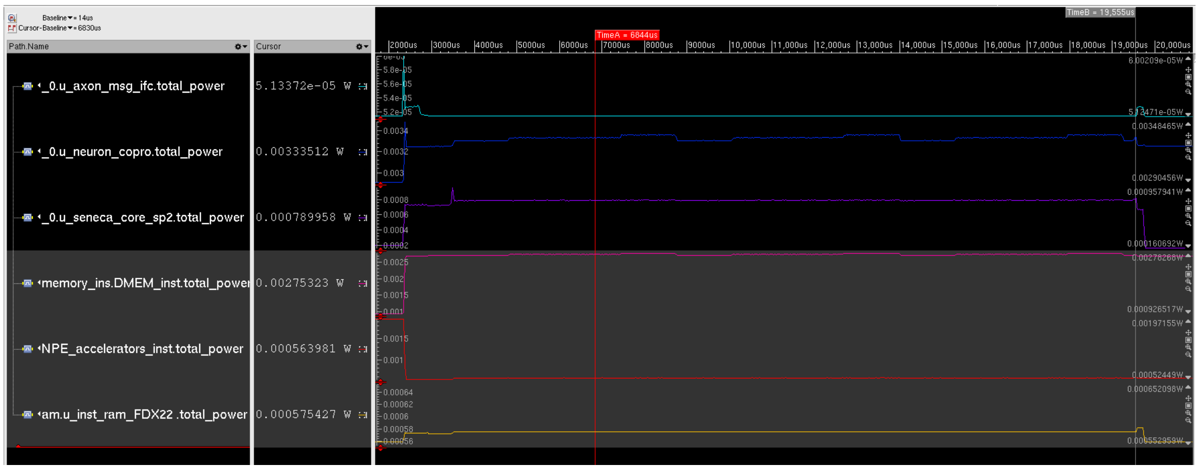


Figure 5.11: Time-based power graph of version 2 together with several signals.

Meanwhile, the three bottom graphs are the power estimation generated by Joules©. The labels on the left-hand side are color-coded to indicate the core. These power consumption graphs indicate the total power consumed by the three cores that are being used. As expected, the cores consume the most power during the times they are active, which corresponds to the sections marked by the cursors in Figure 5.11.

The three measured cores are identical, and they also run similar programs, differing only in the size of the neural network layer. This also makes the power consumption more or less identical. During active time, the power consumption is about 4.8 mW, while during idle time it decreases to around 4 mW. To determine which components of SENeCA are the most power hungry, a more detailed observation can be made by looking at Figure 5.12.


Baseline = 14us
Cursor-Baseline = 6830us

Path.Name	Cursor
_0.u_axon_msg_ifc.total_power	5.13372e-05 W
_0.u_neuron_copro.total_power	0.00333512 W
_0.u_seneca_core_sp2.total_power	0.000789958 W
memory_ins.DMEM_inst.total_power	0.00275323 W
NPE_accelerators_inst.total_power	0.000563981 W
am.u_inst_ram_FDX22 .total_power	0.000575427 W

Figure 5.12: Time-based power graph of version 2, broken down into components.

Figure 5.12 shows a more detailed version of the time-based power graph of core 0, with the different parts also being shown. Some parts are omitted, and the ones shown here are the ones that consume the most power. The cursor labeled ”TimeB” is placed in the middle of the execution time of Layer 1. On the left-hand side, the column labeled ”Cursor” shows the instantaneous power consumed at the time of TimeB. Although power consumption varies slightly over time, the numbers here serve as a useful rough estimate. The uppermost graph is the total power consumed by core 0, the two bottommost graphs are components of u_neuron_copro (the graph right above them), while the others are components of core 0. From this graph we can infer that the most power-hungry components are the instruction memory (2nd from the top) and u_neuron_copro. The latter component is then divided, since it contains both the NPE and the data memory. By breaking down u_neuron_copro, we can see that the NPEs do not consume much power, and the lion’s share of the power is consumed by the data memory. These results are in line with the explanation in Chapter 1, that the memory is the most power-hungry component in computers in general. Therefore, to optimize the software with regard to power consumption, the number of commands that access the memory must be minimized.

In order to obtain more precise numbers, the average mode power estimation of Joules© was performed on the three time periods, Active 1, Active 2, and Active 3. In this way, we will obtain more detailed power consumption data, including dynamic and static power. Also, by performing it on the 3 time periods, the average power consumed by a core when it is active and idle can be known. The results of this estimation are presented in Tables 5.9, 5.10, and 5.11. From the tables it can be inferred that the average power during active time is 4.5 to 4.8 mW, while during idle time it decreases to around 4.0 mW, similar to our conclusions of the time-based power estimation.

Table 5.9: Average power consumption of version 2, during the active time of core 1 (Active 1). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_0	25.00%	2.80E-03	5.30E-03	4.86E-04	8.59E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 5.10: Average power consumption of version 2, during the active time of core 2 (Active 2). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_1	25.00%	2.80E-03	5.30E-03	4.86E-04	8.59E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 5.11: Average power consumption of version 2, during the active time of core 3 (Active 3). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_2	25.00%	2.80E-03	5.30E-03	4.86E-04	8.59E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Similarly to what we did in the previous section, we would like to know the energy consumption of SENeCA running version 2 of the benchmark. By replacing all the variables in Equation 5.1 with the numbers obtained from the tables, we get the total energy consumption of 0.548 mJ per inference.

5.6. Power Estimation of Basic NPE Operations

In this section, a different type of test will be performed. Instead of running a full application and measuring the performance for the entire program, this test will have the NPEs perform basic operations such as arithmetic operations, bitwise operations, and memory operations, and measure the energy consumption of each operation. The data obtained from this experiment can then be used to estimate the energy consumption of applications that will be run on SENeCA in the future even before implementation. Furthermore, future researchers can compare the performance and energy consumption of their design with those of SENeCA using the data obtained from this experiment.

This experiment is performed on the same setup as explained above, but with the clock speed of SENeCA set to 500 MHz. The NPEs are made to perform loops of different instructions one after the other with slight delays between them to allow us to discern the change in power consumption when executing an operation. Possible operations for NPEs are listed in Table 3.1. However, the power consumption of a specific instruction can vary depending on the operands. For example, multiplying a number by zero and a non-zero value can lead to differences in power consumption. Therefore, for this experiment, a different list of operations will be used. They are listed in Table 5.12.

Table 5.12: List of NPE operations whose energy consumption is to be measured.

No	Operation	OpCode	Explanation
0	Baseline	-	Baseline power consumption (static power)
1	AMI Event	AMI	Interrupt indicating a message received by the AMI
2	Multiplication	MUL	Multiplication with changing operands
3	Multiplication (A*0)	MUL	Multiplication with one operand being zero
4	Addition	ADD	Addition with changing operands
5	Division	DIV	Division with changing operands
6	No Operation	NOP	No operation (Pipeline stall)
7	Greater than	GTH	Greater than operation with changing operands
8	Equal	EQL	Equal operation with changing operands
9	Maximum	MAX	Maximum operation with changing operands
10	Absolute value	ABS	Absolute value operation with changing operands
11	Bitwise AND	AND	Bitwise AND operation with changing operands
12	Bitwise OR	ORR	Bitwise OR operation with changing operands
13	Logical shift left	SHL	Left shift operation with changing operands
14	Memory store	MST	Memory store operation with changing operands
15	Memory load	MLD	Memory load operation with changing operands
16	Event generation (NT)	EVC	Event generation operation with the EVC not triggered
17	Event generation	EVC	Event generation operation with the EVC triggered

As mentioned previously, all of the operations in the NPE require one clock cycle to be performed. With the clock speed being set to 500 MHz, one instruction takes 2 ns to complete. For each instruction loop, the instruction is repeated 64 times, meaning that each loop takes about 128 ns. Between these instruction loops, the RISC-V core takes over and spends some time to configure the instruction kernel of the NPEs, during which the components other than the RISC-V become idle. This idle-active-idle cycle gives us a good opportunity to estimate the energy consumption. Furthermore, in contrast to the results presented for versions 1 and 2 above, in this experiment the power consumption of individual components of SENeCA will be analyzed, instead of individual cores (since this experiment only uses one core). A screenshot of the power consumption of different SENeCA components running the instructions listed in Table 5.12 is shown in Figure 5.13.

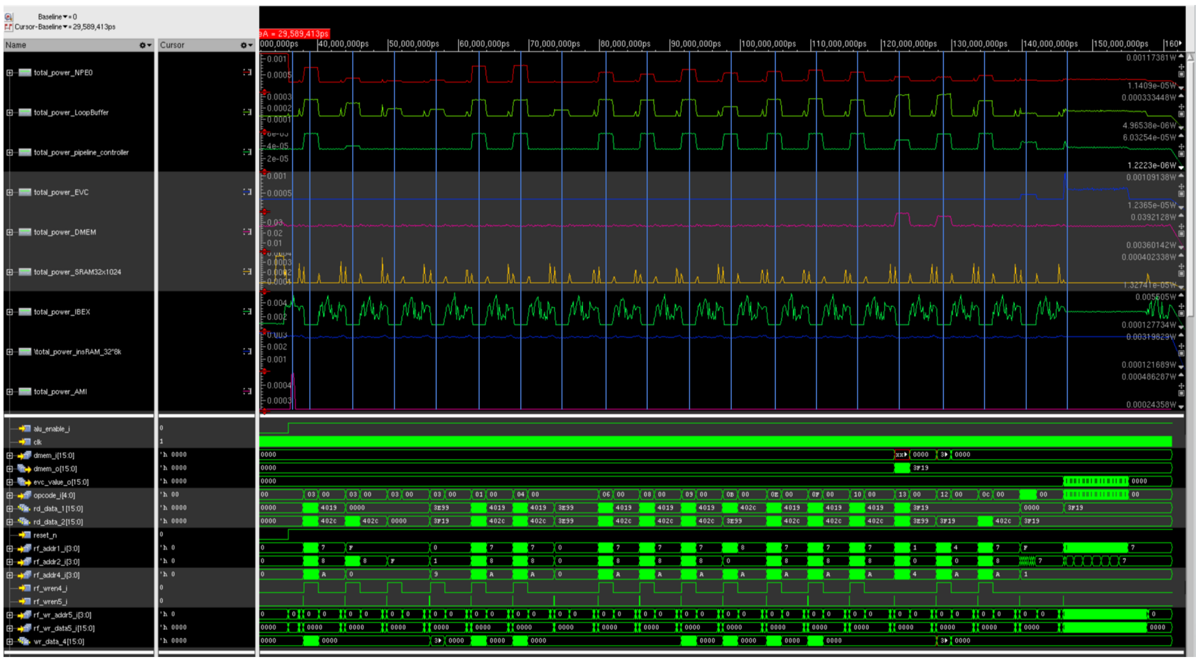


Figure 5.13: Time based power consumption graph of individual NPE operations.

In Figure 5.13 above, the screen is divided vertically into 2 parts. The upper part contains the power consumption graphs for SENeCA's components, which can be identified by the labels on the leftmost column. The lower part contains the signals that SENeCA produces during the simulation, which are used to identify which instruction is being performed at a specific time. In the upper part, each vertical blue line indicates one instruction loop performed by the NPEs (except the leftmost one, which is one interrupt triggered by the AMI). There are 20 blue lines, each corresponding to an instruction listed in Table 5.12. The order of the instruction loops performed is the same as that of the instructions listed in the table.

To calculate the energy consumption of an NPE instruction, the power consumption must be multiplied by the time it takes to execute the instruction. The waveforms shown in Figure 5.13 allow us to know the instantaneous power consumption of a single component by simply placing the cursor on the time at which the instruction is being performed. Ideally, integration can be done over the time of 2 ns (the time it takes to complete one instruction), but since that is difficult to do manually, and approximation is done instead. This approximation simply calculates the maximum power consumption in 1 loop by 2 ns, obtaining the energy. By doing this for all loops and some of the components shown in Figure 5.13, we obtain the energy values. These are presented in Table 5.13. Regarding the abbreviations for the component names, PC stands for pipeline controller, EVC stands for event capture unit, DMEM stands for data memory, and AMI stands for the Axon Messaging Interface. The table covers most of the operations listed in Table 5.12. A dash (-) indicates that the energy consumption of the component is the same as that of the baseline, that is, the component is switched off. For comparison, the energy required for Loihi to perform one synaptic operation is 80 pJ [72].

Table 5.13: Energy consumption of NPE operations (calculated from the total power values). All values are expressed in femtojoules (fJ).

No	Operation	Code	NPE	LB	PC	EVC	DMEM	AMI	RISC	Total
0	Baseline	N/A	1.2	0.8	0.04	2.6	16444	18.48	194	16661
1	AMI Event	AMI	-	-	-	-	-	1114	-	17757
2	Multiplication	MUL	1158	554	120	-	-	-	-	18491
3	Multiplication (A*0)	MUL	474	486	82	-	-	-	-	17701
4	Addition	ADD	1252	550	120	-	-	-	-	18581
5	Division	DIV	1304	548	122	-	-	-	-	18633
6	No Operation	NOP	60	366	70	-	-	-	-	17155
7	Greater than	GTH	784	548	122	-	-	-	-	18113
8	Equal	EQL	684	538	120	-	-	-	-	18001
9	Maximum	MAX	958	546	120	-	-	-	-	18283
10	Bitwise AND	AND	802	552	120	-	-	-	-	18133
11	Bitwise OR	ORR	884	560	120	-	-	-	-	18223
12	Logical shift left	SHL	772	540	120	-	-	-	-	18091
13	Memory store	MST	336	612	98	-	41504	-	-	42765
14	Memory load	MLD	346	638	116	-	35744	-	-	37059
15	Event generation (NT)	EVC	76	306	92	916	-	-	-	18046
16	Event generation	EVC	144	350	76	1200	-	-	-	18426
17	RISC-V core process	N/A	-	-	-	-	-	-	7020	23487

The baseline (No. 0) is the energy consumed by the components every clock cycle when they are not operating. Therefore, the numbers represent the static power consumption of the components. Regarding the NPE, LB, PC, and EVC, since they do not store any data, turning them off when not used saves a significant amount of energy. The data memory, meanwhile, cannot be turned off, since the data stored will be lost. The data memory is by far the largest consumer of energy.

The arithmetic operations (operations 2 to 5) are implemented with zero skipping, so multiplications with zero (No. 3), for example, consume less energy than normal multiplications (No. 2). Since the number format used is BF16, addition consumes more energy than multiplications, since additions require the mantissas (7 bits) of the numbers to be aligned using their exponents before they can be added.

Afterwards, normalization of the addition result must also be done. Alignment and normalization require multiple conditional checks to occur in one clock cycle. Multiplications, on the other hand, only require multiplications of the mantissas, whereas the exponents can simply be added. Division is implemented by taking an inverse of the second operand and then multiplying it by the first operand. Therefore, the extra energy for the divisions comes from the inverse operations, with the rest of the process being identical. A NOP operation (No. 6), where nothing is produced, is different from the baseline, since the NPEs still need to store the data in the registers and the LB and PC still need to control the NPE process. Therefore, it still consumes energy.

Operations 7,8,9 all involve comparisons of the two operands. The first operation performed is to check whether the two operands are equal. In the case of operation 8, the result of that check is simply set as the final result. For the other 2 operations, further checks to determine which of the operands is greater are done. This extra check explains why the energy consumption values of operations 7 and 9 are greater than those of operation 8 for the NPE. The bitwise operations (11, 12, and 13) all involve basic logic gates, and their energy consumption values are quite similar. The memory operations (14 and 15) see a spike in the energy consumption of the memory, as expected. Note that the energy consumption of the memory depends on the number and size of the memory blocks. Here, 256 blocks of 32 KBit each are used.

Finally, the event generation operations (16 and 17) involve the EVC and NPE, so their energy consumption values increase. The NPE only has to check the value in one of its registers, so the increase is not significant. However, the EVC does most of the work, so its power consumption spikes. Finally, operation No. 17 shows the energy consumed per clock cycle when the RISC-V core is processing while the other parts (the NCP) are idle.

6 Optimizations and Final Results

As explained in Chapter 5, the baseline version is not fast enough to process the data in real-time. The energy consumption could also be reduced. Therefore, optimizations with respect to both execution time and power consumption are necessary. First, analysis done on the bottlenecks and power-hungry components of one version will be done, allowing the the identification of a potential target to be improved. Then, the optimization method is considered, using principles of both neuromorphic computing and computer architecture in general, discussed in Chapter 1. The method is then implemented, creating a new version that would ideally be more efficient than the previous version. This new version will then again be analyzed, creating a loop that aims to produce the most power-efficient version at the end.

6.1. Results of Version 3 : Loop Unrolling With 2 Elements

As mentioned in the discussion about results of version 2 presented in Chapter 5, the neural network operations of layer 1 are the most time-consuming, while the data memory is the most power-hungry. In this version, we will attempt to improve upon that version by using the principle known as memory locality.

6.1.1. Optimization method and Implementation

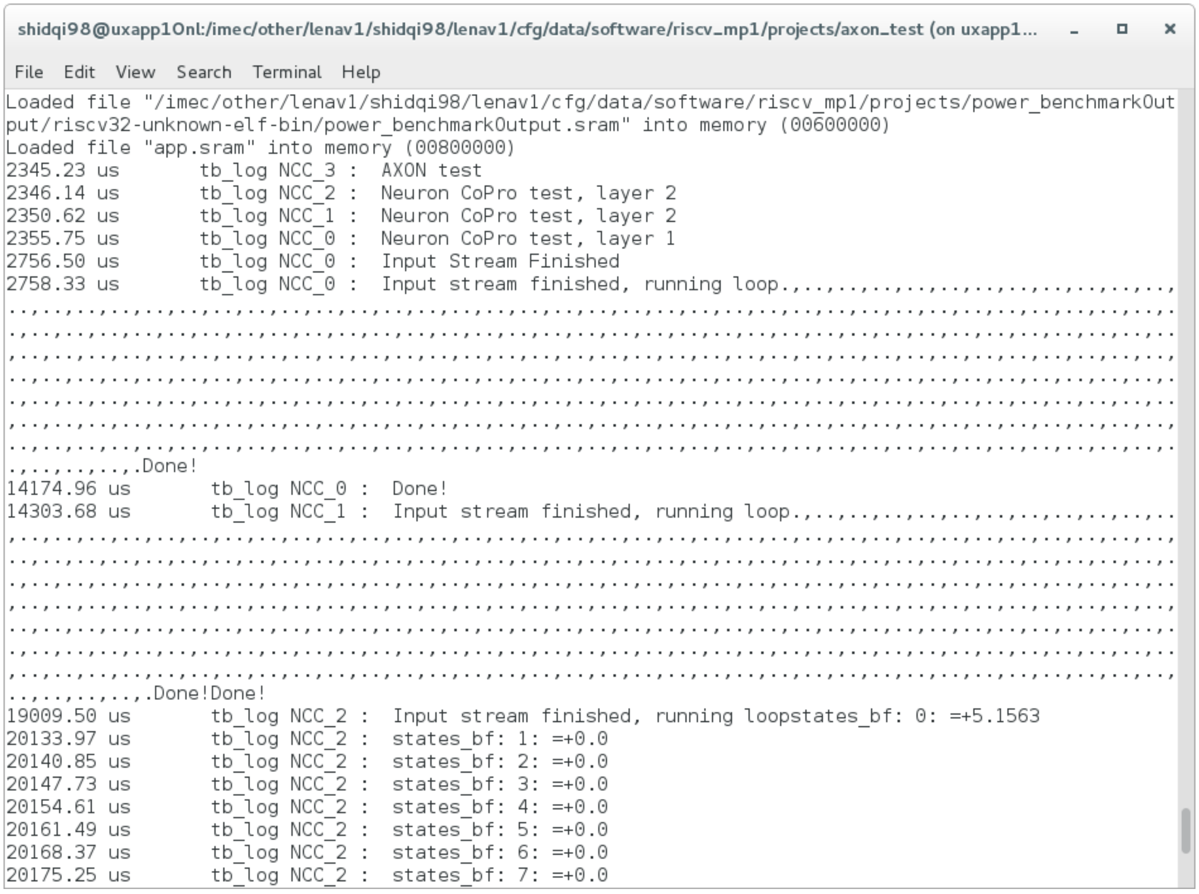
In conventional CPUs, cache memory is used as a way to reduce energy consumption and processing time by using it to save data that will be reused, minimizing the number of accesses to the main memory[80]. This spatial locality principle is widely used in various architectures, and it can also be used in this case. In Chapter 3, we mentioned that the NPEs have a number of registers to store variables. In version 2, these registers were used to store 1 input value, 1 weight value, and 1 neuron value per iteration, to be used as operands in the MAC operation. Since there are 16 available registers in total, one possible method of improvement is to use these registers as a cache, since access to the registers does not consume as much energy and time as access to the memory. The registers can store additional values, for instance 1 extra input value and 1 extra weight value, allowing 2 MAC operations in a single iteration. While this does increase the execution time of each iteration, it decreases the number of iterations and reduces the number of read/write commands to the memory. Listing 6.1 shows the implementation of this improvement in the source code.

Listing 6.1: Source code of NPE Loop of Version 3

```

void KERNEL_SENECA(float datapoint, float datapoint2)
{
    last_miram_1stHalf_addr_used = 0x0;
    last_miram_2ndHalf_addr_used = (MIRAM_1ST_HALF_LAST_INDEX);
    uint32_t eval_reg = 0u;
    uint32_t eval_reg2 = eval_reg + 1;

    //Convert first datapoint
    float ev_val_F = datapoint;
    uint16_t ev_val_BF = 0;
    FLOAT2BF(&ev_val_F, &ev_val_BF);

    //Convert second datapoint
    float ev_val_F2 = datapoint2;
    uint16_t ev_val_BF2 = 0;
    FLOAT2BF(&ev_val_F2, &ev_val_BF2);

    //Write datapoints to registers
    write_to_regfile(eval_reg, ev_val_BF);
    write_to_regfile(eval_reg2, ev_val_BF2);

    //Setup of weight register 1
    uint32_t weight_reg = eval_reg2 + 1;
    uint32_t weight_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    //Setup of weight register 2
    uint32_t weight_reg2 = weight_reg + 1;
    uint32_t weight2_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    //Setup of state register
    uint32_t state_reg = weight_reg2 + 1;
    uint32_t state_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    //1st MAC operation
    uint32_t mul_res_reg = state_reg + 1;
    append_to_miram_command(MUL, eval_reg, weight_reg, mul_res_reg);
    uint32_t new_state_reg = mul_res_reg + 1;
    append_to_miram_command(ADD, mul_res_reg, state_reg, new_state_reg);

    //2nd MAC operation
    append_to_miram_command(MUL, eval_reg2, weight_reg2, mul_res_reg);
    append_to_miram_command(ADD, mul_res_reg, new_state_reg, state_reg);
    append_to_miram_write_reg_to_mem(state_adr_addr_ptr, state_reg, 0x01u); // DO_INCREAMENT
        by 0x01u
}

void DO_LOOP(void * beginning_of_weightsV, void* beginning_of_statesV, void *
    beginning_of_weightsV2)
{
    last_miram_1stHalf_addr_used = 0x0;
    last_miram_2ndHalf_addr_used = (MIRAM_1ST_HALF_LAST_INDEX);
    uint32_t eval_reg = 0u;
    uint32_t eval_reg_2 = eval_reg + 1;

    uint32_t weight_reg = eval_reg_2 + 1;
    uint32_t weight_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr, beginning_of_weightsV,
        weight_reg, 0x01u);

    uint32_t weight_reg2 = weight_reg + 1;
    uint32_t weight_adr_addr_ptr2 = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr2, beginning_of_weightsV2,
        weight_reg2, 0x01u);

    uint32_t state_reg = weight_reg2 + 1;
    uint32_t state_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(state_adr_addr_ptr, beginning_of_statesV, state_reg,
        0x00u);

    last_miram_2ndHalf_addr_used += 5u;
    uint32_t loopcfg = LOOP_CONFIG_INST(1u, last_miram_2ndHalf_addr_used,
        MIRAM_1ST_HALF_LAST_INDEX+0x01u);
    WriteLoopConfig(loopcfg);

    WFI();
}
```

Compared to version 2, each iteration of version 3 is longer, but because each iteration performs twice the number of operations, only a half the number of iterations are required. Furthermore, 2 iterations of version 2 have in total 2 read operations and 2 write operations for the neuron state, while this version's iteration requires only 1 read operation and 1 write operations. Essentially, this technique is similar to the optimization method known as loop unrolling[44]. However, one downside of this method is that zero-skipping can only happen if both input values to be processed in the same iteration are zero. This reduces the likelihood of zero-skipping, since a zero input value adjacent to a nonzero input value cannot be skipped.

6.1.2. Execution Time Measurements


shidqi98@uxapp10nl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/axon_test (on uxapp1...
File Edit View Search Terminal Help

```
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_benchmarkOut
put/riscv32-unknown-elf-bin/power_benchmarkOutput.sram" into memory (00600000)
Loaded file "app.sram" into memory (00800000)
2345.23 us       tb_log NCC_3 :  AXON test
2346.14 us       tb_log NCC_2 :  Neuron CoPro test, layer 2
2350.62 us       tb_log NCC_1 :  Neuron CoPro test, layer 2
2355.75 us       tb_log NCC_0 :  Neuron CoPro test, layer 1
2756.50 us       tb_log NCC_0 :  Input Stream Finished
2758.33 us       tb_log NCC_0 :  Input stream finished, running loop.,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,.Done!
14174.96 us       tb_log NCC_0 :  Done!
14303.68 us       tb_log NCC_1 :  Input stream finished, running loop.,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,.Done!Done!
19009.50 us       tb_log NCC_2 :  Input stream finished, running loopstates_bf: 0: =+5.1563
20133.97 us       tb_log NCC_2 :  states_bf: 1: =+0.0
20140.85 us       tb_log NCC_2 :  states_bf: 2: =+0.0
20147.73 us       tb_log NCC_2 :  states_bf: 3: =+0.0
20154.61 us       tb_log NCC_2 :  states_bf: 4: =+0.0
20161.49 us       tb_log NCC_2 :  states_bf: 5: =+0.0
20168.37 us       tb_log NCC_2 :  states_bf: 6: =+0.0
20175.25 us       tb_log NCC_2 :  states_bf: 7: =+0.0
```

Figure 6.1: Output messages obtained from SENeCA while running the simulation with version 3.

Figure 6.1 shows the simulation of version 3 after a successful run. The execution times extracted from the simulation is shown in Table 6.1. The grand total is 17.7 ms, still too slow for real-time processing. Compared to the results of version 2 shown in Table 5.8, however, the total runtime of the program decreased by around 5000 μs, a major improvement. As expected, The majority of the improvement comes from the reduction of the execution time of layer 1, accounting for more than 100% of the improvement. This is offset, however, by the other layers. The execution time of layers 2 and 3 actually increased. This due to the zero-skipping being performed less frequently in version 3, as explained above. Layer 1 is not affected by it, since none of the elements of the input vector is zero. Layer 3, meanwhile, receives many zero inputs, some that cannot be skipped. This inefficiency will be fixed in version 5.

Table 6.1: Execution time measurements for version 3. All times are in microseconds (μs).

Version 3						
Timestamps		Execution Time				
2355	Start	Loop 1	11318	Sending 1	401
2756	Input stream 1 finished	Loop 2	4411	Sending 2	229
14074	Loop 1 finished	Loop 3	1230	Sending 3	189
14303	Input stream 2 finished	Total	16959	Total	819
18714	Loop 2 finished	Grand total			17778
18903	Input stream 3 finished				
20133	Loop 3 finished				

6.1.3. Power Measurements

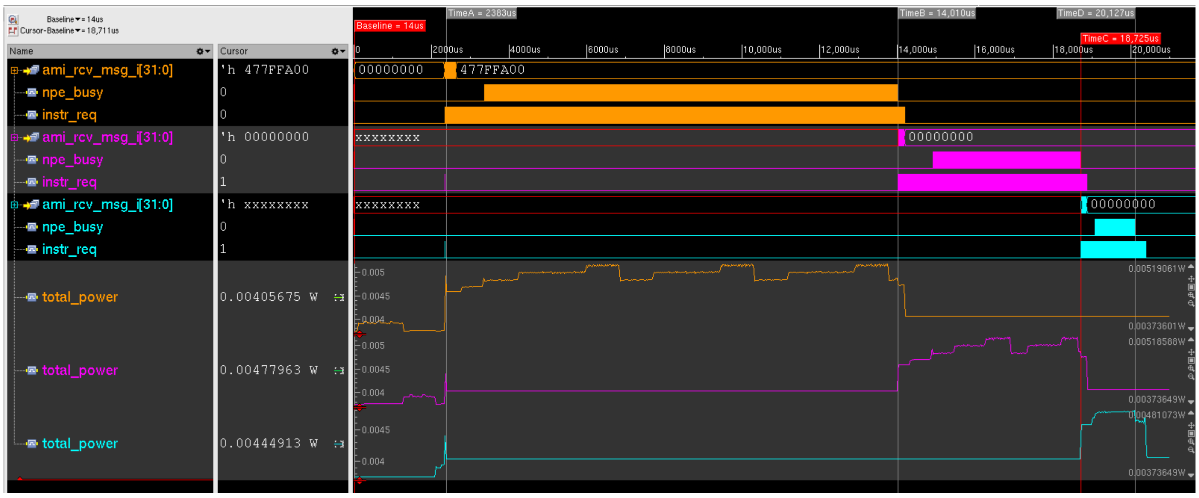


Figure 6.2: Time-based power graph of version 3 together with several signals.

The power consumption of the cores running version 3 of the program was estimated in the time-based mode, and Figure 6.2 shows the resulting graph. As before, the cores consume most power when they are actively executing the neural network operations. A bit less power is consumed when the cores when the cores are communicating with each other. For example, at the time indicated by cursor ”TimeA”, the instantaneous power consumption of core 0 is 4.73 mW, while that of core 1 is 4.58 mW (not shown in the figure). The peak, meanwhile, is around 5.15 mW for all cores. This is an increase from the previous version, although not by a significant amount. To find out where the extra power consumption is coming from, a breakdown of power consumption values per component was performed.

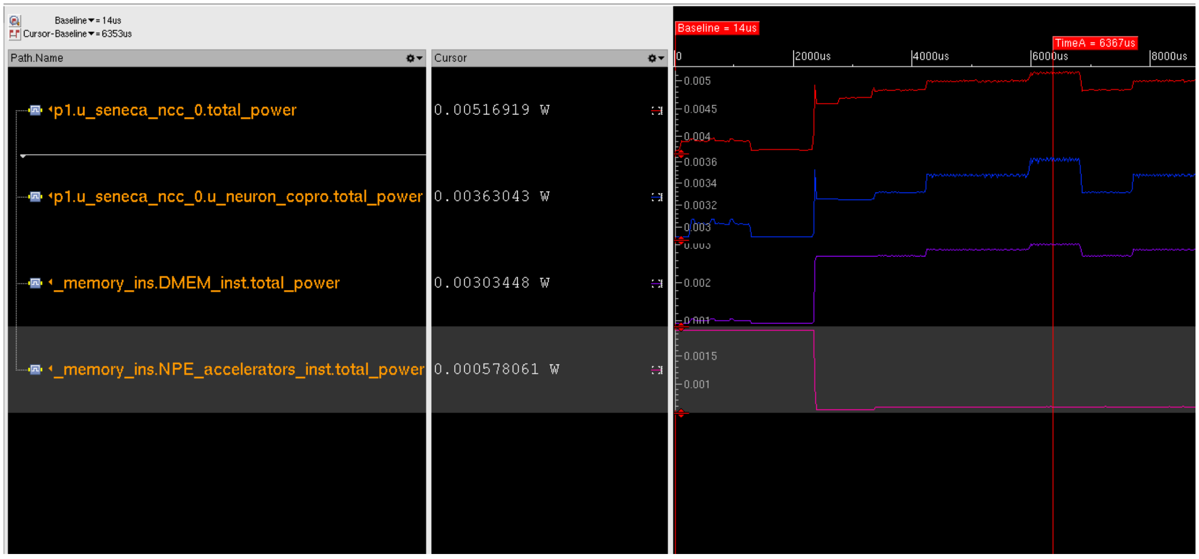

Baseline = 14us
Cursor-Baseline = 6353us
Path.Name
Cursor
p1.u_seneca_ncc_0.total_power
p1.u_seneca_ncc_0.u_neuron_copro.total_power
_memory_ins.DMEM_inst.total_power
_memory_ins.NPE_accelerators_inst.total_power
0.00516919 W
0.00363043 W
0.00303448 W
0.000578061 W
TimeA = 6367us

Figure 6.3: Time-based power graph of version 3, broken down into components (core 1).

Figure 6.3 shows the power consumption graphs of several components. After inspection, we found out that most of the fluctuations and the rise in peak power consumption can be attributed to the data memory. The green graph indicates the total power, while the purple graph shows the power consumption of the data memory alone. The pink graph shows the power consumption of the NPEs, while the blue graph is the sum of the DRAM and NPE power consumption values. Similar to the previous version, average power will also be estimated to obtain more precise numbers.

Table 6.2: Average power consumption of version 3, during the active time of core 1 (Active 1). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_0	25.00%	2.80E-03	5.35E-03	4.86E-04	8.64E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.3: Average power consumption of version 3, during the active time of core 2 (Active 2). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_1	25.00%	2.80E-03	5.35E-03	4.86E-04	8.64E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.4: Average power consumption of version 3, during the active time of core 3 (Active 3). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_2	25.00%	2.80E-03	5.35E-03	4.86E-04	8.64E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

The results of the average power analysis is shown in Tables 6.2, 6.3, 6.4. The values in the tables correspond to the time periods Active 1, Active 2, and Active 3, respectively. Compared to version 2, the average power consumed by the active core increased slightly, while those of the idle cores do not show any noticeable change. The percentages of the leakage, internal, and switching powers also did not change significantly. The increase in power consumption of the active core is attributed to the fact that instructions are executed more frequently by the NPEs. The number of iterations required to complete the matrix multiplication operations decreased by half, as previously mentioned. Between each loop, time is required by the RISC-V processor to execute the branch and jump operations required to realize for loops in C. Reducing the number of these operations decrease the execution time, but that also means that memory access operations by the NPEs become more frequent. Therefore, the power consumption increases.

By plugging in the numbers of Tables 6.2, 6.3, 6.4 to Equation 5.1, we obtain the total energy spent for one inference for version 3. The total energy is 0.000439 Joules, or 0.439 mJ.

6.2. Results of Version 4: Loop Unrolling With 4 Elements

6.2.1. Optimization Method and Implementation

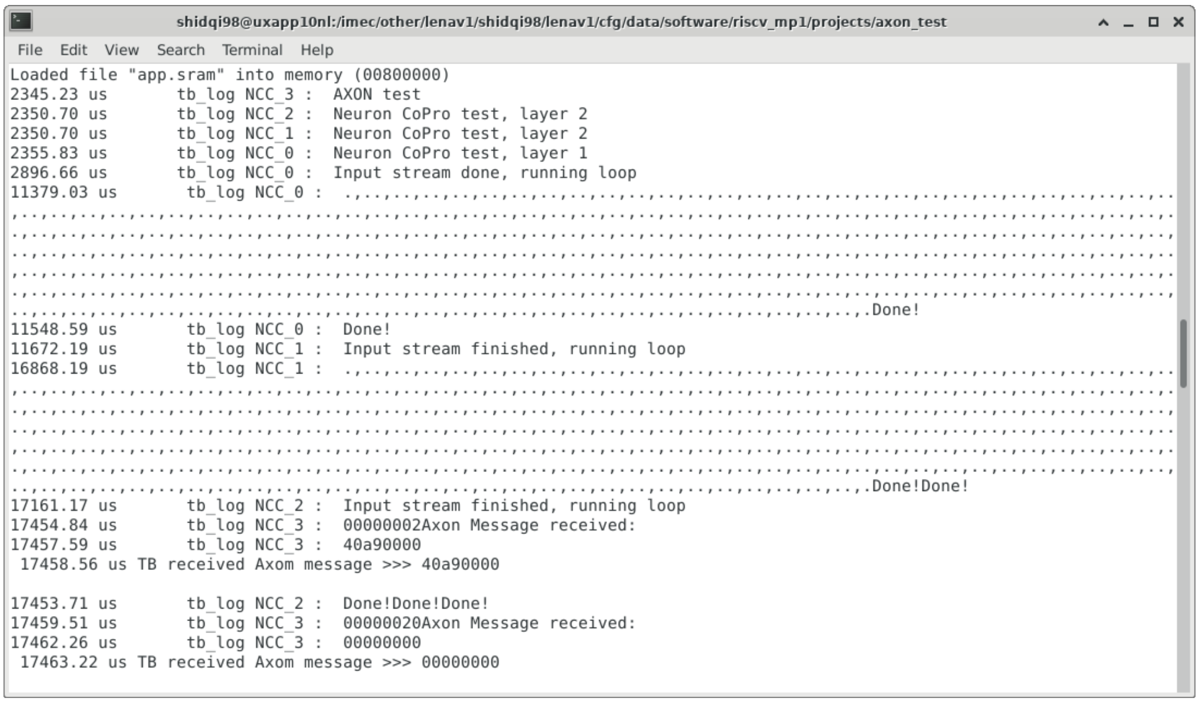
In this version, we seek to utilize the registers in the NPEs even more compared to the previous version, using them to store 4 input values instead of 2. This way we can implement the principle of memory locality to a higher degree, which will hopefully result in a more efficient implementation. Even though the improvement in performance may be offset by the lower number of possible zero-skippings, as explained in the execution time measurement results of the previous version, it would be interesting to find out which one will have the larger effect. As the principle itself was already discussed in the explanation of version 3, it will not be repeated here. The source code to implement the 4-input value loop unrolling algorithm is shown in Listing 6.2. For the sake of brevity, the DO_LOOP function here is omitted, since the change in version 4 is simply adapting to the function KERNEL_SENECA shown here.

Listing 6.2: Source code of NPE Loop of Version 4

```

```

```c
void KERNEL_SENECA(float datapoint, float datapoint2, float datapoint3, float datapoint4)
{
    uint32_t eval_reg = 0u;
    uint32_t eval_reg2 = eval_reg + 1;
    uint32_t eval_reg3 = eval_reg2 + 1;
    uint32_t eval_reg4 = eval_reg3 + 1;

    //Convert first datapoint
    float ev_val_F = datapoint;
    uint16_t ev_val_BF = 0;
    FLOAT2BF(&ev_val_F, &ev_val_BF);

    //Convert second datapoint
    float ev_val_F2 = datapoint2;
    uint16_t ev_val_BF2 = 0;
    FLOAT2BF(&ev_val_F2, &ev_val_BF2);

    //Convert third datapoint
    float ev_val_F3 = datapoint3;
    uint16_t ev_val_BF3 = 0;
    FLOAT2BF(&ev_val_F3, &ev_val_BF3);

    //Convert fourth datapoint
    float ev_val_F4 = datapoint4;
    uint16_t ev_val_BF4 = 0;
    FLOAT2BF(&ev_val_F4, &ev_val_BF4);

    //Write datapoints to registers
    write_to_regfile(eval_reg, ev_val_BF);
    write_to_regfile(eval_reg2, ev_val_BF2);
    write_to_regfile(eval_reg3, ev_val_BF3);
    write_to_regfile(eval_reg4, ev_val_BF4);

    //Setup of weight register 1
    uint32_t weight_reg = eval_reg4 + 1;
    uint32_t weight_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    //Setup of weight register 2
    uint32_t weight_reg2 = weight_reg + 1;
    uint32_t weight2_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    //Setup of weight register 3
    uint32_t weight_reg3 = weight_reg2 + 1;
    uint32_t weight3_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    //Setup of weight register 4
    uint32_t weight_reg4 = weight_reg3 + 1;
    uint32_t weight4_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    //Setup of state register
    uint32_t state_reg = weight_reg4 + 1;
    uint32_t state_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    ++last_miram_2ndHalf_addr_used;

    //1st MAC operation
    uint32_t mul_res_reg = state_reg + 1;
    append_to_miram_command(MUL, eval_reg, weight_reg, mul_res_reg);
    uint32_t new_state_reg = mul_res_reg + 1;
    append_to_miram_command(ADD, mul_res_reg, state_reg, new_state_reg);

    //2nd MAC operation
    append_to_miram_command(MUL, eval_reg2, weight_reg2, mul_res_reg);
    append_to_miram_command(ADD, mul_res_reg, new_state_reg, state_reg);

    //3rd MAC operation
    append_to_miram_command(MUL, eval_reg3, weight_reg3, mul_res_reg);
    append_to_miram_command(ADD, mul_res_reg, state_reg, new_state_reg);

    //4th MAC operation
    append_to_miram_command(MUL, eval_reg4, weight_reg4, mul_res_reg);
    append_to_miram_command(ADD, mul_res_reg, new_state_reg, state_reg);

    //Final write back to memory
    append_to_miram_write_reg_to_mem(state_adr_addr_ptr, state_reg, 0x01u);
}
```

6.2.2. Execution Time Measurements


shidqi98@uxapp10nl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/axon_test
File Edit View Search Terminal Help

```
Loaded file "app.sram" into memory (00800000)
2345.23 us       tb_log NCC_3 :  AXON test
2350.70 us       tb_log NCC_2 :  Neuron CoPro test, layer 2
2350.70 us       tb_log NCC_1 :  Neuron CoPro test, layer 2
2355.83 us       tb_log NCC_0 :  Neuron CoPro test, layer 1
2896.66 us       tb_log NCC_0 :  Input stream done, running loop
11379.03 us       tb_log NCC_0 :  .,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.Done!
11548.59 us       tb_log NCC_0 :  Done!
11672.19 us       tb_log NCC_1 :  Input stream finished, running loop
16868.19 us       tb_log NCC_1 :  .,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..
,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.
.,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,
..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,..,.Done!Done!
17161.17 us       tb_log NCC_2 :  Input stream finished, running loop
17454.84 us       tb_log NCC_3 :  00000002Axon Message received:
17457.59 us       tb_log NCC_3 :  40a90000
 17458.56 us TB received Axom message >>> 40a90000

17453.71 us       tb_log NCC_2 :  Done!Done!Done!
17459.51 us       tb_log NCC_3 :  00000020Axon Message received:
17462.26 us       tb_log NCC_3 :  00000000
 17463.22 us TB received Axom message >>> 00000000
```

Figure 6.4: Output messages obtained from SENeCA while running the simulation with version 4.

By executing the SENeCA simulation with version 4 of the program, the screen depicted in Figure 6.4 was obtained. At first glance, it can be seen that there was indeed a reduction in the execution time of the program. In order to obtain more accurate measurements of the execution times, the simulation dump file was opened with a waveform viewer program. This allows us to know precisely when a certain component (for example the AMI receiver) stops receiving messages, indicating that the receiving of input values was complete. This was done in this version since some parts take only a short time, potentially causing measurements by print statements to be inaccurate. The waveforms, along with the cursors placed in specific milestones, is shown in Figure 6.5.

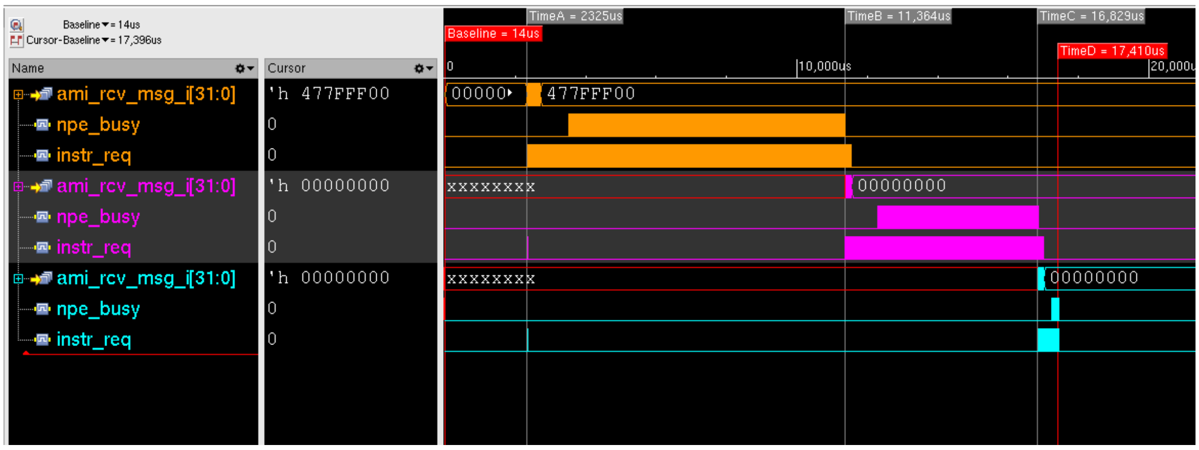


Figure 6.5: Waveforms of the signals produced by SENeCA when simulated using version 4 of the benchmark.

Since not all of the time values indicated by the cursors are visible, the times are presented in Table 6.5, similar to the previous versions. Similar to the change between versions 2 and 3, in this version the main improvement again comes from the execution time of neural network operations of layer 1. Also, the execution time of neural network operations in layer 2 increased, similar to version 3. From this result, we can infer that the execution time of layers 2 and 3 depend highly on the data being used as input. To mitigate this, in the next version of the benchmarking program, the event capture unit (EVC) of SENeCA (refer to Chapter 3) will be used. This component will allow us to leverage the abundant zero outputs of layers 1 and 2 to decrease the execution time. In other words, utilize the sparsity of neural networks.

Table 6.5: Execution time measurements for version 4. All times are in microseconds (μs).

Version 4					
Timestamps		Execution Time			
2349	Start	Loop 1	8648	Sending 1	399
2748	Input stream 1 finished	Loop 2	5340	Sending 2	150
11396	Loop 1 finished	Loop 3	399	Sending 3	177
11546	Input stream 2 finished	Total	14387	Total	726
16886	Loop 2 finished	Grand total			15113
17063	Input stream 3 finished				
17462	Loop 3 finished				

6.2.3. Power Measurements

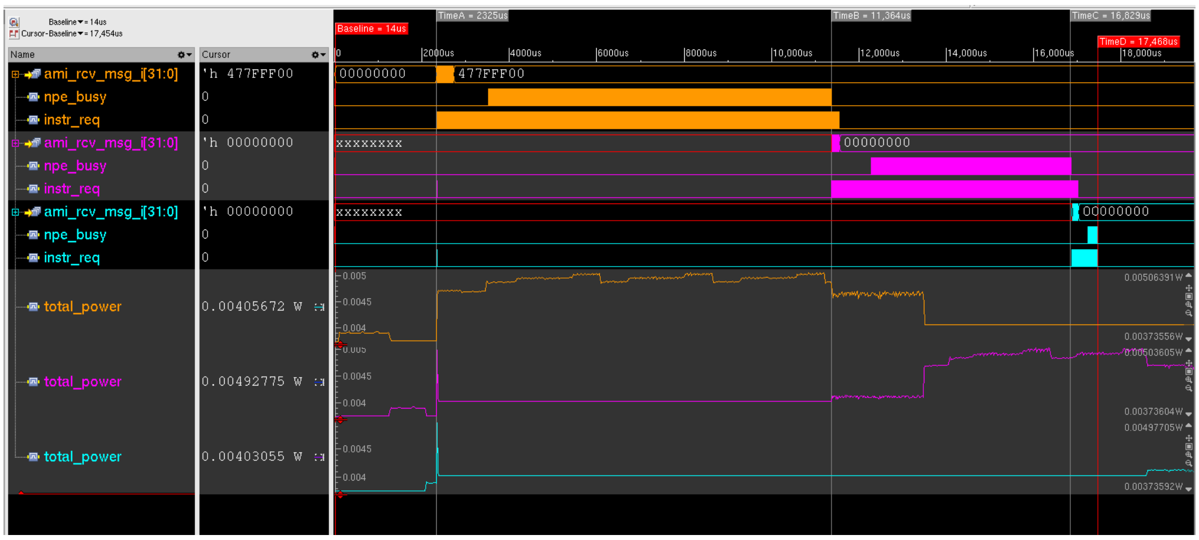


Figure 6.6: Time-based power graph of version 4 together with several signals.

With regards to power consumption, the change from version 2 to version 3 is more pronounced in this version, since the number of instructions accessing the data memory per iteration was doubled by performing 4 MAC operations. The graphs depicting the power consumption with regards to time produced by the time-based power estimation is shown in Figure 6.6. What is also noticeable here, is the fact that the difference between the power consumed by the cores when they are active is not so different than the power consumed when they are idle. To obtain more precise power consumption measurements, average-mode power estimation was run. The results are listed in Tables 6.6, 6.7, 6.8.

Table 6.6: Average power consumption of version 4, during the active time of core 1 (Active 1). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_0	25.00%	2.80E-03	5.50E-03	4.86E-04	8.78E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.7: Average power consumption of version 4, during the active time of core 2 (Active 2). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_1	25.00%	2.80E-03	5.50E-03	4.86E-04	8.78E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.8: Average power consumption of version 4, during the active time of core 3 (Active 3). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_2	25.00%	2.80E-03	5.50E-03	4.86E-04	8.78E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

To obtain the energy consumed for one iteration, Equation 5.1 will again be used. Plugging in the numbers of the execution time and power consumption tables above, we obtain the value of 0.000375 J, or 0.375 mJ. This is less than the energy consumed by the previous version.

6.3. Results of Version 5: Utilization of the Event Capture Unit

In version 5, we will attempt to make use of the sparse nature of neural networks to minimize the execution time and energy consumption, by using SENeCA's Event Capture (EVC) unit, explained in more detail in Chapter 3. The optimization method of version 4, the 4-datapoint loop unrolling, is implemented in an identical fashion.

6.3.1. Optimization Method and Implementation

The idea behind the EVC unit is that not all neurons produce a nonzero output after the application of the activation function to their values after bias addition. Therefore, not all of the neuron values have to be recorded and transmitted to the next layer, as the zero values will not have an effect on the calculations of the next layer. This conserves both memory and time, especially with regards to the sending times between cores. To trigger the EVC unit, a specific command has to be executed by the NPE, called EVC. It is listed in Table 3.1. When the EVC command is executed, the register that is passed as the operand will be checked if it contains a nonzero value. If it contains a nonzero value, the EVC unit will copy the neuron value and write three values into a register that is readable by the RISC-V core. The three values are the neuron value itself, the ID number of the source NPE, as well as a configurable tag number. The combination of the NPE ID number and the tag number allows the RISC-V core to determine which neuron the output value came from. Afterwards, the EVC unit will trigger an interrupt to the RISC-V core, allowing it to read the three values written by the EVC unit.

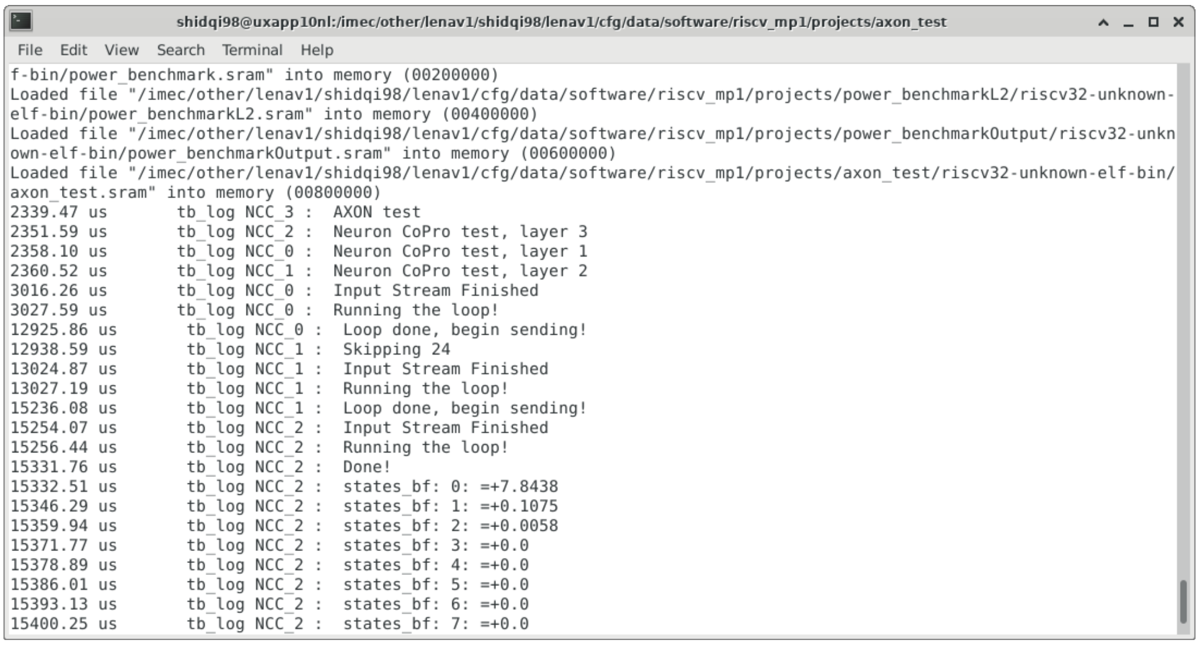
Listing 6.3 shows the source code to implement the concept explained above. The fucntion that is changed is KERNEL_SENECA_B, the function containing the commands to execute the bias addition and activation function application. Line 24 shows the addition of the EVC command. The first operand, the mul_res_reg, is holds the neuron value at the end of the process and thus will be checked by the EVC unit. The second and third operands determine the tag value. The second function, neuron_spikes_handler is the interrupt handler function called when the EVC unit triggers an interrupt to the RISC-V core. In the handler, the neuron value, tag value, and NPE ID number are read from the designated register. Then the neuron number is identified. However, since the neurons with zero output values are skipped, we cannot store the neuron values in a simple array like we did in the previous versions. A new data structure to hold both the neuron numbers and their values was created, named result_list (line 1). This structure is used at the end of the handler function to store the neuron numbers and values.

Listing 6.3: Modified source code part of the NPE Loop of Version 5

```
1  struct result_list {
2      uint16_t neuron_no;  /* Addr 0xFFFF = End Of Stream */
3      int16_t value;
4  };
5
6  void KERNEL_SENECA_B()
7  {
8      last_miram_1stHalf_addr_used = 0x7;
9      last_miram_2ndHalf_addr_used = 0x17;
10     uint32_t zero_reg = 0u;
11     float ev_val_F = 0.0;
12     uint16_t ev_val_BF = 0;
13     FLOAT2BF(&ev_val_F, &ev_val_BF);
14
15     write_to_regfile(zero_reg, ev_val_BF);
16
17     uint32_t temp_reg = zero_reg + 1;
18     uint32_t temp_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
19     ++last_miram_2ndHalf_addr_used;
20     uint32_t bias_reg = temp_reg + 1;
21     uint32_t bias_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
22     ++last_miram_2ndHalf_addr_used;
23     uint32_t add_res_reg = bias_reg + 1;
24     append_to_miram_command(ADD, bias_reg, temp_reg, add_res_reg);
25     uint32_t comp_res_reg = add_res_reg + 1;
26     append_to_miram_command(GTH, add_res_reg, zero_reg, comp_res_reg);
```

```
    uint32_t mul_res_reg = comp_res_reg + 1;
    append_to_miram_command(MUL, add_res_reg, comp_res_reg, mul_res_reg);
    append_to_miram_command(EVC, mul_res_reg, 0u, 8u);
}

ATTR_INTR void neuron_spikes_handler()
{
        DEV_WRITE(NEURO_COPRO_BASE+136*4, 0u); //clear the intrrupt
        uint32_t events_val = 0u;
    uint32_t npe_id = 0u;
    uint32_t neuron_number = 0u;

        while(DEV_READ(NEURO_COPRO_BASE+133*4) != 0x0)
        {
            events_val = DEV_READ(NEURO_COPRO_BASE+130*4);
        npe_id = DEV_READ(NEURO_COPRO_BASE+131*4);
        tag = DEV_READ(NEURO_COPRO_BASE+132*4);
        neuron_number = tag + npe_id;
        neuron_list[neuron_counter].neuron_no = neuron_number;
        neuron_list[neuron_counter].value = events_val;
            neuron_counter++;
    }
}
```

6.3.2. Execution Time Measurements


shidqi98@uxapp10nl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/axon_test
File Edit View Search Terminal Help

```
f-bin/power_benchmark.sram" into memory (00200000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_benchmarkL2/riscv32-unknown-
elf-bin/power_benchmarkL2.sram" into memory (00400000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_benchmarkOutput/riscv32-unkn
own-elf-bin/power_benchmarkOutput.sram" into memory (00600000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/axon_test/riscv32-unknown-elf-bin/
axon_test.sram" into memory (00800000)
2339.47 us      tb_log NCC_3 :  AXON test
2351.59 us      tb_log NCC_2 :  Neuron CoPro test, layer 3
2358.10 us      tb_log NCC_0 :  Neuron CoPro test, layer 1
2360.52 us      tb_log NCC_1 :  Neuron CoPro test, layer 2
3016.26 us      tb_log NCC_0 :  Input Stream Finished
3027.59 us      tb_log NCC_0 :  Running the loop!
12925.86 us      tb_log NCC_0 :  Loop done, begin sending!
12938.59 us      tb_log NCC_1 :  Skipping 24
13024.87 us      tb_log NCC_1 :  Input Stream Finished
13027.19 us      tb_log NCC_1 :  Running the loop!
15236.08 us      tb_log NCC_1 :  Loop done, begin sending!
15254.07 us      tb_log NCC_2 :  Input Stream Finished
15256.44 us      tb_log NCC_2 :  Running the loop!
15331.76 us      tb_log NCC_2 :  Done!
15332.51 us      tb_log NCC_2 :  states_bf: 0: =+7.8438
15346.29 us      tb_log NCC_2 :  states_bf: 1: =+0.1075
15359.94 us      tb_log NCC_2 :  states_bf: 2: =+0.0058
15371.77 us      tb_log NCC_2 :  states_bf: 3: =+0.0
15378.89 us      tb_log NCC_2 :  states_bf: 4: =+0.0
15386.01 us      tb_log NCC_2 :  states_bf: 5: =+0.0
15393.13 us      tb_log NCC_2 :  states_bf: 6: =+0.0
15400.25 us      tb_log NCC_2 :  states_bf: 7: =+0.0
```

Figure 6.7: Output messages obtained from SENeCA while running the simulation with version 5.

Figure 6.7 shows the simulation screen when running version 5 of the benchmark. Again, by observing the timestamps on the screen, we can infer the timing at which the parts of the program were completed. Table 6.9 shows these numbers, as well as the execution time obtained by subtracting appropriate timestamps. Looking at the overall execution time, we can see that there is a reduction by around 3000 μs, around 20% of the total execution time of version 4. This can mainly be attributed to the decreased execution times of layers 2 and 3. While zero-skipping was implemented in all previous versions, they were flawed since the fact that multiple input values are being processed in a single iteration (loop unrolling) prevented all zeros from being skipped. This version, however, implements it in a way that allows all zeros to be skipped while still allowing the speedup from loop unrolling. The sending time also decreased, since there are simply fewer values to send.

Table 6.9: Execution time measurements for version 5. All times are in microseconds (μs).

Version 5					
Timestamps		Execution Time			
2358	Start	Loop 1	9909	Sending 1	658
3016	Input stream 1 finished	Loop 2	2212	Sending 2	99
12925	Loop 1 finished	Loop 3	77	Sending 3	18
13024	Input stream 2 finished	Total	12198	Total	775
15236	Loop 2 finished	Grand total			12973
15254	Input stream 3 finished				
15331	Loop 3 finished				

Figure 6.8 shows the waveforms of the signals produced by SENeCA during simulation. Similar to the previous versions, there are three signals picked from each core, showing when the AMIs, NPEs, and RISC-V processors of each core are active. The cursors correspond roughly to times when one core stops processing and the next one starts. From this image we can clearly see that the execution time of layer 1, indicated by the signal NPE_busy that is colored orange, is the one that dominates the total execution time. Therefore, for the next iteration, tackling the execution time of layer 1 by analyzing if there are any inefficiencies might be worthwhile.

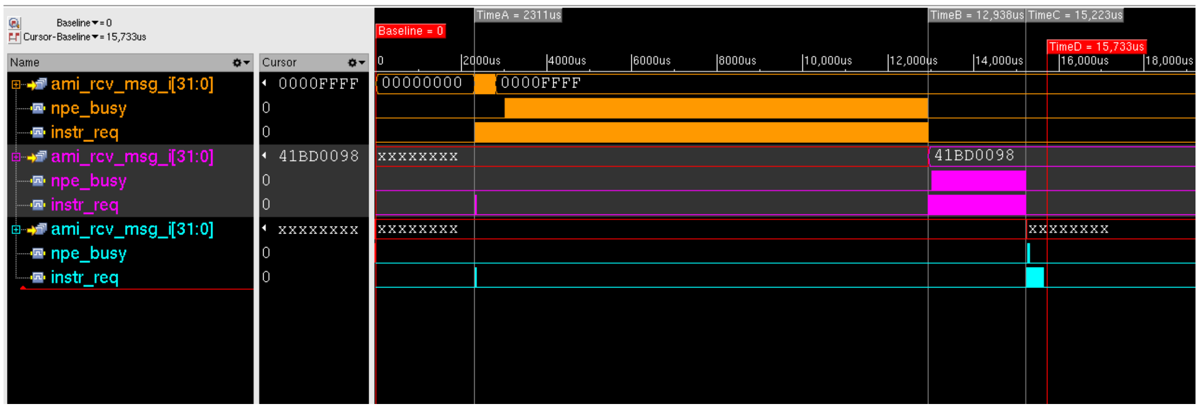


Figure 6.8: Waveforms of the signals produced by SENeCA when simulated using version 5 of the benchmark.

By observing more closely Table 6.9, we can see that the execution time of layer 1 actually increased by around 1300 μs. While this increase is not fatal as the total execution time is still lower than version 4, the cause for this increase could be investigated to determine the next possible optimization. By zooming in on the part where core 1 finishes its process and starts sending data to core 2, we can observe how the usage of the EVC unit can cause the execution time to decrease. Figure 6.9 shows the result of magnification. By observing the orange-colored npe_busy and instr_req signals, we can see that the NPEs are busy and idle in an interleaved fashion. Since context switching happens every time the NPEs starts or stops execution (by triggering an interrupt), time is spent. The addition of the EVC adds the required number of context switching needed, since the EVC also triggers interrupt. A way to reduce the the number of context switching can significantly reduce the execution time of the neural network operations.

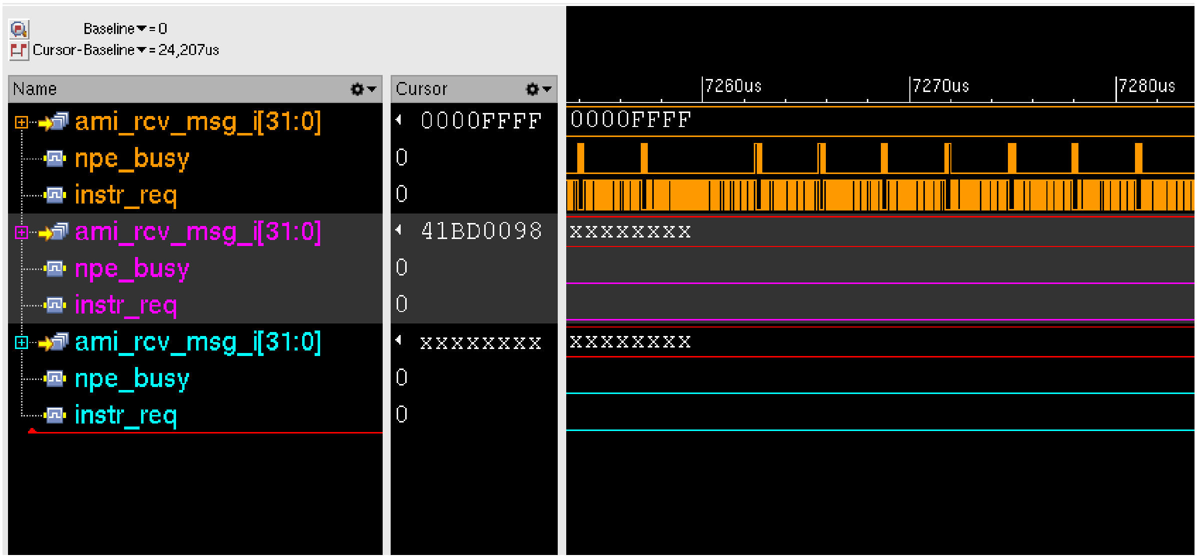


Figure 6.9: The time period in which cores 2 and 3 are active is magnified here to provide a more detailed look.

6.3.3. Power Measurements

Similar to previous versions, time-based power estimation was performed for version 5. The resulting graph can be observed in Figure 6.10. The power graphs were added to the waveforms presented in Figure 6.8, to better display the relation between the activities of the core and its respective power consumption.

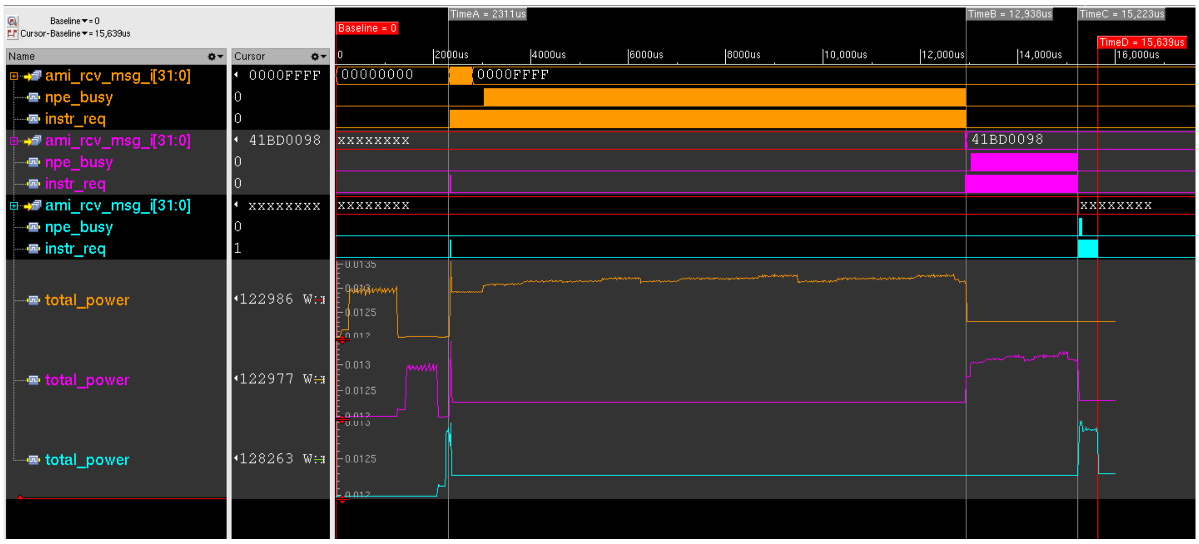


Figure 6.10: Time-based power graph of version 5 together with several signals.

By looking at the power graphs, we can observe that the power consumption fluctuates between 0.012 W and 0.013 W per core, similar to version 4. To obtain more accurate numbers, average mode power estimation was again done on the time periods of Active 1, Active 2, and Active 3 of version 5. The results can be observed in Tables 6.10, 6.11, 6.12. The difference in power consumption between the active and idle cores are similar to what we found in version 4, around 0.1 W. This indicates that the use of the EVC does not affect the power consumption of SENeCA by a significant amound.

Table 6.10: Average power consumption of version 5, during the active time of core 1 (Active 1). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_0	25.00%	2.80E-03	5.60E-03	4.86E-04	8.89E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.11: Average power consumption of version 5, during the active time of core 2 (Active 2). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_1	25.00%	2.80E-03	5.60E-03	4.86E-04	8.89E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.12: Average power consumption of version 5, during the active time of core 3 (Active 3). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_2	25.00%	2.80E-03	5.60E-03	4.86E-04	8.89E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

By plugging in the total power consumption values found in the tables above in equation 5.1, we can calculate the total energy of one inference using version 5 with SENeCA. That number turns out to be 0.000323 J, or 0.323 mJ. This less than version 4, indicating that the extra energy required by the use of the EVC unit is more than compensated by the reduction in the execution time.

6.4. Results of Version 6: Utilization of the Loop Buffer

Analysis of the previous version showed that the time for context switching between the NPE process and the RISC-V process can be a significant bottleneck. In this version, we will attempt to minimize the spent time by deploying another component of SENeCA described in Chapter 3, the loop buffer.

6.4.1. Optimization Method and Implementation

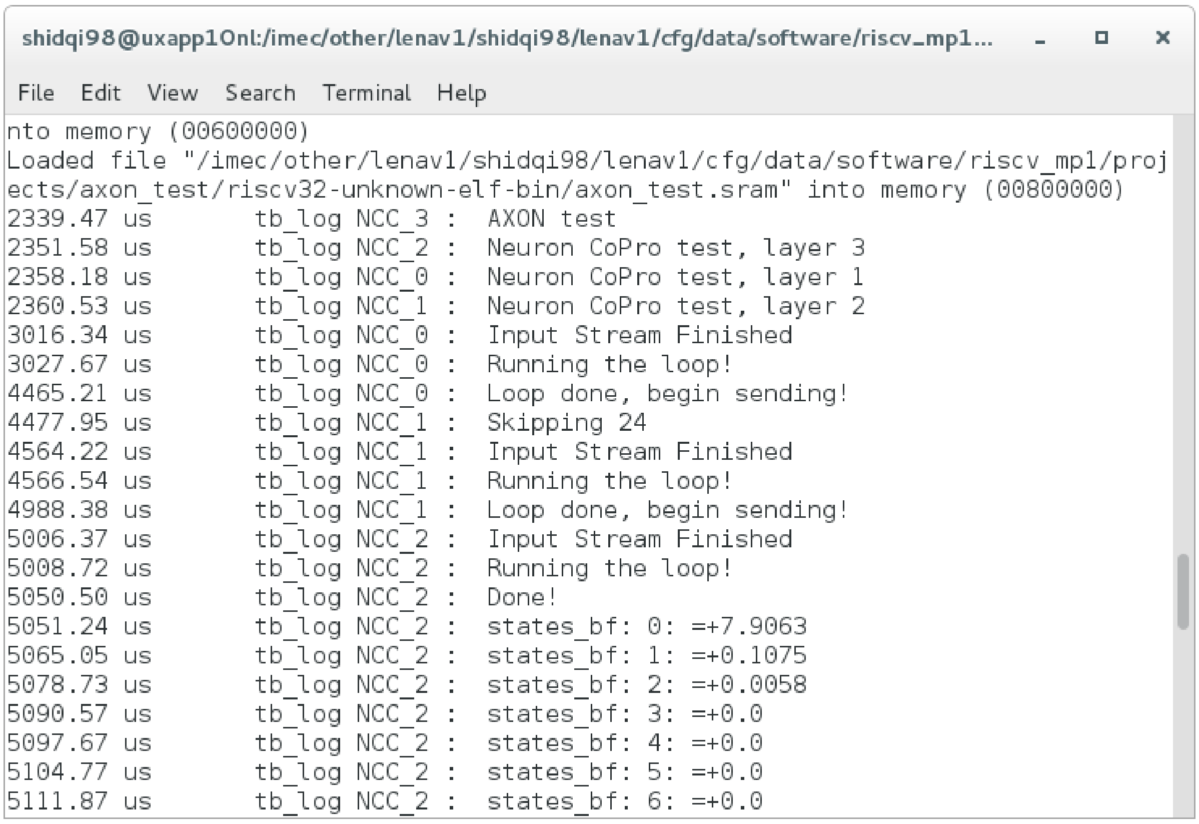
The software implementation of the program that uses the loop buffer does not differ much from the previous version. The only function changed is DO_LOOP, the function that actually configures the data that will be written to the instruction kernel. The modified part is shown in Listing 6.4. Specifically, line 31 shows the macro that combines all of the instructions written previously and adds other parameters such as the starting instruction, final instruction, and number of repetitions. To make use of the loop buffer, we only need to change the number of repetitions to suit the number of inputs. For example, this function is taken from layer 2, where the number of inputs is 256, and since there are 8 NPEs, 32 repetitions are done by each NPE to process all of the incoming data. Hence, we put 32u as the first parameter of LOOP_CONFIG_INST. While the change to the software might be simple, the reduction of the number of context switches that can be achieved with this method will help to reduce the total execution time by a significant amount.

Listing 6.4: Modified source code part of the NPE Loop of Version 6

```
1 void DO_LOOP(void * beginning_of_weightsV, void* beginning_of_statesV, void *
      beginning_of_weightsV2, void * beginning_of_weightsV3, void * beginning_of_weightsV4)
2 {
3     last_miram_1stHalf_addr_used = 0x0;
4     last_miram_2ndHalf_addr_used = (MIRAM_1ST_HALF_LAST_INDEX);
5     uint32_t eval_reg = 0u;
6     uint32_t eval_reg_2 = eval_reg + 1;
```

```
    uint32_t eval_reg_3 = eval_reg_2 + 1;
    uint32_t eval_reg_4 = eval_reg_3 + 1;

    uint32_t weight_reg = eval_reg_4 + 1;
    uint32_t weight_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr, beginning_of_weightsV,
        weight_reg, 0x01u);

    uint32_t weight_reg2 = weight_reg + 1;
    uint32_t weight_adr_addr_ptr2 = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr2, beginning_of_weightsV2,
        weight_reg2, 0x01u);

    uint32_t weight_reg3 = weight_reg2 + 1;
    uint32_t weight_adr_addr_ptr3 = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr3, beginning_of_weightsV3,
        weight_reg3, 0x01u);

    uint32_t weight_reg4 = weight_reg3 + 1;
    uint32_t weight_adr_addr_ptr4 = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr4, beginning_of_weightsV4,
        weight_reg4, 0x01u);

    uint32_t state_reg = weight_reg4 + 1;
    uint32_t state_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(state_adr_addr_ptr, beginning_of_statesV, state_reg,
         0x00u);

    last_miram_2ndHalf_addr_used += 9u;
    uint32_t loopcfg = LOOP_CONFIG_INST(32u, last_miram_2ndHalf_addr_used,
        MIRAM_1ST_HALF_LAST_INDEX+0x01u);
    WriteLoopConfig(loopcfg);

    WFI();
}
```

6.4.2. Execution Time Measurements


shidqi98@uxapp10nl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1...
File Edit View Search Terminal Help

```
nto memory (00600000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/proj
ects/axon_test/riscv32-unknown-elf-bin/axon_test.sram" into memory (00800000)
2339.47 us      tb_log NCC_3 :  AXON test
2351.58 us      tb_log NCC_2 :  Neuron CoPro test, layer 3
2358.18 us      tb_log NCC_0 :  Neuron CoPro test, layer 1
2360.53 us      tb_log NCC_1 :  Neuron CoPro test, layer 2
3016.34 us      tb_log NCC_0 :  Input Stream Finished
3027.67 us      tb_log NCC_0 :  Running the loop!
4465.21 us      tb_log NCC_0 :  Loop done, begin sending!
4477.95 us      tb_log NCC_1 :  Skipping 24
4564.22 us      tb_log NCC_1 :  Input Stream Finished
4566.54 us      tb_log NCC_1 :  Running the loop!
4988.38 us      tb_log NCC_1 :  Loop done, begin sending!
5006.37 us      tb_log NCC_2 :  Input Stream Finished
5008.72 us      tb_log NCC_2 :  Running the loop!
5050.50 us      tb_log NCC_2 :  Done!
5051.24 us      tb_log NCC_2 :  states_bf: 0: =+7.9063
5065.05 us      tb_log NCC_2 :  states_bf: 1: =+0.1075
5078.73 us      tb_log NCC_2 :  states_bf: 2: =+0.0058
5090.57 us      tb_log NCC_2 :  states_bf: 3: =+0.0
5097.67 us      tb_log NCC_2 :  states_bf: 4: =+0.0
5104.77 us      tb_log NCC_2 :  states_bf: 5: =+0.0
5111.87 us      tb_log NCC_2 :  states_bf: 6: =+0.0
```

Figure 6.11: Output messages obtained from SENeCA while running the simulation with version 6.

The simulation screen of version 6 is depicted by Figure 6.11. Similar to previous versions, we can infer the timestamps of completions of parts of the benchmark, and by finding the time difference between these timestamps we can infer the execution time. The results of these calculations are shown in Table 6.13. By simply observing the total execution time for this version and comparing it to the previous version (version 5) we can see that there is a massive improvement in the execution time. This improvement is the most significant when compared to the improvements achieved by the previous methods, as the resulting execution time is less than four times that of its predecessor. Layers 1 and 2 both receive a sizable reduction in their execution times, with the execution time of layer 1 being around 15% of its previous iteration, while the execution time of layer 2 is a more modest 20% of its predecessor. Layer 3, meanwhile, sees its execution time actually increase, but since it was insignificant compared to the total execution time to begin with, the increase does not significantly affect the outcome.

Table 6.13: Execution time measurements for version 6. All times are in microseconds (μs).

Version 6						
Timestamps		Execution Time				
2358	Start	Loop 1	1449	Sending 1	658
3016	Input stream 1 finished	Loop 2	424	Sending 2	99
4465	Loop 1 finished	Loop 3	44	Sending 3	18
4564	Input stream 2 finished	Total	1917	Total	775
4988	Loop 2 finished	Grand total			2692
5006	Input stream 3 finished				
5050	Loop 3 finished				

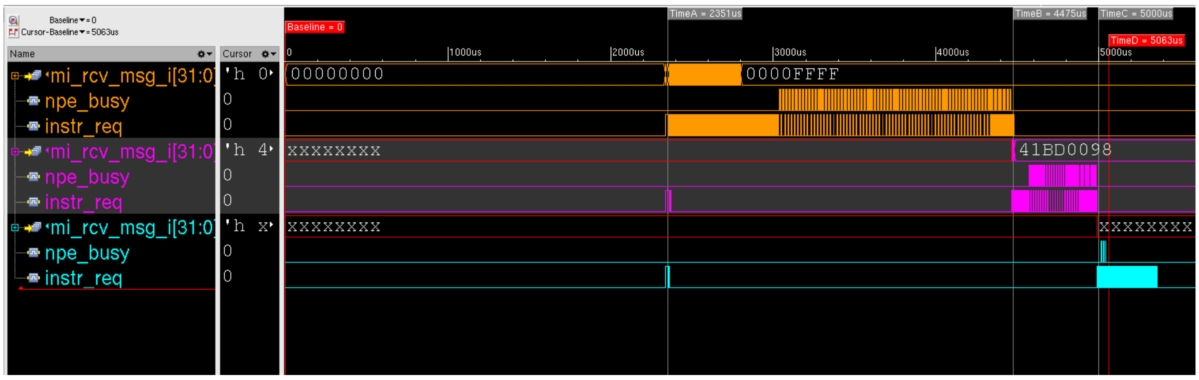


Figure 6.12: Waveforms of the signals produced by SENeCA when simulated using version 6 of the benchmark.

To look into how further optimizations might be possible, once again we will have a look at the waveforms produced by SENeCA when running the benchmarking program. Figure 6.12 shows the waveforms of the three cores. Again, the waveforms are color-coded by the names (shown on the far left column), where orange denotes core 1, purple denotes core 2, and turquoise denotes core 3. The cursors are placed at the approximate timestamps when one core stops the execution and another core takes over. Cursor TimeC meanwhile, is placed at the time when core 3 is finished with the execution and starts writing the results. We can see that after cursor TimeC the signal named npe_busy turns off (because it is done processing the data), while instr_req (which denotes the signal produced by the RISC-V core when it is requesting an instruction) is still active. This is due to the fact that after TimeC, the core still writes the results to the screen, but since this is not part of the execution, it is omitted in the calculations for the execution time. Similar to previous versions, we see that the longest execution time is again the NPE loops of layer 1, so a deeper look into what is happening there might yield more insight.

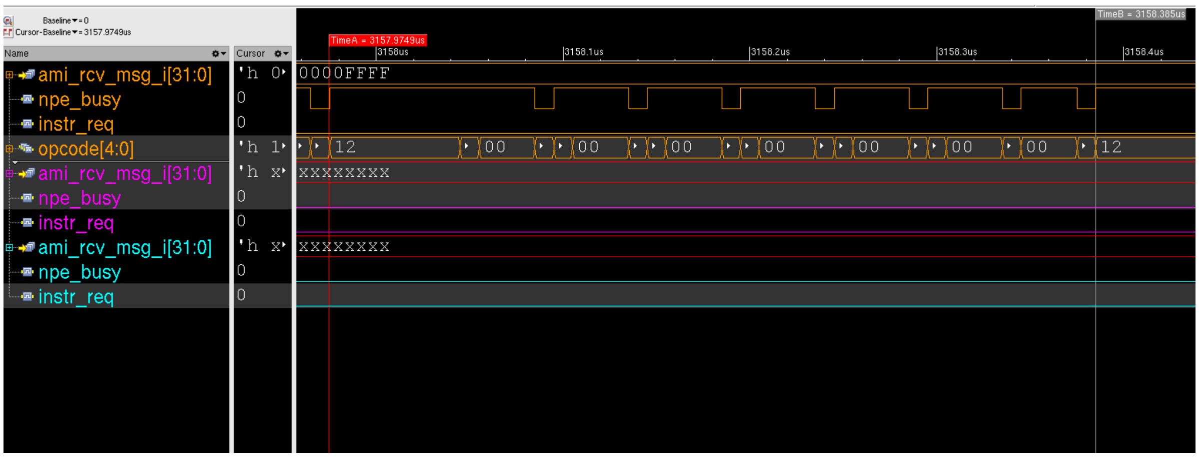


Figure 6.13: A more detailed look of the MM loop pipeline in version 6.

Figure 6.13, meanwhile, shows the resulting waveforms if we zoom in and focus onto a single loop iteration of layer 1. To know what operation is being executed at any given time, the signal opcode[4:0] is added. These codes represent the operation that is executed by the NPEs. To know which operation each hex number represents, refer to Table 3.1. The cursor labeled Baseline is placed on the start of an NPE loop. The instruction that happens directly after has a code 0x12, which denotes loading from memory. Since we are loading 4 data values from memory (4 weight values and 1 neuron state value) without any data dependencies, the load commands is expected to run without any pipeline stalls between each other. However, we observe that there is one clock cycle in which the instruction with the code of 0x00 is executed. This indicates a pipeline stall, and is caused by a minor bug in the hardware. This will be fixed in the following version.

After five cycles of loading data memory, the next set of instructions, MAC, can be executed. Indeed, we find that instruction 0x03 (multiplication) is executed after 0x12. As explained in Chapter 3 however, each instruction produces its result only after 4 clock cycles, so another instruction trying to access a register that will contain the result of a previous incomplete operation will be stalled until that result is available. Therefore, the next operation, which is the addition operation to accumulate the multiplication result with the current neuron value, is stalled by 3 cycles. Indeed, after 3 cycles of 0x00, we can find the instruction coded 0x01 being executed. Due to these stalls, the execution time can be longer that it needs to be, providing us with a window of optimization.In total, this entire loop for the matrix multiplication operations takes 42 clock cycles. Indeed, for the next version, pipeline optimization will be one of the methods implemented.

6.4.3. Power Measurements

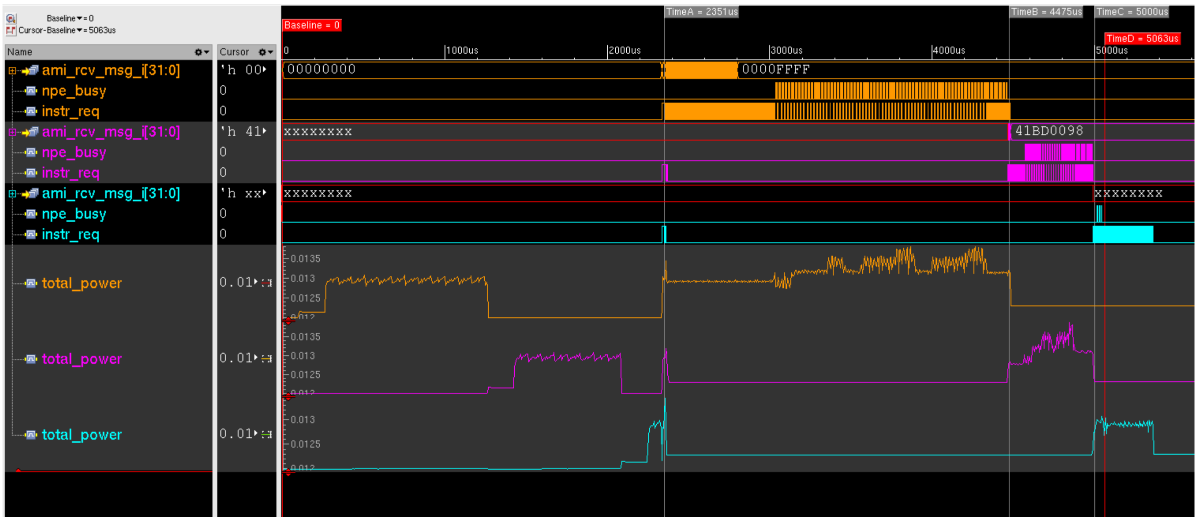


Figure 6.14: Time-based power graph of version 6 together with several signals.

To know the power consumption of SENeCA while executing version 6, we do a time-based power analysis once again. The results can be seen in Figure 6.14. Looking at the approximate level of power consumption that the cores consume when active (around 0.012 to 0.013 W), we can see that the power consumption has increased compared to the previous version. This is to be expected, since with the decrease of time that is spent in context switching, more power is spent doing the actual calculations. These operations consume more power since they involve more components, such as the NPEs, loop buffers, and memory, compared to the context switches that only involve the RISC-V core. To get a better understanding of the power consumption, we once again run average power analysis over the three timespans, active 1, active 2, and active 3.

Table 6.14: Average power consumption of version 6, during the active time of core 1 (Active 1). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_0	25.00%	2.80E-03	5.51E-03	4.86E-04	8.80E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.15: Average power consumption of version 6, during the active time of core 2 (Active 2). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_1	25.00%	2.80E-03	5.51E-03	4.86E-04	8.80E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.16: Average power consumption of version 6, during the active time of core 3 (Active 3). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_2	25.00%	2.80E-03	5.51E-03	4.86E-04	8.80E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

The results of the average power analysis are shown in Tables 6.14, 6.15, 6.16. As noted before, the power consumption did increase slightly. However, since the execution time is much shorter, the expected energy consumption is also expected to drop. By inserting the numbers in the tables above to Equations 5.1, we obtain the total energy consumed for one inference to be 0.000066 J, or 0.066 mJ.

6.5. Results of Version 7: Final Version

In this section, Version 7, the final version in this study, will be presented. Similar to the other versions, the execution time, power consumption, and energy consumption will be measured. Additionally, the ideal energy consumption will be calculated, and an accuracy measurement will be performed to ascertain that the achieved decrease in energy consumption does not come at the cost of accuracy.

6.5.1. Optimization Method and Implementation

As mentioned previously, in this version the main optimization will target the pipeline, namely by eliminating pipeline stalls. Most of the stalls happening in version 6 involve read-before-write hazards [80]. These occur when an instruction wants to read a register before a previous instruction has written into it (since each instruction requires 4 cycles to complete). To avoid this, we must arrange the instructions in such a way as to minimize the data dependencies. Listing 6.5 shows the modified part of the source code that implements this change.

Since the most time-consuming part of the program is the matrix multiplication loop, we will focus on eliminating the pipeline stalls in that loop. Version 6 uses the same loop explained in version 4 (Listing 6.2). Since each loop iteration deals with 4 input values, there are 4 MUL operations and 4 ADD operations. In version 4, the ADD operation for one data value takes place after the MUL operation for the same data value. Since the ADD operation depends on the results of the MUL operation, this will generate a pipeline stall (visible in Figure 6.13). To remedy this, the loop in this version rearranges the MUL and ADD operations. To avoid data dependency, all 4 of the MUL operations can be placed in succession (since none of them involve the same operand), followed by the 4 ADD operations. To accommodate this, extra registers are needed, but since there are unused registers in the NPEs anyway, this comes at no extra cost. Also, since the ADD operations depend on each other, some pipeline stalls are inevitable.

This concept is implemented in the source code shown in Listing 6.5 as follows. Lines 4 to 34 remain unchanged from version 4. The extra registers required to store the multiplication results are declared in lines 37 to 40. Since there are 4 multiplications, a register is declared for each of them to ensure that none of them are dependent upon one another. The multiplication operations are written to the instruction kernel in lines 43 to 46. Note that all operands show up exactly once. At lines 49 and 50, two new registers are declared to store the addition results, reducing the number of stalls between the addition operations. The first addition is scheduled to begin execution three cycles after the second multiplication, upon whose data it depends (mul_res_reg2). Therefore, the addition stalls for 1 cycle to wait for the result. The second addition, meanwhile, needs the result of the fourth multiplication (mul_res_reg4), scheduled 2 cycles before it. Therefore, it stalls for 2 cycles before beginning execution. Similar to the second addition, the third addition also needs the result of an instruction two cycles before it (new_state_reg, result of addition 1), so it will stall for 2 cycles as well. The final addition, meanwhile, is dependent on the addition directly preceding it (mul_res_reg), so it will stall for 3 cycles. The final writeback to memory is in a similar situation, stalling for 3 cycles. In total, the entire MM loop should take 23 clock cycles to complete.

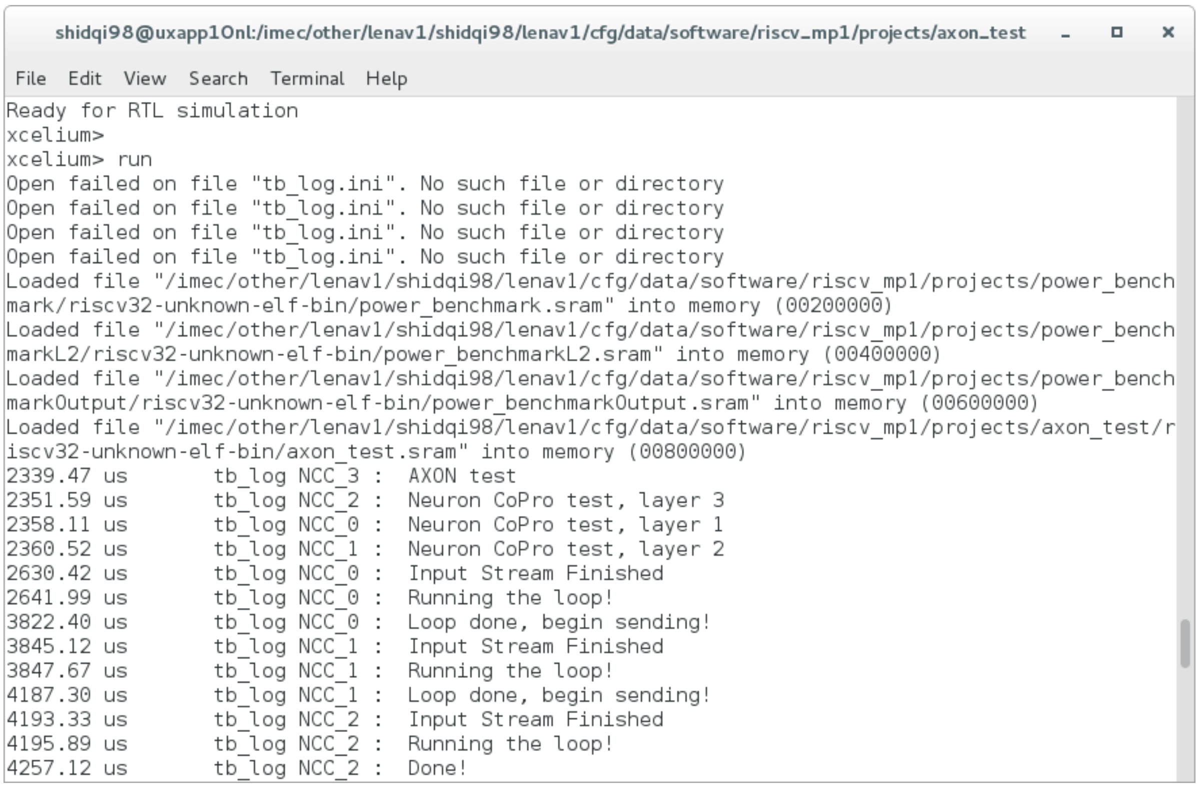
One other change in the implementation is the main function run_kernel_seneca() is structured. In this version, the function KERNEL_SENECA() contains all of the commands that remain constant between iterations of the loop that iterates through the input values. For example, the MUL instructions always use the same operands, so they do not have to be rewritten every loop iteration. Line 113 of Listing 6.5 shows the placement of the KERNEL_SENECA function, outside of the for loops. The function DO_LOOP, meanwhile, contains the commands which need to be executed every loop iteration. For example, the input values need to be written to the NPE registers every iteration (lines 67 to 70), as well as the commands to read/write, since the addresses change (lines 77 to 95). This function is placed inside the for loop (line 128). This avoids unnecessary repetition of instructions that occurred in the source code of version 4. One final minor modification is to the interrupt handler of the Axon Messaging Interface (function ami_handler). The added line (line 129 of Listing 6.5) changes the way the core receives the data from the AMI. Previously, the interrupt is triggered every time a new message arrives, with the handler only receiving one input value every time it is executed. In this version, the interrupt handler consumes all of the contents of the FIFO buffer of the AMI, receiving all messages in one interrupt. This reduces the number of context switches.

Listing 6.5: Modified source code part of the NPE Loop of Version 7

```
1
2  void KERNEL_SENECA()
3  {
4      last_miram_1stHalf_addr_used = 0x0; // IT CAN NOT BE LARGER THAN
           MIRAM_1ST_HALF_LAST_INDEX
5      last_miram_2ndHalf_addr_used = (MIRAM_1ST_HALF_LAST_INDEX);
6      uint32_t eval_reg = 0u;
7      uint32_t eval_reg2 = eval_reg + 1;
8      uint32_t eval_reg3 = eval_reg2 + 1;
9      uint32_t eval_reg4 = eval_reg3 + 1;
10
11     //Setup of weight register 1
12     uint32_t weight_reg = eval_reg4 + 1;
13     uint32_t weight_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
14     ++last_miram_2ndHalf_addr_used;
15
16     //Setup of weight register 2
17     uint32_t weight_reg2 = weight_reg + 1;
18     uint32_t weight2_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
19     ++last_miram_2ndHalf_addr_used;
20
21     //Setup of weight register 3
22     uint32_t weight_reg3 = weight_reg2 + 1;
23     uint32_t weight3_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
24     ++last_miram_2ndHalf_addr_used;
25
26     //Setup of weight register 4
27     uint32_t weight_reg4 = weight_reg3 + 1;
28     uint32_t weight4_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
29     ++last_miram_2ndHalf_addr_used;
30
31     //Setup of state register
32     uint32_t state_reg = weight_reg4 + 1;
33     uint32_t state_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
34     ++last_miram_2ndHalf_addr_used;
35
36     //Multiply result registers
37     uint32_t mul_res_reg = state_reg + 1;
38     uint32_t mul_res_reg2 = mul_res_reg + 1;
39     uint32_t mul_res_reg3 = mul_res_reg2 + 1;
40     uint32_t mul_res_reg4 = mul_res_reg3 + 1;
41
42     //Multiplication operations
43     append_to_miram_command(MUL, eval_reg, weight_reg, mul_res_reg);
44     append_to_miram_command(MUL, eval_reg2, weight_reg2, mul_res_reg2);
45     append_to_miram_command(MUL, eval_reg3, weight_reg3, mul_res_reg3);
46     append_to_miram_command(MUL, eval_reg4, weight_reg4, mul_res_reg4);
```

```

    //New state register
    uint32_t new_state_reg = mul_res_reg + 1;
    uint32_t new_state_reg2 = new_state_reg + 1;

    //Additions
    append_to_miram_command(ADD, mul_res_reg, mul_res_reg2, new_state_reg);
    append_to_miram_command(ADD, mul_res_reg3, mul_res_reg4, new_state_reg2);
    append_to_miram_command(ADD, new_state_reg, state_reg, mul_res_reg);
    append_to_miram_command(ADD, mul_res_reg, new_state_reg2, state_reg);

    //Final write back to memory
    append_to_miram_write_reg_to_mem(state_adr_addr_ptr, state_reg, 0x01u);
}

void DO_LOOP(uint16_t datapoint, uint16_t datapoint2, uint16_t datapoint3, uint16_t
    datapoint4, void * beginning_of_weightsV, void* beginning_of_statesV, void *
    beginning_of_weightsV2, void * beginning_of_weightsV3, void * beginning_of_weightsV4)
{
    last_miram_1stHalf_addr_used = 0x0;
    last_miram_2ndHalf_addr_used = (MIRAM_1ST_HALF_LAST_INDEX);

    write_to_regfile(0, datapoint);
    write_to_regfile(1, datapoint2);
    write_to_regfile(2, datapoint3);
    write_to_regfile(3, datapoint4);

    uint32_t eval_reg = 0u;
    uint32_t eval_reg_2 = eval_reg + 1;
    uint32_t eval_reg_3 = eval_reg_2 + 1;
    uint32_t eval_reg_4 = eval_reg_3 + 1;

    uint32_t weight_reg = eval_reg_4 + 1;
    uint32_t weight_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr, beginning_of_weightsV,
        weight_reg, 0x01u);

    uint32_t weight_reg2 = weight_reg + 1;
    uint32_t weight_adr_addr_ptr2 = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr2, beginning_of_weightsV2,
        weight_reg2, 0x01u);

    uint32_t weight_reg3 = weight_reg2 + 1;
    uint32_t weight_adr_addr_ptr3 = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr3, beginning_of_weightsV3,
        weight_reg3, 0x01u);

    uint32_t weight_reg4 = weight_reg3 + 1;
    uint32_t weight_adr_addr_ptr4 = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(weight_adr_addr_ptr4, beginning_of_weightsV4,
        weight_reg4, 0x01u);

    uint32_t state_reg = weight_reg4 + 1;
    uint32_t state_adr_addr_ptr = ++last_miram_1stHalf_addr_used;
    append_to_miram_read_from_mem_to_reg(state_adr_addr_ptr, beginning_of_statesV, state_reg,
         0x00u);

    last_miram_2ndHalf_addr_used += 9u;
    uint32_t loopcfg = LOOP_CONFIG_INST(32u, last_miram_2ndHalf_addr_used,
        MIRAM_1ST_HALF_LAST_INDEX+0x01u);
    WriteLoopConfig(loopcfg);

    WFI();

}

void run_kernel_seneca()
{

    uint16_t datapoint;
    uint16_t datapoint2;
    uint16_t datapoint3;
    uint16_t datapoint4;

    KERNEL_SENECA();
    for (uint32_t  i = 0; i < NUM_COLS/4; i++)
        {

            datapoint = input[4*i];
            datapoint2 = input[(4*i)+1];
            datapoint3 = input[(4*i)+2];
            datapoint4 = input[(4*i)+3];

            beginning_of_weightsV = (void*)&weightV_g[((4*i)*NUM_ROWS)];
            beginning_of_weightsV2 = (void*)&weightV_g[(((4*i)+1)*NUM_ROWS)];
            beginning_of_weightsV3 = (void*)&weightV_g[(((4*i)+2)*NUM_ROWS)];
            beginning_of_weightsV4 = (void*)&weightV_g[(((4*i)+3)*NUM_ROWS)];
            beginning_of_statesV = (void*)&stateV_g[0];

            DO_LOOP(datapoint, datapoint2, datapoint3, datapoint4, beginning_of_weightsV,
                beginning_of_statesV, beginning_of_weightsV2, beginning_of_weightsV3,
                beginning_of_weightsV4);
        }
    for (bias_loop_counter = 0; bias_loop_counter < (NUM_ROWS/8); bias_loop_counter++)
    {
        beginning_of_statesV = (void*)&stateV_g[bias_loop_counter*8];
        KERNEL_SENECA_B();
        beginning_of_biasesV = (void*)&biasesV_g[bias_loop_counter*8];
        DO_LOOP_B(beginning_of_biasesV, beginning_of_statesV);
    }

    ATTR_INTR void ami_handler(void) {
    while(ami_irq_stat() != 0x00)
    {
            int i = 0;
            u union_float;
            uint32_t event = ami_msg_recv();
            union_float.i =event;

            if ((event == EVENT_EOS) || ((event >> 16 ) == 0xff)) {
                puts("Input Stream Finished");
            } else {
                benchmark_update(event);
                counter_input++;
            }
     }
}

}
```

6.5.2. Execution Time Measurements


shidqi98@uxapp10nl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/axon_test
File Edit View Search Terminal Help

```
Ready for RTL simulation
xcelium>
xcelium> run
Open failed on file "tb_log.ini". No such file or directory
Open failed on file "tb_log.ini". No such file or directory
Open failed on file "tb_log.ini". No such file or directory
Open failed on file "tb_log.ini". No such file or directory
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_bench
mark/riscv32-unknown-elf-bin/power_benchmark.sram" into memory (00200000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_bench
markL2/riscv32-unknown-elf-bin/power_benchmarkL2.sram" into memory (00400000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/power_bench
markOutput/riscv32-unknown-elf-bin/power_benchmarkOutput.sram" into memory (00600000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/projects/axon_test/r
iscv32-unknown-elf-bin/axon_test.sram" into memory (00800000)
2339.47 us      tb_log NCC_3 :  AXON test
2351.59 us      tb_log NCC_2 :  Neuron CoPro test, layer 3
2358.11 us      tb_log NCC_0 :  Neuron CoPro test, layer 1
2360.52 us      tb_log NCC_1 :  Neuron CoPro test, layer 2
2630.42 us      tb_log NCC_0 :  Input Stream Finished
2641.99 us      tb_log NCC_0 :  Running the loop!
3822.40 us      tb_log NCC_0 :  Loop done, begin sending!
3845.12 us      tb_log NCC_1 :  Input Stream Finished
3847.67 us      tb_log NCC_1 :  Running the loop!
4187.30 us      tb_log NCC_1 :  Loop done, begin sending!
4193.33 us      tb_log NCC_2 :  Input Stream Finished
4195.89 us      tb_log NCC_2 :  Running the loop!
4257.12 us      tb_log NCC_2 :  Done!
```

Figure 6.15: Output messages obtained from SENeCA while running the simulation with version 7.

By simulating the software of version 7 on SENeCA, we obtain the simulation screen displayed in Figure 6.15. We will also use the numbers on the simulation screen to calculate the execution times of parts of the program, as well as the total execution time. Similar to previous versions, we will find the time difference between the timestamps in Figure 6.15 to calculate the results. The results of the calculations are presented in Table 6.17. If we observe the grand total and compare it to the previous version, we can see that the optimization of the pipeline improved the performance by a significant amount, the difference being 793 μs. Compared to the execution time of version 6, that number is around 30%. The main two contributors are Loop 1, which benefits from the reduction in pipeline stalls, and Sending 1, which benefits greatly from the new way the AMI handler works.

Table 6.17: Execution time measurements for version 7. All times are in microseconds (μs).

Version 7					
Timestamps		Execution Time			
2358	Start	Loop 1	1192	Sending 1	272
2630	Input stream 1 finished	Loop 2	342	Sending 2	23
3822	Loop 1 finished	Loop 3	64	Sending 3	6
3845	Input stream 2 finished	Total	1598	Total	301
4187	Loop 2 finished	Grand total			1899
4193	Input stream 3 finished				
4257	Loop 3 finished				

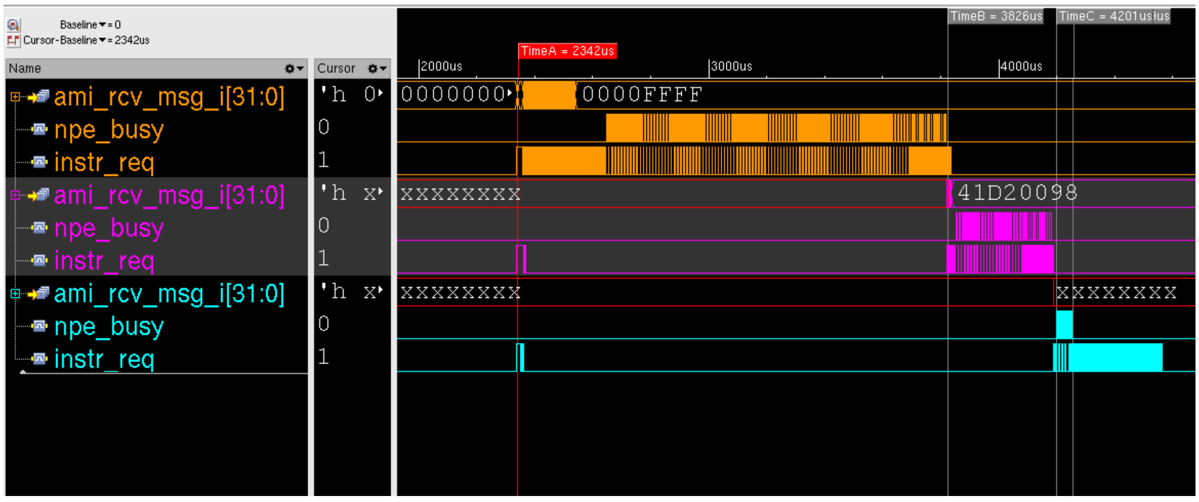


Figure 6.16: Waveforms of the signals produced by SENeCA when simulated using version 7 of the benchmark.

To verify the results obtained in Table 6.17, we will observe the waveforms of SENeCA running version 7 of the benchmark. A screenshot of the simulated waveforms is displayed in Figure 6.16. The cursors are placed approximately at the moments when one core finishes processing and the next core starts. The final cursor, TimeC, is placed approximately at the moment when core 3's NPEs finish processing, indicating the end of the program. Afterwards the RISC-V core of core 3 continues to run, but this is only to print out the results and is not considered part of the execution time. TimeC's timestamp is approximately similar to the final timestamp in Table 6.17 ("Loop 3 finished"). To obtain better insight into the optimizations of the pipeline, we will zoom in to have a closer look at the pipeline.

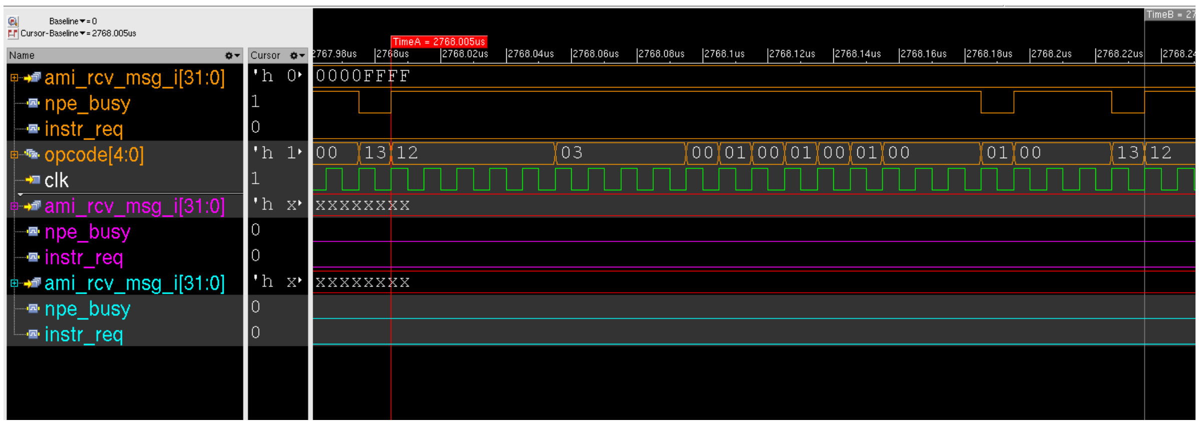


Figure 6.17: A more detailed look of the MM loop pipeline in version 7.

Figure 6.17 shows the MM loop in detail, including the instructions performed by the NPEs. The instructions performed by the NPEs are labeled "opcode", and the waveform is colored turquoise for clarity. The clock, meanwhile, is colored orange, located directly below the opcode waveform. The "Baseline" cursor is placed at the start of the MM loop. Directly after the cursor, the opcode being executed is coded "0x12", standing for an MLD operation. In this version, the MLD commands occur directly one after another without any stalls, owing to a fix in the SENeCA RTL. After 5 MLD operations, the 4 MUL operations (code 0x03) are performed. As expected, since they do not have any dependencies, they are executed without pipeline stalls between them. The next set of instructions, 4 ADD instructions (code 0x01), are executed with the expected number of pipeline stalls. Finally, a single MST (code 0x13) instruction is performed to store the result back to the memory. In total, the entire MM loop takes 23 clock cycles to complete. Compared to 42 clock cycles of the MM loop of version 6, this is a massive improvement.

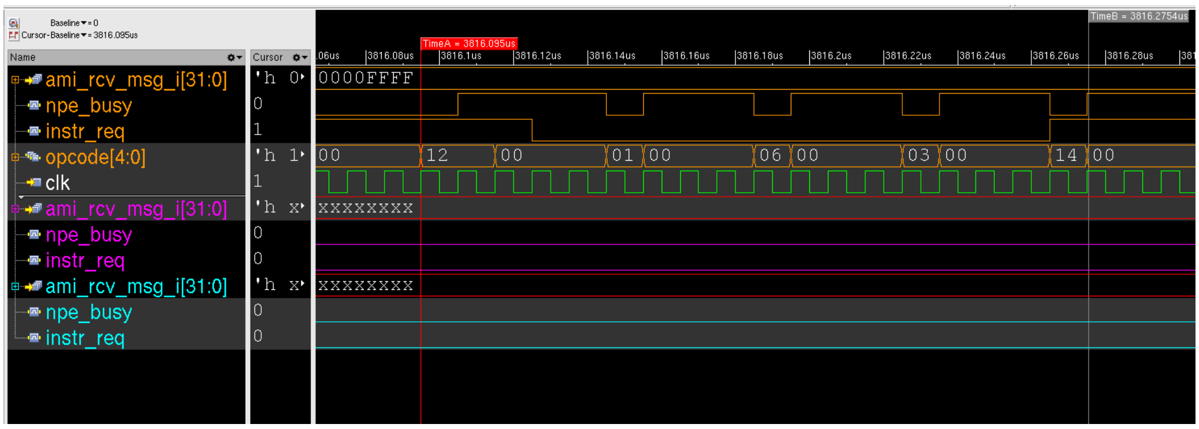


Figure 6.18: A more detailed look of the bias and activation function loop pipeline in version 7.

For comparison, we will also look at the bias and activation loop, displayed in Figure 6.18. In total this entire loop takes 18 clock cyles to run. Table 4.4 shows the instructions involved in this loop, as this part remains relatively unchanged from version 2. Since one loop only processes one neuron, there are no instructions parallelisms available to exploit. Therefore, we can observe in Figure 6.18 that the only instructions that can be run back-to-back are the first 2 MLD operations (code 0x12). The subsequent instructions (ADD (0x01), GTH (0x06), MUL (0x03), and EVC (0x14)) all depend on the instruction directly preceding them, thus all of them stall for 3 clock cycles.

6.5.3. Power Measurements

In this section, the power measurements of SENeCA running version 7 of the program will be performed.

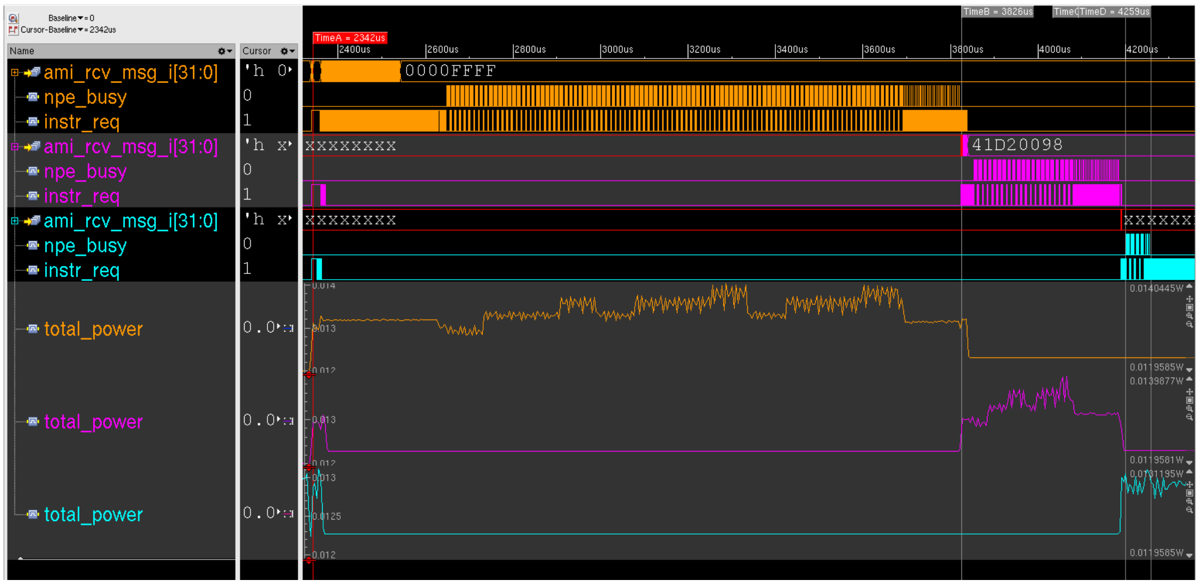


Figure 6.19: Time-based power graph of version 7 together with several signals.

Figure 6.19 shows the results of the time-based power analysis performed on version 7. All of the graphs (signal waveforms and power graphs) are color-coded to the core they belong to. Orange stands for core 1, turquoise stands for core 2, while purple stands for core 3. Similar to previous versions, all of the cores exhibit similar behavior in their consumption of power. This version's power consumption is similar to version 6, with the cores consuming around 0.012 W when idle, and a maximum of 0.0135 when active. To obtain more precise numbers that represent the power consumption, average mode power analysis will be performed on the three active time periods of each core.

Table 6.18: Average power consumption of version 7, during the active time of core 1 (Active 1). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_0	25.00%	2.80E-03	5.71E-03	4.86E-04	9.00E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.19: Average power consumption of version 7, during the active time of core 2 (Active 2). All power values are in Watts.

Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_1	25.00%	2.80E-03	5.71E-03	4.86E-04	9.00E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_2	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

Table 6.20: Average power consumption of version 7, during the active time of core 3 (Active 3). All power values are in Watts.

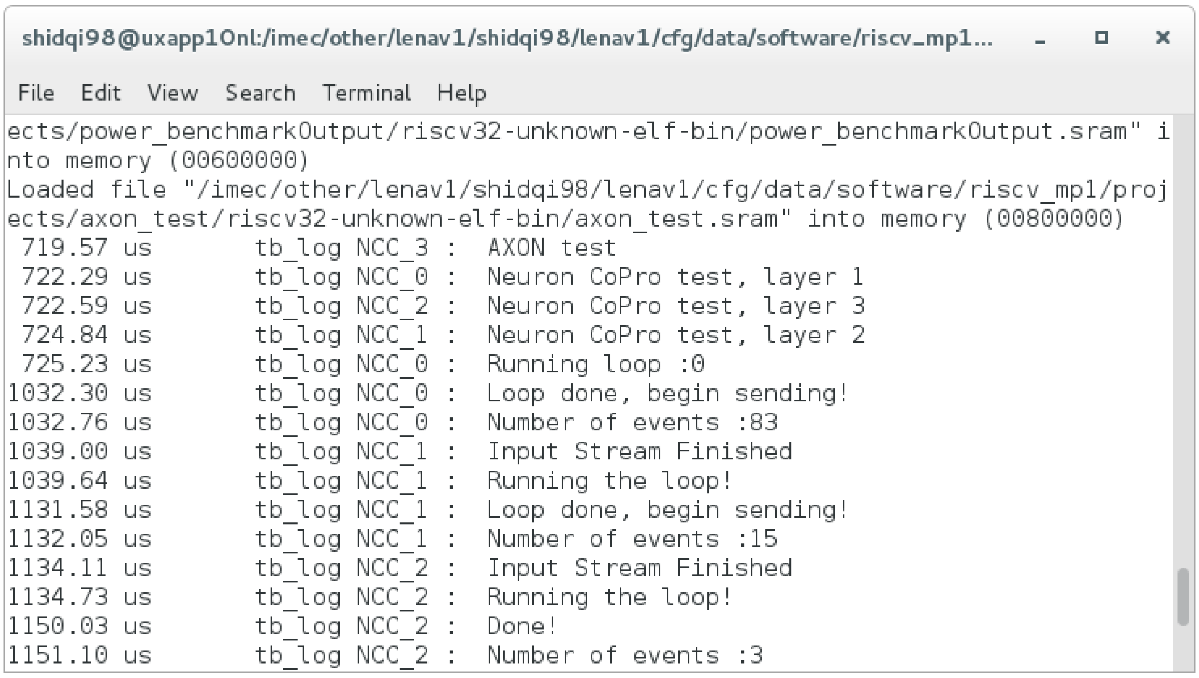
Instance	Pct_cells	Leakage	Internal	Switching	Total
u_seneca_ncc_2	25.00%	2.80E-03	5.71E-03	4.86E-04	9.00E-03
u_seneca_ncc_0	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03
u_seneca_ncc_1	25.00%	2.80E-03	5.11E-03	1.22E-04	8.03E-03

The results of the average power analysis are shown in Tables 6.18, 6.19, and 6.20. The results here are not vastly different from version 6, although the Table 6.18 does show a slight increase in the power consumption of core 1, the active core. This is presumably due to the reduction in the pipeline stalls. While pipeline stalls still consume power, they do so at a lower rate than other operations, and other operations being packed together more densely will cause an increase in power consumption. By using the numbers in the tables above to calculate the total energy consumed with Equation 5.1, we obtain the number of 0.000047 J, or 0.047 mJ.

6.6. Ideal Power Calculation

In this section, we will calculate the ideal energy consumption for a single inference using SENeCA. Earlier, in Chapter 5, we have calculated the energy consumptions of basic NPE operations executed individually. Since the benchmarking program can be broken down to basic NPE operations, it is possible to calculate the theoretical minimum energy required to do one inference, by calculating the types and number of operations required, and multiplying that by the energy consumptions of each operation. Then, we will compare the results from the actual run of the benchmarking program to the theoretical minimum.

Since version 7 is the most energy efficient and the one that has the shortest execution time, we will calculate the number of required operations by referring to the algorithm of version 7. However, it should be noted that the energy consumption of an iteration depends slightly on the input data, since the input data determines the number of events that the EVC captures, for example. Therefore, we will use the first frame of the first audio recording as the input vector, since all of the tests run thus far have used that input vector. To know the number of events that are triggered when processing the input vector, we run version 7 again, and we print the number of events per layer. The simulation screen can be seen in Figure 6.20. This run is done with a 500MHz clock to save time, hence the different timestamps at which the program terminates. We know from this figure that the number of events for each layer are 83, 15, and 3, respectively.


shidqi98@uxapp10nl:/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1...
File Edit View Search Terminal Help

```
ects/power_benchmarkOutput/riscv32-unknown-elf-bin/power_benchmarkOutput.sram" i
nto memory (00600000)
Loaded file "/imec/other/lenav1/shidqi98/lenav1/cfg/data/software/riscv_mp1/proj
ects/axon_test/riscv32-unknown-elf-bin/axon_test.sram" into memory (00800000)
 719.57 us       tb_log NCC_3 :  AXON test
 722.29 us       tb_log NCC_0 :  Neuron CoPro test, layer 1
 722.59 us       tb_log NCC_2 :  Neuron CoPro test, layer 3
 724.84 us       tb_log NCC_1 :  Neuron CoPro test, layer 2
 725.23 us       tb_log NCC_0 :  Running loop :0
1032.30 us       tb_log NCC_0 :  Loop done, begin sending!
1032.76 us       tb_log NCC_0 :  Number of events :83
1039.00 us       tb_log NCC_1 :  Input Stream Finished
1039.64 us       tb_log NCC_1 :  Running the loop!
1131.58 us       tb_log NCC_1 :  Loop done, begin sending!
1132.05 us       tb_log NCC_1 :  Number of events :15
1134.11 us       tb_log NCC_2 :  Input Stream Finished
1134.73 us       tb_log NCC_2 :  Running the loop!
1150.03 us       tb_log NCC_2 :  Done!
1151.10 us       tb_log NCC_2 :  Number of events :3
```

Figure 6.20: Simulation run of version 7, showing the number of events per layer.

To calculate the ideal energy, first we must break down the benchmarking program to its most basic operations. The main components are the NPE loops, of which there are 3 types: the matrix multiplication loop, bias and activation loop with event triggering (denoted T in the table) and bias and activation loop with the event not triggering (denoted NT in the table). The energy consumed to execute one of these loops is calculated by adding invididual operations that make up the loop. The calculation results for the 3 loop types are shown in Table 6.21. The energy values are in nanojoules (nJ). As a side note, the energy consumption for individual NPE operations are taken from Table 5.13.

Table 6.21: Ideal energy consumption of the 3 types of loops found in the benchmarking program. All values are in nanojoules.

MM Loop			Bias and Activation Loop (NT)			Bias and Activation Loop (T)		
OpCode	Number	Energy	OpCode	Number	Energy	OpCode	Number	Energy
MLD	5	0.187	MLD	2	0.075	MLD	2	0.075
MUL	4	0.074	ADD	1	0.019	ADD	1	0.019
ADD	4	0.075	GEQ	1	0.018	GEQ	1	0.018
NOP	9	0.157	MUL	1	0.019	MUL	1	0.019
MST	1	0.043	EVC	1	0.018	EVC	1	0.019
Total		0.536	Total		0.148	Total		0.149

After obtaining the energy consumption values for the 3 types of loops available, we need to calculate how many of these loops are executed in a single inference. Also, we need to consider the other operations required to run the program, namely the communcations between the cores. To make this easier, we will consider the number of operations per layer. Table 6.22 shows the number of loops per layer of each type, and the number of events sent and received by each layer. The number of operations are then accumulated based on the type, and the results are shown under the ”Total” column. The numbers in this column are then multiplied by the energy consumption values of the loops (from Table 6.21) and events (from Table 5.13). All of the energy values shown in this table are in nanojoules (nJ).

Table 6.22: Ideal number of operations and energy consumption (in nJ) of the benchmarking program, shown per layer.

Type	Layer 1	Layer 2	Layer 3	Total	Energy
MM loop	3104	672	32	3808	2.0409
BA loop (T)	11	2	1	14	0.0021
BA loop (NT)	21	30	1	52	0.0077
Event received	390	83	15	488	0.0086
Event sent	83	15	0	98	0.0017
Total					2.06

The final value of 2060 nJ, or 2.06 μJ, is the ideal energy consumption of SENeCA running version 7 of the benchmarking program.

6.7. Accuracy Comparison

In this section we will analyze the accuracy of the SENeCA implementation of the benchmarking program compared to the baseline described in Chapter 4. As noted before, the implementations on SENeCA (from version 2 onwards) all use the bf16 number format. Therefore, we will measure the impact of the precision loss of this decision on accuracy. As mentioned, there are 192 matrices that represent the transformed audio recordings. These audio recordings are of varying lengths. To measure the accuracy, we decided to pick 10 of these recordings as samples, and feed them as input to version 7 of the benchmarking program run on SENeCA. This was done due to time limitations, as SENeCA did not exist yet as a physical chip during the time of writing, and simulations generally run much slower than a physical chip. Since the experiments until now had all been with a single frame as input, we need to modify the source code to allow multi-input processing. This was done by altering the testbench source file so that the testbench feeds multiple vectors in succession as input.

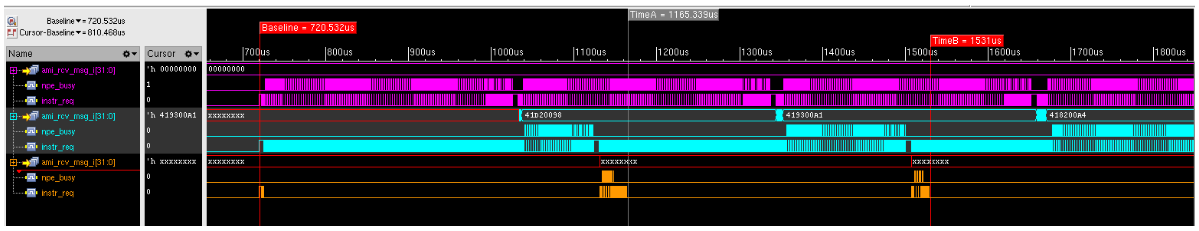


Figure 6.21: Multi-input run waveforms of version 7.

Figure 6.23 shows the waveforms of the simulation running version 7 of the benchmarking program with multiple inputs. As before, the names and traces are color-coded to their source cores, purple for core 1, turquoise for core 2, and orange for core 3. The cursors marks when the program starts (Baseline), when input 1 is finished (TimeA), and when input 2 is finished (TimeB). The process then continues (not shown in the figure).

As mentioned previously in Chapter 4, the output of the neural network (a 29-element vector) represents the possibility of a sound/letter being uttered during the 10ms of recording that the input vector represents. Each element of the output vector represents the probability of one sound being uttered, and the element with the highest probability is chosen for every frame. This results in a sequence of sounds/letters, and from this sequence it is determined if the target phrase (in this dataset, ”Aloha”) is in the character sequence. The conversion of output vectors to character sequences is done by an external program run on a PC. Therefore, to check the accuracy, we can check the output character sequences of version 7 run on SENeCA and compare them to the outputs of the baseline PC version developed in Python. Alternatively, to get a more accurate picture of the accuracy loss, we can also compare the output vectors of the PC version and SENeCA and see the differences.

While the study on which this thesis is based on proved that Loihi can be more accurate than the PC version [14], we did not have access to it during the time of writing and therefore only had the PC version as comparison.

Table 6.23: Accuracy comparisons of the baseline model implemented on a PC with with version 7 on SENeCA.

No	Phrase	Predicted (PC)	Predicted (SENeCA)	Accuracy loss	Error rate
1	”Aloha”	Aloha	Aloha	0%	3.71%
2	”All the while”	aohaf	aohaf	0%	5.20%
3	”Aloha”	Aloha	Aloha	0%	1.20%
4	”Take a load off”	tae lohad f	tae lohad	9%	8.35%
5	”Aloha”	Aloha	Aloha	0%	1.61%
6	”Hello”	lo	lo	0%	9.13%
7	”Aloha”	Aloha	Aloha	0%	2.98%
8	”Metal alloy”	taohaloy	taohaloy	0%	1.52%
9	”Aloha”	Aloha	Aloha	0%	4.93%
10	”How are you”	h oare e e uyu	h oare e e uyu	0%	7.13%
			Average	0.90%	4.58%

Table 6.23 summarizes the comparisons performed on the outputs from the PC and from SENeCA. 10 audio recordings were used in total, with varying lengths. The phrases that are uttered in these recordings are listed in the ”Phrase” column. To the right of that column, the predictions of the PC version and SENeCA are listed. As stated before, the main function of the program is to determine whether or not ”Aloha” is uttered in the recording. In this context, both versions produce the exact same results, with no loss in accuracy by the SENeCA version (phrases 1,3,5,7,9 are predicted as including ”Aloha”, while the others are predicted to not include it). However, they do exhibit differences when the predicted phrases themselves are compared. More specifically, phrase No. 4 (”Take a load off”) is predicted differently by the two versions, with the SENeCA version lacking a final ”f”. The other phrases are predicted identically by both versions.

Finally, we also compared the output vectors of the two versions. Since they use different number formats, we wanted to see how much this affects the results, and we do this by the following method. Taking the first frame from the first recording as an example, this frame is fed into the two programs, generating two output vectors. The greatest element of each output vector is chosen, and the difference between the two numbers is calculated. That difference is then divided by the greatest element of the output vector from the PC version. We take this as the error rate. By doing this for all of the frames in phrase 1, and taking the average of all error rates, we obtain the number 3.71 %. We then repeat this for all of the phrases, and the results are listed in the ”Error rate” column in Table 6.23.

6.8. Summary of Results and Discussion

Table 6.24 gives the summary of the optimizations implemented in the different software versions used in the experiments in this chapter, and 6.25 gives the summary of the results, including the results of the same benchmark performed on Loihi from [14]. The third and fourth columns show the total average power and the corresponding energy-to-solution values calculated from the total average power values. The fifth and sixth columns show the average dynamic power and the corresponding energy-to-solution values. The average dynamic power is calculated by adding the internal and switching power values shown in the average power tables in this chapter.

For versions 2 to 7, a few trends can be observed. First, the power consumption of one version is generally slightly higher than the previous version. That is to be expected, since each version introduces an optimization that either uses a new component or tries to compress the operations time-wise. This naturally results in a higer power consumption. Howeve, the execution time decreases for every version, showing that the optimizations introduced are effective. Compared to the implementation on

Loihi obtained from [14], we can see that the implementation on SENeCA is faster and more energy efficient, with comparable accuracy. One thing that should be noted is that the technology nodes used by SENeCA and Loihi are different (GF 22 nm vs Intel 14 nm), so the comparisons below might be different if the same technology node is used.

Table 6.24: Summary of the optimization implemented in each software version.

Software	Optimization
Version 2	Basic usage of the NPE (neural processors)
Version 3	Loop unrolling using 2 elements
Version 4	Loop unrolling using 4 elements
Version 5	Utilization of the Event Capture Unit
Version 6	Utilization of the Loop Buffer
Version 7	Pipeline optimization of the MM loop

Table 6.25: Summary of the power and energy of all versions and Loihi from [14]

Software	Time (us)	Power (T) (mW)	Energy (T) (mJ)	Power (D) (mW)	Energy (D) (mJ)
Version 2	22253	24.6414	0.548	17.3703	0.387
Version 3	17778	24.6458	0.438	17.5203	0.311
Version 4	15113	24.8418	0.375	17.9583	0.271
Version 5	12973	24.9458	0.324	18.2703	0.237
Version 6	2692	24.9558	0.067	18.0003	0.048
Version 7	1899	25.0558	0.048	18.6003	0.035
Loihi	3378	110.00	0.37	81.00	0.27

One reason why SENeCA performs better than Loihi for this specific DNN application is because Loihi implements rate coding to send events between neurons. In a dense layer of a DNN, if a neuron fires, it sends a numerical value to neurons of the next layer. With rate coding, a neuron sends multiple signals to the destination neuron instead. These signals do not have values, since it is their timing of the transmission that will be interpreted by the destination neuron as the value. Therefore, multiple signals have to be sent every time a neuron is fired. on the other hand, SENeCA uses data packets that is sent by a neuron when it fires. These data packets contain values, so one transmission is enough. This is more efficient with regards to energy. Other than that, the use of an x86 processor [22] in the Loihi chip might also cause it to consume more power than needed, while SENeCA uses a RISC-V Ibex core [5] that is more suited for low power or embedded applications.

As mentioned in Chapter 3, the main design principle of SENeCA is flexibility and efficiency so that it can adapt to multiple types of neural network architectures [109]. Loihi, on the other hand, seems to be more suited to Spiking Neural Network architectures due to its built-in communication protocols between cores using rate coding, while in this study, the use of the DNN forces it to send events using multiple spikes. Therefore, the results might be different if an SNN-based application is used as the benchmark. Furthermore, since Loihi includes an on-chip learning accelerator, a benchmark that includes the learning process (as opposed to this inference-only application) may also highlight its strength more.

To gain a better insight into the power consumption of the individual components that make up a SENeCA NCC core, a table detailing the power consumption of the components running version 7 of the benchmarking software is presented in Appendix A, since the table is quite large. Nevertheless, here we shall describe the main points of interest that can be inferred from that table.

Figure 6.22 shows a graph of the power consumption of a single SENeCA core (NCC) during its active time when running version 7 of the benchmarking software. Since the power consumption of the NCP (containing the NPEs and data memory) greatly exceeds that of the other components, a logarithmic scale is used. From the graph, we can see that the NCP, containing the majority of the memory cells dominate the power consumption. The power consumption of the AMI and the Ibex core was reduced with the use of power gating, which is also applied to several components of the NCP (with the exception of the data memory). For components that cannot be power-gated, such as the SRAM blocks of both the instruction memory and the data memory, clock gating was used. ”Others” here refers to the power consumption the mux/arbiter of the NCC, as well as other components that were not relevant such as the SMPU, RV timer, etc.

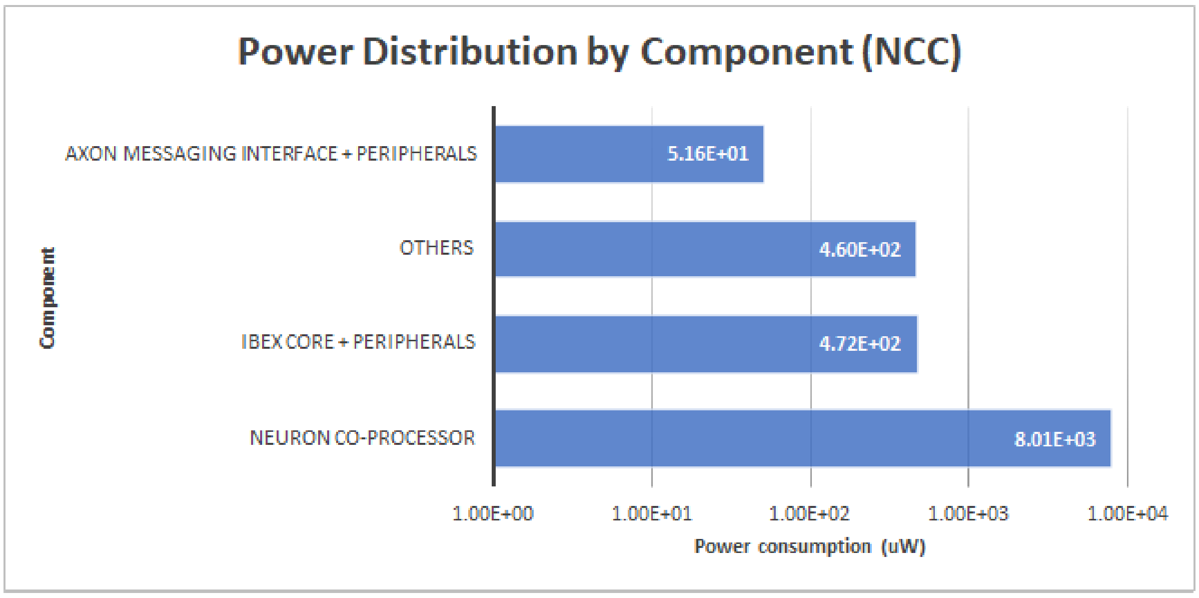


Figure 6.22: Breakdown of power consumption by component for a single core (NCC).

Figure 6.23 shows a logarithmic graph of the power consumption of a the NCP component depicted in the bottom bar in Figure 6.22, broken down further to show the the NCP’s components. Again, as the power consumption of the individual components vary significantly, we use a logarithmic scale in this graph as well. As noted previously, the data memory is by far the most power-hungry component. Note, however, that since the components of SENeCA are largely parametrizable, the power consumption of different instances of SENeCA may vary. Here, 8 NPEs and 64 memory cells are used, as previously stated in Chapter 5. For reference, the power consumption of a single memory cell is 62.1 μW, while the power consumption of a single NPE is 80.3 μW, obtained by dividing their power consumption depicted in Figure 6.23 by 64 and 8, respectively. The ”others” bar refers to the minor components whose power consumption is negligible to the other components.

Observing both Figure 6.23 and Table 6.25, we can see that the accelerators incorporated in SENeCA such as the loop buffer (included in the pipeline controller) and the EVC have significant impact on performance while consuming little power. For example, the use of the EVC to take advantage of the sparsity of the events costs only around 60 μW of extra power per core, but the execution time dropped by around 20% (the change of the execution time from version 4 to version 5). Likewise, the use of the loop buffer to reduce context switches between the NPEs and the Ibex core also yielded significant results, around 80% (the change of the execution time from version 5 to version 6) while consuming only around 10 μW.

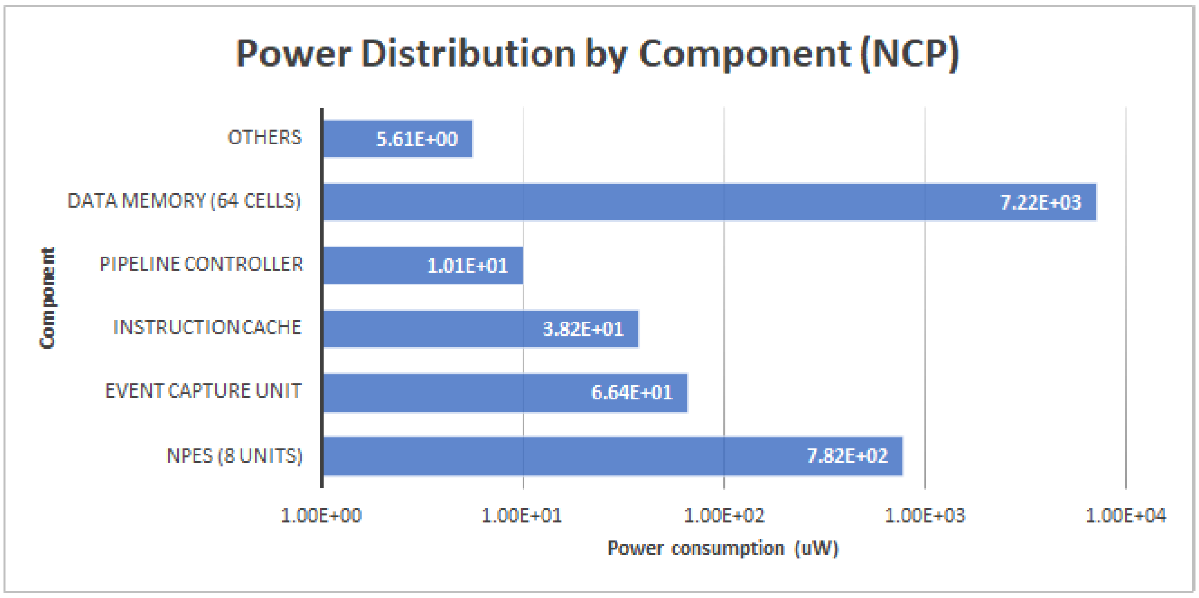


Figure 6.23: Breakdown of power consumption by component for a single NCP.

On the other hand, the power consumption of the memory block is still very high. Observing the table in Appendix A, the percentage of the power consumption wasted as leakage power is quite high, at around 30%. While these numbers are still within the order of magnitude of the leakage power stated by the datasheet of the memory blocks provided by Global Foundries (not disclosed here for confidentiality reasons), optimizations with regard to the power consumption of the memory is desirable. One possible thing is to implement clock gating to individual memory cells instead of the entire block. Also, since the sizes of layers in the neural network used here vary (also applies to the neural networks used in other benchmarking studies, such as [17] and [92]), a core responsible for running a smaller layer does not need as many memory cells active as other cores. Therefore,implementing power gating to unused memory blocks to reduce power consumption might be possible, while still having identical physical memory blocks for each core to maintain flexibility.

7 Conclusion

7.1. Overview

In this thesis, our main goal is to perform benchmarking on SENeCA [109], a new neuromorphic processor design by IMEC The Netherlands. By conducting a literature study of the published research regarding both neuromorphic processor designs and the benchmarking of these designs presented in Chapter 2, we have identified several avenues in which improvement could be made, as follows:

- Since SENeCA is a new design, there are no benchmarking studies performed on it yet.
- Most benchmarks only have workloads running neural network architectures based on actual real-world problems. While this is not an issue by itself, a workload to measure the individual synaptic operations in conjunction with another real-world workload could provide more insight to the performance.
- No benchmarking study thus far reports the power/energy consumption of individual components or parts of a neuromorphic processor, only the power consumption of the entire chip.

To build upon the existing research, we adapted the keyword spotting program used previously as a benchmark in [14] for use on SENeCA instead of developing one from scratch. The program takes in a transformed audio recording as a vector, and infers if a certain keyword phrase is spoken in the original audio recording. It uses a DNN architecture with three layers with two hidden layers. The reasons why this program in particular was chosen, as explained in more detail in Chapter 4, include the potential use of the application in situations that might benefit from lower power consumption provided by neuromorphic architectures. Also, the fact that SENeCA is still in development at the time of writing precluded the use of more complex network architectures such as those used in [17], or integration in frameworks such as SNABSuite [76] and Nengo [11].

The Python source code for the benchmarking program was converted to C and adapted for use in SENeCA, while preserving the architecture. The numbers used were also converted to the BF16 format to conform to SENeCA's requirements. As explained in Chapter 5, two initial versions were developed, one using only the conventional processor contained in SENeCA's cores, and another using the neural processors in an unoptimized fashion. A simple testbench that instantiates a 4-core SENeCA processor was also developed. Using Cadence© XCelium as the simulation program, the two initial versions of the benchmarking program were run on SENeCA, using an instance of SENeCA described in Chapter 5. After obtaining the simulation waveforms, the performance of these two versions were measured. Finally, using Cadence© Joules, the simulation waveforms of both versions were analyzed to obtain an estimation of the power consumption during the run. The power estimations include graphs that show the power consumption of SENeCA's components over time, as well as their average power consumption over the entire run. The power consumption numbers include the leakage, dynamic, as well as the total power. Using these results the energy-to-solution numbers were calculated for both of the initial versions. The results for the two initial versions are shown in Chapter 5.

Apart from the main workload described above, a secondary workload to test the power and energy consumption of individual synaptic operations is also used for benchmarking. This workload has the NPE processors execute a single instruction repeatedly for a time, followed by a loop of another instruction, until all of the possible instructions, listed in Chapter 3 are included. The same analysis as the one done for the two initial versions of the main workload as described above is performed, obtaining the power and energy consumption of each possible instruction. The results are broken down to show the power/energy consumption of individual components making up the SENeCA core, and is presented in Section 5.6.

The results mentioned above indicate that SENeCA's performance was not enough to process the workload in real-time (by having a processing time of less than 10 ms), and the energy-to-solution was significantly higher than Loihi [22], the neuromorphic processor that is used as comparison. To fully realize SENeCA's potential, optimizations were made on the software that uses the components and accelerators present in SENeCA to improve the performance and energy efficiency. Among these are the Event Capture Unit, Loop Buffer, as well as the register blocks present in the neural processors as described in Chapter 3. The optimizations were done in five iterations to show the impact of the use of these components individually. Together with the first two initial versions described above, in total there are seven versions presented. Chapter 6 describes the optimization method and the C implementation in detail, as well as the performance measurements and power consumption analysis for each iteration, similar to what we did for the first two iterations.

The results of all iterations are presented in Section 6.8, including the results of the benchmarking results of Loihi published in [14]. While the numbers for both the energy-to-solution as well as the time-to-solution metrics are worse than Loihi for the earlier versions, the later versions perform comparably to Loihi. Starting from version 5, the energy-to-solution numbers become lower than Loihi, while from version 6, SENeCA outperforms Loihi in terms of both performance and energy consumption. By comparing the performance of SENeCA running the final optimized version of the benchmarking software, we can see that the time-to-solution is 56% that of Loihi, while using only 12% of the energy required by Loihi. Finally, an accuracy test was performed where the outputs of SENeCA are compared to the outputs of the baseline PC implementation. We found that the accuracy loss is around 0.9%. Overall, we can argue that the benchmarking software achieves its objectives; to measure the performance of SENeCA while running a neural network workload that provides insight into its capabilities and limitations for both its developers and future researchers.

7.2. Discussion and Future Work

This section describes topics or ideas that can be further investigated. During the process of writing the thesis, we had some interesting ideas that we were unfortunately not yet able to realize. As such, they may be a useful topic for research in the future.

- Try more extensive applications as a benchmark. In [20], Davies presented a list of suitable candidate applications for a neuromorphic benchmarking suite. Furthermore, in established benchmarks for other fields such as SPEC [91], a suite consists of multiple benchmarks representing different problems. While the neural network implemented in this thesis is one of possible use case of a neuromorphic processor, more extensive and complicated applications using different architectures (such as a CNN as done by Ceolini, et al. [17]) is needed to allow us to draw a better conclusion. Since one of the goals of SENeCA is to be a flexible architecture that is capable of running different architectures efficiently, performing benchmarking with only a single program is not enough.
- Perform benchmarking for the training process as well. Here, the program used as a baseline for the benchmarking software came with pre-trained weights, so we could only perform benchmarking on the inference process. Performing it on the learning process would certainly be useful, as the comparisons between neuromorphic chips can be more extensive. For example, Loihi is

equipped with an on-chip learning accelerator[22], so a benchmarking program that consists only of inference cannot fully show Loihi's strength.

- Implement this methodology on other platforms. As mentioned previously, we did not have access to other platforms than SENeCA, so we had to use data obtained from another study to perform the comparison. Performing this benchmark on other platforms will make it more general, and thus potentially usable.
- Define a more general tool for benchmarking, or integrate SENeCA to other existing frameworks such as SNABSuite [76] or Nengo [11]. While the benchmarking process here did provide insight into the performance of SENeCA, a more general benchmarking suite that employs this methodology would make it more efficient to benchmark multiple platforms.
- For SENeCA, since the component that consumed the most energy was shown to be the data memory, this could be a good target for optimizations. For example, the use of magnetoresistive random access memory (MRAM) [30] could be an option. Alternatively, using clock gating per memory cell instead of for the entire block, or using power gating to turn off the unused memory cells in cores running smaller layers is also a possibility.

References

[1] Abderazek Ben Abdallah and Khanh N Dang. *Neuromorphic computing principles and organization*. Springer Nature, 2022.

[2] Jayesh Bapu Ahire. *The Artificial Neural Networks Handbook: Part 1*. Aug. 2018. URL: `https://www.datasciencecentral.com/the-artificial-neural-networks-handbook-part-1/`.

[3] Filipp Akopyan et al. “Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip”. In: *IEEE transactions on computer-aided design of integrated circuits and systems* 34.10 (2015), pp. 1537–1557.

[4] Ronny Krashinsky et al. “NVIDIA Ampere Architecture In-Depth”. In: (2020). URL: `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`.

[5] PULP Team et al. *lowRISC Ibex*. `https://github.com/lowRISC/ibex`. 2022.

[6] Rami A Alzahrani and Alice C Parker. “Neuromorphic circuits with neural modulation enhancing the information content of neural signaling”. In: *International Conference on Neuromorphic Systems 2020*. 2020, pp. 1–8.

[7] Arnon Amir et al. “A Low Power, Fully Event-Based Gesture Recognition System”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 7388–7397. DOI: `10.1109/CVPR.2017.781`.

[8] Albert-László Barabási. “Network science”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 371.1987 (2013), p. 20120375.

[9] Denilson Barbosa, Ioana Manolescu, and Jeffrey yu Xu. *Microbenchmark*. 2009.

[10] Trevor Bekolay, Terrence C Stewart, and Chris Eliasmith. “Benchmarking neuromorphic systems with Nengo”. In: *Frontiers in Neuroscience* (2015), p. 380.

[11] Trevor Bekolay et al. “Nengo: a Python tool for building large-scale functional brain models”. In: *Frontiers in neuroinformatics* 7 (2014), p. 48.

[12] Ben Varkey Benjamin et al. “Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations”. In: *Proceedings of the IEEE* 102.5 (2014), pp. 699–716.

[13] Drishti Beohar and Akhtar Rasool. “Handwritten digit recognition of MNIST dataset using deep learning state-of-the-art artificial neural network (ANN) and Convolutional Neural Network (CNN)”. In: *2021 International Conference on Emerging Smart Computing and Informatics (ESCI)*. IEEE. 2021, pp. 542–548.

[14] Peter Blouw et al. “Benchmarking keyword spotting efficiency on neuromorphic hardware”. In: *Proceedings of the 7th annual neuro-inspired computational elements workshop*. 2019, pp. 1–8.

[15] Sumon Kumar Bose, Jyotibdha Acharya, and Arindam Basu. “Is my Neural Network Neuromorphic? Taxonomy, Recent Trends and Future Directions in Neuromorphic Engineering”. In: *2019 53rd Asilomar Conference on Signals, Systems, and Computers*. 2019, pp. 1522–1527. DOI: `10.1109/IEEECONF44664.2019.9048891`.

[16] Andrew S Cassidy and Andreas G Andreou. “Beyond Amdahl’s law: An objective function that links multiprocessor performance gains to delay and energy”. In: *IEEE Transactions on Computers* 61.8 (2011), pp. 1110–1126.

[17] Enea Ceolini et al. “Hand-gesture recognition based on EMG and event-based camera sensor fusion: A benchmark in neuromorphic computing”. In: *Frontiers in Neuroscience* 14 (2020), p. 637.

[18] Gal Chechik, Isaac Meilijson, and Eytan Ruppin. “Synaptic pruning in development: a computational account”. In: *Neural computation* 10.7 (1998), pp. 1759–1777.

[19] Federico Corradi and Giacomo Indiveri. “A neuromorphic event-based neural recording system for smart brain-machine-interfaces”. In: *IEEE transactions on biomedical circuits and systems* 9.5 (2015), pp. 699–709.

[20] Mike Davies. “Benchmarks for progress in neuromorphic computing”. In: *Nature Machine Intelligence* 1.9 (2019), pp. 386–388.

[21] Mike Davies et al. “Advancing Neuromorphic Computing With Loihi: A Survey of Results and Outlook”. In: *Proceedings of the IEEE* 109.5 (2021), pp. 911–934. DOI: `10.1109/JPROC.2021.3067593`.

[22] Mike Davies et al. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: `10.1109/MM.2018.112130359`.

[23] Michael V. DeBole et al. “TrueNorth: Accelerating From Zero to 64 Million Neurons in 10 Years”. In: *Computer* 52.5 (2019), pp. 20–29. DOI: `10.1109/MC.2019.2903009`.

[24] Stephen R Deiss et al. “A pulse-coded communications infrastructure for neuromorphic systems”. In: *Pulsed neural networks* (1999), pp. 157–178.

[25] Tobi Delbruck and Shih-Chii Liu. “Data-driven neuromorphic DRAM-based CNN and RNN accelerators”. In: *2019 53rd Asilomar Conference on Signals, Systems, and Computers*. IEEE. 2019, pp. 500–506.

[26] Lei Deng et al. “Tianjic: A Unified and Scalable Chip Bridging Spike-Based and Continuous Neural Computation”. In: *IEEE Journal of Solid-State Circuits* 55.8 (2020), pp. 2228–2246. DOI: `10.1109/JSSC.2020.2970709`.

[27] Travis DeWolf, Pawel Jaworski, and Chris Eliasmith. “Nengo and low-power AI hardware for robust, embedded neurorobotics”. In: *Frontiers in Neurorobotics* 14 (2020), p. 568359.

[28] Rodney Douglas, Misha Mahowald, and Carver Mead. “Neuromorphic analogue VLSI”. In: *Annual review of neuroscience* 18 (1995), pp. 255–281.

[29] Egm4313.s12. *Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals*. File: `Neuron3.png`. 2018. URL: `https://commons.wikimedia.org/wiki/File:Neuron3.png`.

[30] B.N. Engel et al. “A 4-Mb toggle MRAM based on a novel bit and switching method”. In: *IEEE Transactions on Magnetics* 41.1 (2005), pp. 132–136. DOI: `10.1109/TMAG.2004.840847`.

[31] *Enterprise artificial intelligence market revenue worldwide 2016-2025*. en. `https://www.statista.com/statistics/607612/worldwide-artificial-intelligence-for-enterprise-applications/`. Accessed: 2022-9-3.

[32] E. Paxon Frady et al. *Neuromorphic Nearest-Neighbor Search Using Intel’s Pohoiki Springs*. 2020. DOI: `10.48550/ARXIV.2004.12691`. URL: `https://arxiv.org/abs/2004.12691`.

[33] Charlotte Frenkel, Jean-Didier Legat, and David Bol. “MorphIC: A 65-nm 738k-Synapse/mm^2 Quad-Core Binary-Weight Digital Neuromorphic Processor With Stochastic Spike-Driven Online Learning”. In: *IEEE Transactions on Biomedical Circuits and Systems* 13.5 (2019), pp. 999–1010. DOI: `10.1109/TBCAS.2019.2928793`.

[34] Charlotte Frenkel et al. “A 0.086-mm ^212.7-pJ/SOP 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm CMOS”. In: *IEEE transactions on biomedical circuits and systems* 13.1 (2018), pp. 145–158.

[35] Steve B. Furber et al. “The SpiNNaker Project”. In: *Proceedings of the IEEE* 102.5 (2014), pp. 652–665. DOI: `10.1109/JPROC.2014.2304638`.

[36] Todor Ganchev, Nikos Fakotakis, and George Kokkinakis. “Comparative evaluation of various MFCC implementations on the speaker verification task”. In: *Proceedings of the SPECOM*. Vol. 1. 2005. 2005, pp. 191–194.

[37] Ankit Garg et al. “Dynamics of soil water content using field monitoring and AI: A case study of a vegetated soil in an urban environment in China”. In: *Sustainable Computing: Informatics and Systems* 28 (2020), p. 100301. ISSN: 2210-5379. DOI: `https://doi.org/10.1016/j.suscom.2019.01.003`. URL: `https://www.sciencedirect.com/science/article/pii/S221053791830235X`.

[38] *Gartner forecasts worldwide Artificial Intelligence Software Market to reach $62 billion in 2022*. URL: `https://www.gartner.com/en/newsroom/press-releases/2021-11-22-gartner-forecasts-worldwide-artificial-intelligence-software-market-to-reach-62-billion-in-2022`.

[39] Marc-Oliver Gewaltig and Markus Diesmann. "Nest (neural simulation tool)". In: *Scholarpedia* 2.4 (2007), p. 1430.

[40] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[41] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[42] K. Hara and K. Nakayamma. "Comparison of activation functions in multilayer neural network for pattern classification". In: *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*. Vol. 5. 1994, 2997–3002 vol.5. DOI: `10.1109/ICNN.1994.374710`.

[43] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[44] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[45] Geoffrey Hinton and Terrence J Sejnowski. *Unsupervised learning: foundations of neural computation*. MIT press, 1999.

[46] Sebastian Höppner et al. "The SpiNNaker 2 processing element architecture for hybrid digital neuromorphic computing". In: *arXiv preprint arXiv:2103.08392* (2021).

[47] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[48] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". In: IEEE, Feb. 2014, pp. 10–14. ISBN: 978-1-4799-0920-9. DOI: `10.1109/ISSCC.2014.6757323`.

[49] Xiaohe Huang et al. "In-memory computing to break the memory wall". In: *Chinese Physics B* 29.7 (2020), p. 078504.

[50] Giacomo Indiveri and Shih-Chii Liu. "Memory and information processing in neuromorphic systems". In: *Proceedings of the IEEE* 103.8 (2015), pp. 1379–1397.

[51] Dmitry Ivanov et al. "Neuromorphic Artificial Intelligence Systems". In: *arXiv preprint arXiv:2205.13037* (2022).

[52] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[53] Nick Jakobi. "Minimal simulations for evolutionary robotics". PhD thesis. University of Sussex, 1998.

[54] Nick Jakobi, Phil Husbands, and Inman Harvey. "Noise and the reality gap: The use of simulation in evolutionary robotics". In: *European Conference on Artificial Life*. Springer. 1995, pp. 704–720.

[55] Norman Jouppi et al. "A domain-specific architecture for deep neural networks". In: *Communications of the ACM* 61 (Aug. 2018), pp. 50–59. DOI: `10.1145/3154484`.

[56] Norman Jouppi et al. "Motivation for and Evaluation of the First Tensor Processing Unit". In: *IEEE Micro* 38.3 (2018), pp. 10–19. DOI: `10.1109/MM.2018.032271057`.

[57] Norman P. Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 1–12. DOI: `10.1145/3079856.3080246`.

[58] Dhiraj Kalamkar et al. "A study of BFLOAT16 for deep learning training". In: *arXiv preprint arXiv:1905.12322* (2019).

[59] Kumar Kapil et al. "Importance of String Matching in Real World Problems". In: 3 (July 2014), pp. 2319–7242.

[60] Aechan Kim, Mohyun Park, and Dong Hoon Lee. “AI-IDS: Application of Deep Learning to Real-Time Web Intrusion Detection”. In: *IEEE Access* 8 (2020), pp. 70245–70261. DOI: 10.1109/ACCESS.2020.2986882.

[61] Jin Wook Kim, Eunsang Kim, and Kunsoo Park. “Fast Matching Method for DNA Sequences”. In: *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Ed. by Bo Chen, Mike Paterson, and Guochuan Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 271–281. ISBN: 978-3-540-74450-4.

[62] Urs Köster et al. “Flexpoint: An adaptive numerical format for efficient training of deep neural networks”. In: *Advances in neural information processing systems* 30 (2017).

[63] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[64] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. “A 128 x 128 120db 30mw asynchronous vision sensor that responds to relative intensity change”. In: *2006 IEEE International Solid State Circuits Conference-Digest of Technical Papers*. IEEE. 2006, pp. 2060–2069.

[65] Mike Loukides. *AI adoption in the enterprise 2021*. Apr. 2021. URL: https://www.oreilly.com/radar/ai-adoption-in-the-enterprise-2021/.

[66] Matthew Thomas Martinez. “An Overview of Google’s Machine Intelligence Software TensorFlow.” In: (2016).

[67] Peter Mattson et al. “Mlperf training benchmark”. In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 336–349.

[68] Christian Mayr, Sebastian Hoeppner, and Steve Furber. “Spinnaker 2: A 10 million core processor system for brain simulation and machine learning”. In: *arXiv preprint arXiv:1911.02385* (2019).

[69] Vittorio Mazzia et al. “Real-Time Apple Detection System Using Embedded Systems With Hardware Accelerators: An Edge AI Application”. In: *IEEE Access* 8 (2020), pp. 9102–9114. DOI: 10.1109/ACCESS.2020.2964608.

[70] Carver Mead. “Neuromorphic Electronic Systems”. In: *Proceedings of the IEEE* 78 (10 1990), pp. 1629–1636. ISSN: 15582256. DOI: 10.1109/5.58356.

[71] Paul A. Merolla et al. “A million spiking-neuron integrated circuit with a scalable communication network and interface”. In: *Science* 345.6197 (Aug. 2014), pp. 668–673. DOI: 10.1126/science.1254642.

[72] Orlando Moreira et al. “NeuronFlow: a neuromorphic processor architecture for live AI applications”. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 840–845.

[73] John Nolte. “The human brain”. In: *An Introduction to Its Functional Anatomy.* (2002).

[74] Kyoung Su Oh and Keechul Jung. “GPU implementation of neural networks”. In: *Pattern Recognition* 37 (6 June 2004), pp. 1311–1314. ISSN: 0031-3203. DOI: 10.1016/J.PATCOG.2004.01.013.

[75] Garrick Orchard et al. “Efficient neuromorphic signal processing with loihi 2”. In: *2021 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2021, pp. 254–259.

[76] Christoph Ostrau et al. “Benchmarking and Characterization of event-based Neuromorphic Hardware”. In: 2019.

[77] Christoph Ostrau et al. “Benchmarking of neuromorphic hardware systems”. In: *Proceedings of the Neuro-inspired Computational Elements Workshop*. 2020, pp. 1–4.

[78] Eustace Painkras et al. “SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation”. In: *IEEE Journal of Solid-State Circuits* 48.8 (2013), pp. 1943–1953. DOI: 10.1109/JSSC.2013.2259038.

[79] Eustace Painkras et al. “SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation”. In: *IEEE Journal of Solid-State Circuits* 48.8 (2013), pp. 1943–1953.

[80] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. ISSN. Elsevier Science, 2008. ISBN: 9780080922812. URL: `https://books.google.de/books?id=3b63x-0P3%5C_UC`.

[81] Jing Pei et al. “Towards artificial general intelligence with hybrid Tianjic chip architecture”. In: *Nature* 572.7767 (2019), pp. 106–111.

[82] Mihai A Petrovici et al. “Characterization and compensation of network-level anomalies in mixed-signal neuromorphic modeling platforms”. In: *PloS one* 9.10 (2014), e108590.

[83] Thomas Pfeil et al. “Six networks on a universal neuromorphic computing substrate”. In: *Frontiers in neuroscience* 7 (2013), p. 11.

[84] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[85] Marcel Salathé, Thomas Wiegand, and Markus Wenzel. “Focus group on artificial intelligence for health”. In: *arXiv preprint arXiv:1809.04797* (2018).

[86] Catherine D Schuman et al. “A survey of neuromorphic computing and neural networks in hardware”. In: *arXiv preprint arXiv:1705.06963* (2017).

[87] Biswa Sengupta, Simon Barry Laughlin, and Jeremy Edward Niven. “Consequences of converting graded to action potentials upon neural information coding and energy efficiency”. In: *PLoS computational biology* 10.1 (2014), e1003439.

[88] Shy Shoham, Daniel H O’Connor, and Ronen Segev. “How silent is the brain: is there a “dark matter” problem in neuroscience?” In: *Journal of Comparative Physiology A* 192.8 (2006), pp. 777–784.

[89] Sumit B Shrestha and Garrick Orchard. “Slayer: Spike layer error reassignment in time”. In: *Advances in neural information processing systems* 31 (2018).

[90] Fedor Shvetsov, Anton Konushin, and Anna Sokolova. “Neural Network Model for Face Recognition from Dynamic Vision Sensor”. In: *Proceedings of the 30th International Conference on Computer Graphics and Machine Vision (GraphiCon 2020). Part 2* (Dec. 2020), short17–1. DOI: `10.51130/graphicon-2020-2-4-17`.

[91] *Standard Performance Evaluation Corporation*. URL: `https://www.spec.org/benchmarks.html`.

[92] Terrence C Stewart et al. “Closed-loop neuromorphic benchmarks”. In: *Frontiers in neuroscience* 9 (2015), p. 464.

[93] Andreas Stöckel et al. “Binary associative memories as a benchmark for spiking neuromorphic hardware”. In: *Frontiers in computational neuroscience* 11 (2017), p. 71.

[94] John E Stone, David Gohara, and Guochun Shi. “OpenCL: A parallel programming standard for heterogeneous computing systems”. In: *Computing in science & engineering* 12.3 (2010), p. 66.

[95] Jan Stuijt et al. “μBrain: An event-driven and fully synthesizable architecture for spiking neural networks”. In: *Frontiers in neuroscience* 15 (2021), p. 538.

[96] Fati Tahiru. “AI in education: A systematic literature review”. In: *Journal of Cases on Information Technology (JCIT)* 23.1 (2021), pp. 1–20.

[97] Monideepa Tarafdar, Cynthia M Beath, and Jeanne W Ross. “Using AI to enhance business operations”. In: *MIT Sloan Management Review* 60.4 (2019), pp. 37–44.

[98] Ryan Taylor and Xiaoming Li. “A micro-benchmark suite for AMD GPUs”. In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE. 2010, pp. 387–396.

[99] Gianvito Urgese et al. “Benchmarking a many-core neuromorphic platform with an MPI-based dna sequence matching algorithm”. In: *Electronics* 8.11 (2019), p. 1342.

[100] Sacha J Van Albada et al. “Performance comparison of the digital neuromorphic hardware SpiNNaker and the neural network simulation software NEST for a full-scale cortical microcircuit model”. In: *Frontiers in neuroscience* 12 (2018), p. 291.

[101] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. “The NumPy array: a structure for efficient numerical computation”. In: *Computing in science & engineering* 13.2 (2011), pp. 22–30.

[102] Reinhold P Weicker. “Dhrystone: a synthetic systems programming benchmark”. In: *Communications of the ACM* 27.10 (1984), pp. 1013–1030.

[103] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1498765.1498785.

[104] John Wuu et al. “3D V-Cache: the Implementation of a Hybrid-Bonded 64MB Stacked Cache for a 7nm x86-64 CPU”. In: *2022 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 65. 2022, pp. 428–429. DOI: 10.1109/ISSCC42614.2022.9731565.

[105] Cheng-Xin Xue et al. “16.1 a 22nm 4mb 8b-precision reram computing-in-memory macro with 11.91 to 195.7 tops/w for tiny ai edge devices”. In: *2021 IEEE International Solid-State Circuits Conference (ISSCC)*. Vol. 64. IEEE. 2021, pp. 245–247.

[106] Joseph Yacim and Douw Boshoff. “Impact of Artificial Neural Networks Training Algorithms on Accurate Prediction of Property Values”. In: *Journal of Real Estate Research* 40 (Nov. 2018), pp. 375–418. DOI: 10.1080/10835547.2018.12091505.

[107] Yexin Yan et al. “Low-Power Low-Latency Keyword Spotting and Adaptive Control with a SpiNNaker 2 Prototype and Comparison with Loihi”. In: *arXiv preprint arXiv:2009.08921* (2020).

[108] Amirreza Yousefzadeh et al. “Multiplexing AER asynchronous channels over LVDS links with flow-control and clock-correction for scalable neuromorphic systems”. In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2017, pp. 1–4.

[109] Amirreza Yousefzadeh et al. “SENeCA: Scalable Energy-efficient Neuromorphic Computer Architecture”. In: ().

A

Breakdown of Power Consumption by Component

This appendix contains the table of the average power consumption of Core 1 during its active time running version 7 of the benchmarking software. Components that are high in the hierarchy are color coded, with red for the NCC (Core), yellow for the components that are directly below that, and beige for the components directly below the yellow ones. All power consumption values are shown in Watts (W).

Instance	Cells	Pct_cells	Leakage	Internal	Switching	Total	Lvl
NCC	121083	25.00%	2.80E-03	5.72E-03	4.86E-04	9.00E-03	1
NCP	70966	14.66%	2.78E-03	4.95E-03	2.76E-04	8.01E-03	2
NPEs + DMEM	70442	14.55%	2.78E-03	4.94E-03	2.74E-04	8.00E-03	3
Data Memory	27548	5.69%	2.78E-03	4.32E-03	1.25E-04	7.22E-03	4
DMEM1	185	0.04%	4.33E-05	2.10E-03	3.57E-06	2.15E-03	5
DMEM2	180	0.04%	4.33E-05	2.36E-05	1.43E-06	6.84E-05	5
DMEM3	180	0.04%	4.33E-05	2.36E-05	1.42E-06	6.84E-05	5
DMEM4	188	0.04%	4.33E-05	2.26E-05	1.59E-06	6.75E-05	5
DMEM5	180	0.04%	4.33E-05	2.26E-05	1.58E-06	6.75E-05	5
DMEM6	180	0.04%	4.33E-05	1.77E-05	1.35E-06	6.24E-05	5
DMEM7	180	0.04%	4.33E-05	1.76E-05	1.32E-06	6.22E-05	5
DMEM8	188	0.04%	4.33E-05	1.75E-05	1.30E-06	6.22E-05	5
DMEM9	180	0.04%	4.33E-05	1.75E-05	1.28E-06	6.22E-05	5
DMEM10	188	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM11	180	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM12	180	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM13	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM14	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM15	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM16	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM17	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM18	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM19	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM20	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM21	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM22	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM23	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM24	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM25	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5

Instance	Cells	Pct_cells	Leakage	Internal	Switching	Total	Lvl
DMEM26	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM27	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM28	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM29	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM30	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM31	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM32	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM33	188	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM34	180	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM35	180	0.04%	4.33E-05	1.75E-05	1.27E-06	6.21E-05	5
DMEM36	180	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM37	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM38	180	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM39	180	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM40	180	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM41	180	0.04%	4.33E-05	1.75E-05	1.26E-06	6.21E-05	5
DMEM42	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM43	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM44	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM45	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM46	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM47	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM48	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM49	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM50	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM51	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM52	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM53	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM54	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM55	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM56	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM57	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM58	180	0.04%	4.33E-05	1.75E-05	1.25E-06	6.21E-05	5
DMEM59	180	0.04%	4.33E-05	1.75E-05	1.24E-06	6.21E-05	5
DMEM60	180	0.04%	4.33E-05	1.75E-05	1.24E-06	6.21E-05	5
DMEM61	180	0.04%	4.33E-05	1.75E-05	1.24E-06	6.21E-05	5
DMEM62	180	0.04%	4.33E-05	1.75E-05	1.24E-06	6.21E-05	5
DMEM63	180	0.04%	4.33E-05	1.75E-05	1.23E-06	6.21E-05	5
DMEM64	180	0.04%	4.33E-05	1.75E-05	1.23E-06	6.21E-05	5
NPEs	42880	8.86%	7.13E-06	6.26E-04	1.49E-04	7.82E-04	4
NPE1	4285	0.88%	6.61E-07	6.53E-05	1.46E-05	8.05E-05	5
NPE2	4285	0.88%	6.61E-07	6.52E-05	1.46E-05	8.05E-05	5
NPE3	4285	0.88%	6.61E-07	6.53E-05	1.45E-05	8.05E-05	5
NPE4	4285	0.88%	6.61E-07	6.52E-05	1.46E-05	8.04E-05	5
NPE5	4285	0.88%	6.60E-07	6.52E-05	1.45E-05	8.04E-05	5
NPE6	4285	0.88%	6.60E-07	6.52E-05	1.45E-05	8.04E-05	5
NPE7	4285	0.88%	6.61E-07	6.52E-05	1.45E-05	8.03E-05	5
NPE8	4285	0.88%	6.61E-07	6.52E-05	1.45E-05	8.03E-05	5
Event Capture Unit	6172	1.27%	1.38E-06	6.48E-05	1.44E-07	6.64E-05	5
Instruction Cache	2198	0.45%	4.15E-07	3.05E-05	7.28E-06	3.82E-05	5

Instance	Cells	Pct_cells	Leakage	Internal	Switching	Total	Lvl
Pipeline Controller	142	0.03%	2.35E-08	8.10E-06	1.98E-06	1.01E-05	5
DMEM TLuL Adapter	290	0.06%	7.42E-08	3.00E-06	1.59E-06	4.67E-06	3
TLuL Adapter Registers	66	0.01%	1.48E-08	8.32E-07	8.80E-08	9.35E-07	3
Ibex Core + Peripherals	32601	6.73%	5.21E-06	2.78E-04	1.89E-04	4.72E-04	2
Ibex Core	32521	6.72%	5.19E-06	2.77E-04	1.87E-04	4.70E-04	3
Ibex: Registers	4740	0.98%	9.63E-07	1.25E-04	2.22E-05	1.48E-04	4
Ibex: EX Block	5729	1.18%	9.16E-07	4.36E-05	7.23E-05	1.17E-04	4
Ibex: PMP	14230	2.94%	1.61E-06	2.14E-05	3.35E-05	5.65E-05	4
Ibex: IF Block	1399	0.29%	2.85E-07	3.56E-05	1.67E-05	5.26E-05	4
Ibex: CSR Registers	4082	0.84%	9.08E-07	2.35E-05	1.73E-05	4.18E-05	4
Ibex: ID Block	1620	0.33%	3.49E-07	1.26E-05	1.64E-05	2.93E-05	4
Ibex: WB Block	123	0.03%	4.43E-08	9.76E-06	4.61E-06	1.44E-05	4
Ibex: LS Block	525	0.11%	8.61E-08	4.61E-06	2.44E-06	7.14E-06	4
Ibex: Instruction FIFO	39	0.01%	1.37E-08	9.69E-07	9.66E-07	1.95E-06	3
Ibex: Data FIFO	33	0.01%	4.22E-09	3.38E-08	1.07E-07	1.45E-07	3
Ibex: TLuL Adapter	7	0.00%	6.30E-10	5.91E-09	2.64E-08	3.29E-08	3
Instruction Memory	6	0.00%	3.61E-06	3.81E-04	5.14E-07	3.85E-04	2
AMI + Peripherals	9750	2.01%	2.27E-06	4.92E-05	1.63E-07	5.16E-05	2
AMI	9659	1.99%	2.25E-06	4.84E-05	1.16E-07	5.08E-05	3
AMI Receive Buffer	7524	1.55%	1.74E-06	3.43E-05	6.81E-08	3.62E-05	4
AMI Send Buffer	2019	0.42%	4.79E-07	1.38E-05	5.54E-09	1.43E-05	4
AMI TLuL Adapter	88	0.02%	2.17E-08	7.59E-07	4.56E-08	8.26E-07	3
Multiplexer/Arbiter	4414	0.91%	6.36E-07	1.69E-05	1.13E-05	2.89E-05	2
IMEM TLuL Adapter	454	0.09%	1.04E-07	1.58E-05	9.13E-06	2.51E-05	2
External Memory DMA	1623	0.34%	3.46E-07	1.42E-05	7.42E-07	1.53E-05	2
RV Timer	1240	0.26%	1.83E-07	6.22E-06	0.00E+00	6.40E-06	2
Debugger	7	0.00%	6.47E-10	0.00E+00	0.00E+00	6.47E-10	2