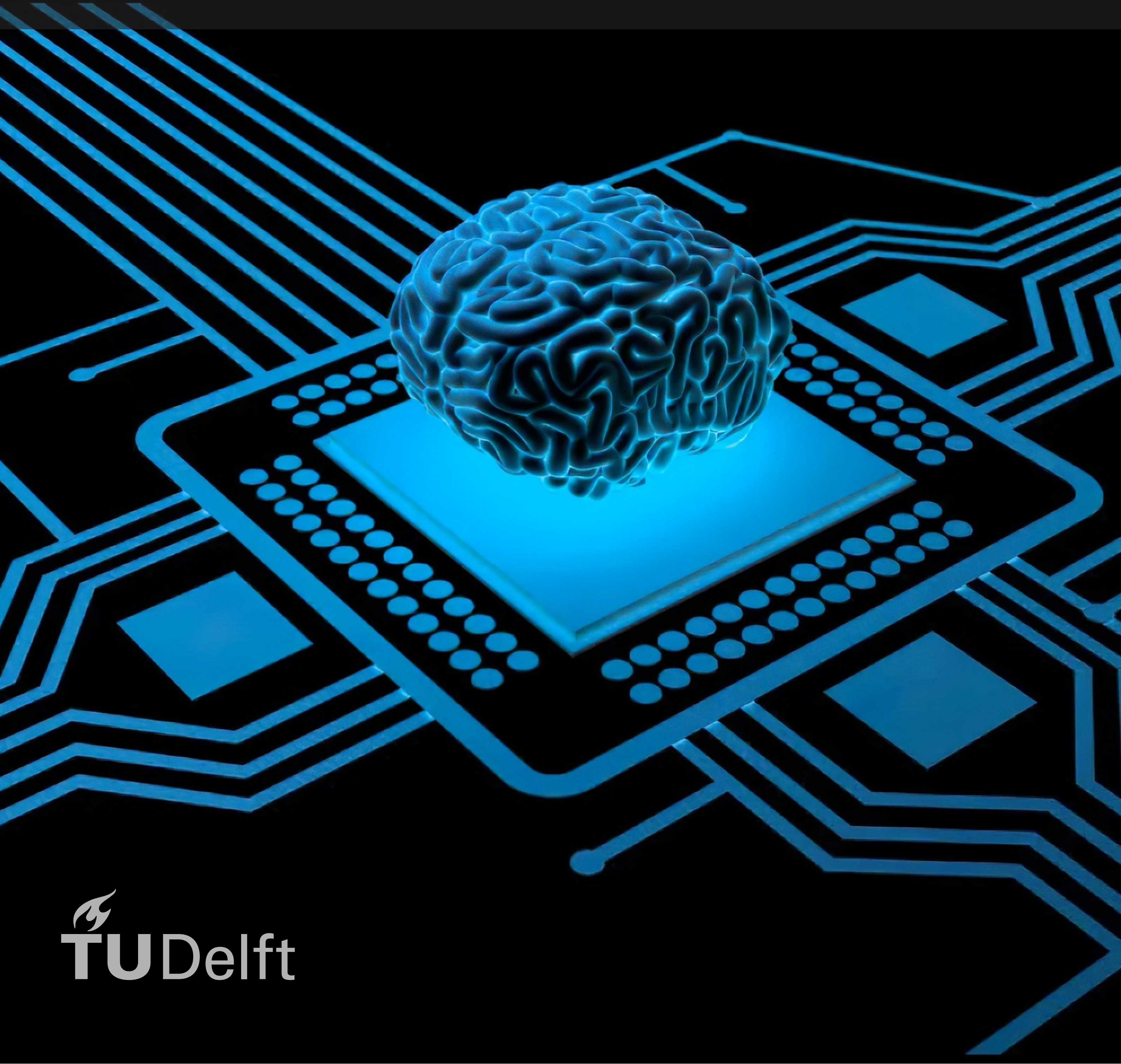
# Efficient mapping of large-scale SNN and rate-based DNN on SENeCA

Prithvish Vijaykumar Nembhani



# Efficient mapping of large-scale SNN and rate-based DNN on SENeCA

by

Prithvish Vijaykumar Nembhani

towards the partial fulfillment of Master of Science in Computer Engineering

at the Delft University of Technology,

to be defended publicly on February 28, 2023 at 2:00 PM.

| | | |
|---|---|---|
| Student number: | 5352576 | |
| Project duration: | December 1, 2021 – Saturday 25th February, 2023 | |
| Thesis committee: | Dr. ir. Zaid Al-Ars, | TU Delft, supervisor |
| | Dr. Przemysław Pawełczak, | TU Delft |
| | Dr. ir. Amirreza Yousefzadeh, | IMEC |
| | Ir. Gert Jan Van Schaik, | IMEC |

TU Delft

# Abstract

Artificial intelligence, machine learning, and deep learning have been the buzzwords in almost every industry (medical, automotive, defense, security, finance, etc.) for the last decade. As the market moves towards AI-based solutions, so does the computation need for these solutions increase and change with time. With the rise of smart cities and cyberphysical systems, the need for edge devices and efficient computation on the edge increases. While most of these newly developed deep learning models are quite large and wasteful in terms of energy, there have been recent methods that help improve the performance on the edge. However, due to their size, variety, and irregularity, the computing and power requirements are often too large to deploy these models on edge devices. This prohibits the application of such models within a rich field of application that requires high-throughput and real-time execution.

SENeCA[1] [2] is a next-generation RISC-V-based neuromorphic computing architecture that was designed primarily for ultralow-edge applications where adaptivity is required. To mathematically model SENeCA, SENSIM (Scalable Energy Efficient Simulator, an open source simulator developed by the Interuniversity Microelectronic Center) provides an accurate mathematical software model of SENeCA, which helps in the early development and realization of a spiking neural network and deep neural network. This thesis work develops an efficient mapping tool SENMap (Scalable Energy-Efficient Neuromorphic Computing Architecture Mapper) on top of SENSIM which maps spiking neural networks efficiently. Having a faster, scalable realization software solution that can cater to large-scale neural networks can speed up the development procedure.

SENMap is developed in such a way that it supports flexible SNN/DNN application replacement, multiple single- and multi-objective optimization algorithms; the flexibility to choose from different optimization strategies; and also varying architectural parameters at the time of experimentation. Results show that mapping and neural processing elements (NPEs) depend primarily on the rate at which the sensor processes the data. On the basis of the rate, an early realization of SNN- and DNN-based edge AI chips SENMap. Depending on the actual parameters used, the maximum achieved improvements in energy consumption was around 40%.

***Keywords:*** *Spiking neural network, Deep neural network, Rate-based deep neural network, SNN mapping, DNN mapping, SENeCA, EdgeAI, single and multi objective optimization, SNN inference, DNN inference. SENSIM*

# Acknowledgements

SENMap (Scalable Energy-efficient Neuromorphic Computer Architecture Mapper) marks the successful completion of my Master's degree at the Delft University of Technology. Masters and research work can be challenging, intimidating, and full of failure, but with persistence, determination, and a little bit of support from family, friends, and excellent mentors anything is possible. First, I would like to thank my mother, father, and sister for their immense support, from wishing me a good morning to sending me songs and checking on me. Masters would have just been a dream had there not been support from especially my dad (Vijaykumar) and my mom (Payal), my sister (Lavina) for giving way more than what they could to help me reach where I wanted to. I would like to thank my grandfather (Prof. Murlidhar Raisinghani), Dr. Dilip Nandwani and Varsha Nandawani, for extending their hand for help multiple times when I was in need.

I want to extend my gratitude to all professors at TU Delft (Prof. Zaid Al-Ars, Prof. Sorin Cotofana, Prof. Koen Langendoen, Prof. Rene van Leuken, Prof. David Tax, Prof. Georgias, Prof. Julian Kooij, Prof. DiCarlo, Prof. Rajendra Bishnoi, Prof. Mottaqiallah, Prof. Fabio Sebastiano, Prof. Jan van Gemert, Prof. Said Hamdioui) for delivering such great content and making education available to us even in COVID times. I wish I could attend more lectures. I especially would like to thank Prof. Zaid Al-Ars for always motivating me, helping me with TU Delft logistics, correcting and completing my incomplete theories, and at times being critical to improve my work by reviewing it multiple times. I would like to thank Rob, Richard, Bart, and Danyao Wang for collaborating on learning methods of bench-marking and modeling hardware with machine learning with proprietary hardware. Would like also like to thank Dr. John Sahaya Rani Alex, Dr. Venkatasubramanian Krishnamoorthy, and Kamlesh Sharma for writing me recommendation letters.

I also would like to thank the researchers and team (Dr.ir. Amirreza Yousefzadeh, Ir. Gert-Jan, Dr. Paul Detterer, Dr. Manolis Sifalakis, Dr. Mario Konijnenburg, and Marsha Loomans for helping me with the development of the Project. I especially would like to thank Amirreza for answering my queries very patiently, right from Python debugging, to convince me how the brain is an asynchronous system; Gert-Jan for teaching me good standards for programming; Paul for helping me set up conda on IMEC servers. Finally, I would like to thank IMEC for lending me the internal servers to build SENMap.

Staying abroad can be lonely, but having friends beyond your family to check on you is a blessing. First, I would like to thank my embedded boys (Rohan Katkam, Sourav Mohapatra, Vivian Dsouza, and Nishad Mandalik) for always being there for me in the tough times. I would like to thank Shraddha Agrawal, who supported me and motivated me to pursue my Master's, and helped me right from my Bachelor's at VIT to my time in Bangalore. I am forever grateful to have friends like Ritika Adlakha who supported me by always cheering me up with her motivational words like "Rise Prithvish Rise"; Akshita Paliwal for checking on me; Lalitha Prasunna for all the discussions on MBA assignments; Suyi Wen for all the study and coffee breaks; Alessia and her family for hosting me in Italy; Ethereal Matter team (Susanna, Alessia, Alejandro, Stephy, Alessia, Scott, Orson ) for all the fun, learing and great collaboration; Leigh boys (Srijit Sen, Rohan Kaushik, Rohan katkam, Divaay Mehta, Raghav Shankar, Prateek Jain, Rahul Ramesh) for supporting me when I did not have a place to stay in the Netherlands; Folks of the 13th (Akshat Sharma, Antariksh Narain, Prithvijit Chakraborty, Athishay Kiran, Shreyas Rao, Sarvesh, Sai Pranav, Rishabh Agrawal, Niranj Jyotish) for late-night online games in COVID; Villagers of Bengland (Sourav Mohapatra, Vivian Dsouza, Rohan katkam, Nishad Mandlik, Meenu Sarah Mathai, Kanhai Joshi, Avanika, Renuka, Allan, Adila and Pratyush) for card games and fun filled dinner nights; Sachin for always teaching me humbleness; Sagar Ahuja, Vijay Negi, Jay, Mitu, Sneha, Aniket, Himanshu and the complete team at Arista Networks for making my transition towards my masters easy, Prajwala and her family for helping me make the transition to Netherlands in COVID times; Kevin and Alexandra for laughing at my jokes on our way to Eindhoven; Henri J. Norden and Rijk for teaching how to make life easy with automation; Ahmed Aouchi and Milos Grubur for great collaboration on re-configurable computing project, and Tan Xin for the machine learning and deep learning projects; Ikenna and Floris for the great collaboration on Processor design Project.

Kudos to the complete batch of 2020 who did not back off from all the obstacles. Keep learning, and keep growing!!

*Prithvish Vijaykumar Nembhani*
*Delft, February 2023*

# Contents

# Abbreviations

| | |
|---|---|
| **AGE-MOEAD** | Adaptive Geometry Estimation based MOEA |
| **AI** | Artificial Intelligence |
| **AMI** | Axon Messaging Interface |
| **BRKGE** | Biased Random Key Genetic Algorithm |
| **C-TAEA** | Two-Archive Evolutionary Algorithm for Constrained Multiobjective Optimization |
| **CBInfer** | Change-Based Inference for Convolutional Neural Networks on Video Data |
| **CMAES** | Covariance matrix adaptation evolution strategy |
| **DE** | Differential Evolution |
| **DNN** | Deep Neural Networks |
| **GE** | Genetic Algorithm |
| **GUI** | Graphical User Interface |
| **IMEC** | Interuniversity Microelectronics Centre |
| **IPU** | Intelligent Processing Units |
| **ISRES** | Improved Stochastic Ranking Evolutionary Strategy |
| **MOEAD** | Multi-objective evolutionary algorithm based on decomposition |
| **NCC** | Neuron Compute Cluster |
| **NCP** | Neural Co-Processor |
| **NoC** | Network on Chip |
| **NPE** | Neural Processing Element |
| **NSGA-II** | Non-dominated Sorting Genetic Algorithm 2 |
| **NSGA-III** | Non-dominated Sorting Genetic Algorithm |
| **PSO** | Particle Swarm Optimization |
| **PyGMO** | Parallel Global Multi-objective Optimizer |
| **Pymoo** | Python Multi Objective Optimizaiton |
| **R-NSGA-II** | Reference point based Non-dominated Sorting Genetic Algorithm 3 |
| **SDK** | Software Development Kit |
| **SENeCA** | Scalable Energy-efficient Neuromorphic Computer Architecture |
| **SENSIM** | SENeCA Simulator |
| **SIMD** | Single Instruction Multiple Data |
| **SMPU** | Share Memory Pre-fetch Unit |
| **SNN** | Spiking Neural Networks |
| **SRES** | Stochastic Ranking Evolutionary Strategy |
| **TPU** | Tensor Processing Units |
| **U-NSGA-I** | Non-dominated Sorting Genetic Algorithm |

# List of Figures

# List of Tables

# 1 Introduction

## 1.1. Context and problem statement

Neural networks have become very prominent in the industry due to promising results in various applications. As many industries are moving towards AI-based solutions, the requirement for computing platforms that support such AI workloads are also increasing. And since there is a hype in supporting AI-based applications, this increases the requirement for computing platforms for these workloads even further. A data center filled with GPUs (graphics processing units), TPUs (tensor processing units), and IPUs (intelligent processing units) tends to support such workloads, but for this process, sending data to the cloud, computing and getting the data back could be expensive in terms of latency. This makes it important to run our AI algorithms at the edge form some applications. However, this puts significant low-power requirements on such edge applications.

To reduce energy consumption at the edge, researchers in the field have developed neuromorphic and Edge AI-based hardware, and research has been increasing to support such workloads better than classical Von Neumann architectures. As various industries such as defense, movie animation, medical, industrial automation, autonomous driving, etc. are moving towards AI-based solutions, the range of applications and size of the used deep neural networks (DNNs) has also been increasing with time. Since there is a great variation in the applications, industries, workloads, and size of DNNs, having a single computing platform that performs efficiently for all applications is quite a challenge. The challenge is to place the application in such a manner that results in the least energy and latency. The huge number of parameters that vary for various DNN applications makes it more difficult to come up with a single mapping strategy that reduces energy and improves latency. In addition, when dedicated silicon chips are manufactured for these applications, the fabrication of the chip is expensive; therefore, in the design phase, an estimate of the energy and latency of execution is important to make some critical architectural design decisions. Although neuromorphic chips try to reduce energy by only activating chips that require processing in spiking neural networks (SNNs), it was the first time that SNNs could be replaced by NNs and became computationally more powerful [3].

Neuromorphic hardware provides efficient implementations of SNNs and promises unprecedented energy efficiency for artificial intelligence [4] [5]. However, SNNs lack the precision and robustness of commonly applied deep neural networks. To overcome this gap, the 2nd generation SpiNNaker system integrates hardware accelerators for DNN layers in the individual processing elements (PEs). This flexibility allows one to choose between spiking and deep neural networks, depending on what is more efficient for the task at hand. Published research investigated a systematic distribution strategy for large convolutional neural networks (CNN) on SpiNNaker2, achieving inference times and energy efficiency that are orders of magnitude better than DNN applications on previous SpiNNaker systems [6] and competitive with dedicated DNN chips.

Although DNNs are, in essence, bioinspired, they have not been able to find the balance between power consumption and accuracy yet, especially when dealing with computationally heavy streaming signals. On the other hand, the brain's neocortex handles complex tasks such as sensory perception, planning, attention, and motor control while consuming less than 20 W [3]. Scalability, in-memory computation, parallel processing, communication using spikes, low-precision computation, sparse distributed representation, asynchronous execution, and fault tolerance are some of the characteristics of biological neural networks that can be leveraged to bridge the energy consumption gap between the brain and deep neural networks. Among these, the proposed methodology focuses on the viability of using sparsity within deep neural networks to achieve energy efficiency. During a matrix-vector multiplication between a weight matrix and an activation vector, zero elements in the tensor can be skipped, leading to reduced computational and memory access. Spiking neural networks models leverage the sparsity factor and improve the computation process in comparison to deep neural networks. In order to gain maximum performance, mapping these neural networks onto the neuromorphic chip and hardware is an essential task. The thesis work address a few research questions in the direction of mapping these neurons to a chip efficiently.

## 1.2. Challenges and contribution

In order to design and implement a hardware mapper (SENMap) that maps SNN/DNN applications to a custom neuromorphic platform/simulator (SENeCA/SENSIM), several challenges were presented from a software, hardware, and algorithm perspective at different stages of the project. To summarize the various challenges posed over the duration could be framed as a research and design question listed below in chronological order.

**Design, implementation and research questions**

1. **S** How to make the mapper flexible for various large-scale applications, make it compatible with the existing software framework and make it scalable for future projects?
2. **A** How to design a mapping algorithm and what preexisting algorithms could come into use?
3. **HA** How to partition or combine neurons in an SNN to map onto the hardware for better efficiency?
4. **A** What optimization algorithms to use for mapping, and what would be the criteria to choose from the available set of algorithms?
5. **A** How to form the optimization problem and how to formulate the objective function?
6. **H** What objectives need to be focused on while formulating the objective function?
7. **HA** What would be the constraints and bounds for the problem one has to optimize on?
8. **SA** How to speed up the process of finding the optimal mapping?
9. **SH** How to add more parameters to the existing mapping problem and find ways to design a better architecture (hardware/software co-optimization)?
10. **A** What factors depend on the mapping?

**S (software), H (hardware), and A (algorithms)** in the above list of research questions represents the perspective from which these questions were addressed. Most of these challenges posed as research questions will be addressed and answered throughout the thesis.

**Contributions**

In summary, the contributions of the work are as follows:

1. Design, implementation, and evaluation of SENMap (flexible hardware mapper), which adds to the preexisting software suite for neuromorphic architecture (SENeCA) designed by IMEC.
2. Use SENMap to extract energy-efficient SNN mappings for several SNN applications, such as PilotNet and MNIST.
3. Give a hypothesis of the possible energy reduction in the system.
4. Improve the time-to-solution for SENMap for single and multiobjective.
5. Evaluate possible flaws in the SENSIM and SENMap SDK frameworks and suggest improvements.

## 1.3. Thesis outline

The rest of the thesis is organized as follows. Chapter 2 gives a brief background on SNNs, other neuromorphic architectures, and details on SENeCA (neuromorphic architecture designed by IMEC) and SENSIM software framework, and the PilotNet SNN practical application used for experimentation. Understanding SENeCA and SENSIM will help to understand the execution model of an application on SENSIM and latency and estimation in the SENSIM framework. Chapter 3 dives into similar mapping tools developed for other architectures, algorithmic research available to us to choose from, and open source tools available to us at our disposal for integration in the custom mapper made for mapping applications on SENeCA. Chapter 4 takes into account the knowledge from Chapter 2 and Chapter 3 and details on the design and implementation of the hardware mapper (SENMap) and its integration with SENSIM. Chapter 5 elaborates on experiments conducted with the designed hardware mapper, records performance, and presents interesting results. Finally, Chapter 6 concludes the design and research work and sheds light on future research directions that could help improve the design of the hardware mapper (SENMap).

# 2 Neuromorphic background

This chapter will cover topics that provide a background to understand the design and implementation of the SEN-Map (hardware mapper). Section 2.1 gives a brief introduction about the recent developments in neuromorphic computing by elaborating on SNN models and recently fabricated neuromorphic architectures and chips in the market. This section also briefly describes the advantages recorded of neuromorphic computing in the literature. Section 2.2 takes in depth the SENeCA architecture developed at IMEC. This section also covers major design choices that helped reduce the energy footprint and make it scalable, concluding with a brief comparison with other similar neuromorphic architectures. This section also provides information on the software development kit in development that is used to simulate and map SNN/DNN applications to the SENeCA architecture. Section 2.3 details on the hardware-aware simulator that is used to run SNN applications in SENeCA. The section also covers details on several parameters it uses, the execution model implemented in SENSIM to mimic SENeCA's execution behavior, and details on energy and latency estimation in the process. The section concludes with an evaluation of SENSIM and briefs on the large-scale spiking neural network such as (PilotNet) used for experimentation and mapping of SENSIM in Section 2.4. For validation purposes, a 4 layered DNN trained on MNIST was also used.

## 2.1. General neurmorphic introduction

### 2.1.1. Spiking neural networks

Compared to deep neural networks, spiking neural networks (SNNs) go one step further and try to mimic natural biological neurons further by incorporating the concepts of time into their operating model. In addition to the state, weights, and biases of neurons in conventional neural networks, SNNs incorporate the concept of time into their operating model. The SNN neurons do not transmit information at each cycle (as happens with conventional neural networks), but rather transmit information only when a membrane potential, an intrinsic quality of the neuron, reaches a specific value, called the threshold. When the membrane potential reaches the threshold, the neuron fires and generates a signal (called spikes) that travels to other neurons, which also maintain a membrane potential in turn to change (increase or decrease) their potentials in response to this signal. A neuron model that fires at the moment of threshold crossing is also called a spiking neuron model.

Spiking neural networks, the third generation of neural network model entered the picture in [3] after the first-generation perceptrons and the second-generation deep and recurrent neural networks. First-generation neural network that has binary activation level, where the second generation implements a continuous activation function, making it suitable for analog input/output data. Compared to first- and second-generation neural networks, SNNs increase biological realism by using individual spikes. This allows the incorporation of spatial-temporal information in communication and computation, as do real neurons. Therefore, instead of sending data at a particular rate (rate coding) in second-generation neural networks, these neurons use spikes (pulse coding), where a train of spikes encoded in a message is transmitted once the threshold is reached. Mechanisms in which neurons receive and send individual pulses have been developed that allow multiplexing of information such as frequency and amplitude of sound. Recent discoveries in the field of neurology have shown that neurons in the cortex perform analog computations at incredible speed, as that humans analyze and classify visual input (i.e. facial recognition) in less than 100ms. It takes at least 10 synaptic steps from the retina to the temporal lobe; this leaves approximately 10 ms of processing time per neuron. This time window is much too small to allow an averaging mechanism such as rate coding. When speed is an issue, pulse coding schemes are preferred compared to rate coding.

Leaky Integrate-and-Fire [7] is one of the most prominent initially developed models where the momentary activation level is modeled as a differential equation of a neuron which is either pushed to a higher value or lower based on the incoming spikes until the state of the neuron eventually either decays below the threshold or is fired if the threshold is reached. After being fired, the state variable is reset to a lower value. There are some disadvantages of the Leaky Integrate and Fire neuron model that it does not contain neuronal adaptation. This disadvantage is removed in generalized integrate-and-fire models that also contain one or several adaptation variables and are able to predict spike times of cortical neurons under current injection to a high degree of accuracy in adaptive leaky integrate and fire. The resonance-and-fire neuron is similar to the integration-and-fire neuron, except that the state variable is complex. The model provides geometric illustrations of many interesting phenomena that occur in biological neurons that have subthreshold-damped oscillations of the membrane potential.

Similarly, there are other neuron models such as current-based leaky integration and fire [7], adaptive leaky integration and fire [8], adaptive resonance and fire [9], adaptive resonance and fire Izhikevich, Sigma Delta [10], etc. which are in parallel being researched. The SNN model implemented with SENSIM is the SIGMA DELTA MODEL described below in Section 2.1.2.

### 2.1.2. Sigma-Delta ($\sum \Delta$) SNN model

Sigma-Delta model originated while studying spiking neural networks, a couple of initial works closely related to the development of the model [11] [10]. Researchers noticed that DNNs can be obscenely wasteful. To improve performance, the researchers suggested sending changes from an activation function to the next layer during inference. Having been inspired by compression schemes, such as jpeg and mpeg, which introduce small compromises to image fidelity in exchange for substantial savings in memory, the sigma-delta model extracts the spatial-temporal sparsity. In neuroscience, [12] estimated that the human retina transmits 8.75 Mbps, which is about the same as compressed 1080p video at 30FPS, suggesting that there is some kind of spatial temporal compression within the human brain. Inspired by the same ideology, the Sigma-delta model was developed by extracting spatial temporal information from a neural network. The computational cost of running such a network would be proportional to the amount of change in input. There are several ways in which spatial and temporal information is extracted and sent from one layer to another for processing. There are methods such as temporal difference communication and rounding [10] which introduce sparsity by introducing Sigma-Delta Networks, where neurons in one layer communicated with neurons in the next layer through discredited delta activations. When it comes to change-based inference, there is an accumulation of potential errors over time, as the method is threshold-based. If the neuron states do not reset periodically, this threshold can cause drift in the approximation of the activation signal and degrade accuracy; hence there are training temporal delta layer based modified discussed in [13].

## 2.2. SENeCA architecture

SENeCA[1] is a RISC-V-based digital neuromorphic processor that accelerates SNNs for ultra-low-power extreme-edge applications where adaptivity is required. SENeCA is primarily targeted at computational load with unstructured spatio-temporal sparsity and erratic data transfers. SENeCA's digital IP comprises an interconnected neuron cluster of Cores with RISC-V-based optimized neuromorphic CoProcessor and its instruction set, and an event-based communication infrastructure.

SENeCA improves the state-of-the-art by addressing the flexibility issue in neuromorphic processors by allowing efficient execution for learning/adaptivity algorithms in addition to fully programmable neuron models. It also improves the state of the art by incorporating a 3-level memory hierarchy, which would also allow for the use of novel embedded memory technologies in the fabrication of the chip. The availability of 4b and 8b resolution data

[1]The content in this section is taken from the [2] and internal documents of SENeCA at IMEC.

types with a shared 8b scaling factor in the neuromorphic cores allow for the efficient deployment of advanced learning mechanisms and optimization algorithms.

Efficient event communication for accelerating neural operations is supported by using a new Network-on-Chip with multicasting, source-based routing, and a data compression mechanism. SENeCA can be tuned to a flexible number of cores and Neural Processing Elements (NPEs) per core and to the optional use of off-chip memory for a given application. For close-loop synthesis/mapping optimization, the SENeCA platform comes with a hardware-aware simulator called the SENSIM, which will be discussed in further detail in Section 2.3.

Figure 2.1 shows an instance of SENeCA with 64 neuron compute clusters (NCC). Each NCC core contains a RISC-V core (Ibex), instruction and data memory, Axon Message Interface (AMI), Network on Chip (NoC), Share Memory Prefetch Unit (SMPU), NPEs (Neural processing element), loop buffer, and much more which make the Neuron-Co-Processor (NCP). The following subsections include details on the subcomponents of the NCC core.

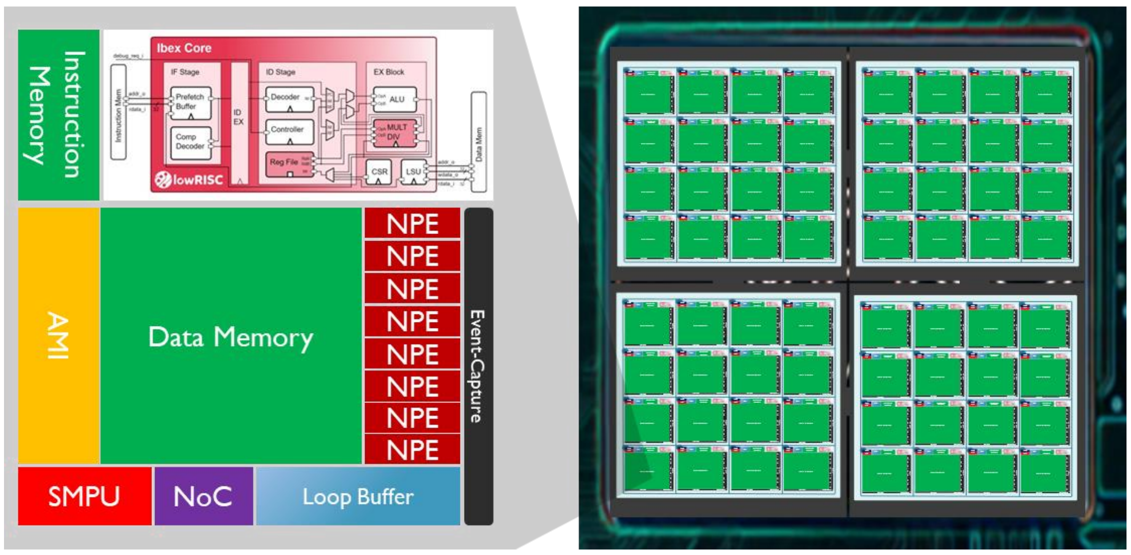


**Figure 2.1:** A 64-core SENeCA architecture [2]

### 2.2.1. Neuron Co-Processor (NCP)

SENeCA was designed for the purpose of having a flexible and programmable event-based accelerator, and the core element that makes it possible is the Neural Co-Processor (NCP). NCP emulates silicon neurons by accelerating the most common neuromorphic instructions at once. As neuromorphic computing is evolving and many new spiking neural network models constantly emerging, the NCP in SENeCA comes with a set of hardware-accelerated instructions (which make it scalable and flexible) to implement a variety of neuron models. NCP is made up of the following digital design blocks.

#### Neural processing element (NPE)

NCPs are made up of a SIMD array of time-multiplied silicon neurons called a neuron processing element (NPE). A single NPE is made up of a small memory (registers) to store data for a short duration and a processing unit to execute a category of the most common neuromorphic instructions. Each NPE can execute one instruction per cycle on average. All NPEs in one NCP will execute shared instructions on their specific input data at once, stored in individual registers. All NPEs are connected to the data memory to avoid memory bottlenecks.

Although the SENeCA architecture keeps the size of NPEs flexible and experimental results show 16-bit, since size of NPEs are the most energy efficient to reduce data read/write (the value might change based on the technology node, sparsity in the neural network, and few other parameters). NPEs support data representation in 4b and 8b resolutions, which are proven to be optimized for neural networks with a trade-off between energy and accuracy. The 4b and 8b resolution types are supported with a shared 8b scaling factor. In extreme cases, the implementation can support one scaling factor for a parameter processed by the processor. BF16 is an example of 8b data and the 8b scaling factor [14]. BF16 is a format suggested by Google to address the narrow dynamic range of IEEE FP16 also supported by TensorFlow for embedded deep learning applications [14]. The sharing of the scale factor is used in several integer quantization schemes [15] and in a few DNN accelerators [16].

With (8b, 4b) data types and shared scaling factor data type support, NPEs instructions implemented in the Ibex controller can also detect zero input early in the pipeline and stop the execution, therefore saving energy at its core in SENeCA. In addition, each NPE can be individually turned on or off by the Ibex to save energy during idle times. There are two power-saving modes in the NPE 1) Sleep and 2) Power-off. In sleep mode, the selected NPE´s will not execute any instruction, but the content of its register file will be preserved. In power-off mode, the selected NPE s will be completely turned off without preserving data in the register files. NPE´s are designed to also be aware of the sparsity in the data and do not compute on 0 input data. The size of the NPEs per NCP core is not fixed, but is kept flexible in the architecture.

**Event generator**
After the computation on the NPEs, spikes are generated, which are stored in the Event Generator unit that converts them into an Address Event Representation (AER) [17]. The Event Generator is part of the FIFO queue, as shown in Figure 2.2. In this format, spikes are encoded into packets of data that contain the address of the neuron that generated the spikes. The spikes can also have an optional value (known as graded spikes). These generated events are collected in the FIFO. When gathering events in the FIFO queue, the Ibex controller is interrupted for further operation.

**Loop buffer**
Whenever an event enters the AMI unit, several hundreds of neurons may need to be updated. Due to time multiplexing, each physical neuron must repetitively execute the same set of instructions in a loop. Involving the Ibex core to manage this loop takes too much of its time and is also less energy efficient (since Ibex needs to fetch each instruction from the instruction memory). This problem is conventionally addressed by a mechanism called a Loop Buffer. SENeCA's loop buffer is a small memory made with registers (which is therefore energy efficient). A local copy of the instructions is stored in the loop buffer and presented to the NPEs for the desired number of repetitions. In SENeCA, Ibex programs the loop buffer and triggers it for input events. Instructions for loop buffers were added. Loop buffer is not only a piece of memory. It contains small logic blocks to act as a program counter and calculate data-memory addresses. The loop buffer is shared among all NPEs. Figure 2.2 shows where the loop buffer would be located in the architecture.

**Axon massage interface (AMI)**
AMI is a programmable accelerator that manages incoming/outgoing events, event filtering, and flow control. AMI interrupts Ibex when there are enough events in the input queue to be processed.

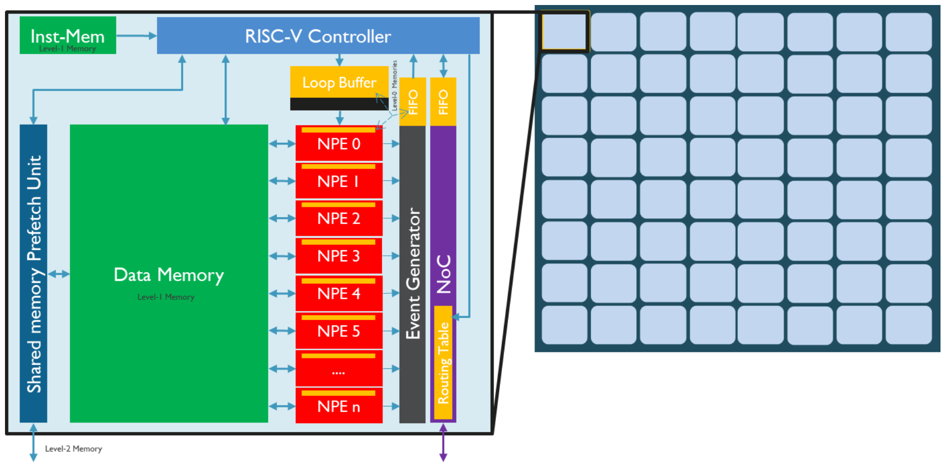


**Figure 2.2:** SENeCA pipeline [2] (internal IMEC)

### 2.2.2. Ibex core (RISC-V controller)

RISC-V ISA is already implemented on hardware with several open-source implementations. SENeCA was designed to be energy-efficient and area efficient. Having an Ibex core as the controller, not a processor, makes SENeCA quite energy efficient. Ibex Core is a small 32-bit processor with a 2-stage pipeline that acts as the controller for the NCC. The processor core can be highly parametrized and is well-suited for embedded control applications. Ibex core uses the RV32IMC instruction set. The highly programmable controller allows energy and area optimization in several ways, making SENeCA scalable and energy efficient. Figure 2.3 shows the internals of an Ibex core. [18]

The primary motivation for using the RISC-V controller for SENeCA was to make the platform user-friendly by integrating the available RISC-V compiler into the SENeCA SDK platform. Therefore, anyone can program SENeCA using a standard programming language. The presence of a RISC-V controller in SENeCA brings great flexibility and several functionalities to significantly improve overall energy and area efficiency. As mentioned above, in neuromorphic (and most modern) processors, memories are the primary source of area and energy consumption. A flexible RISC-V controller allows efficient memory management, which avoids redundant memory access and storage. With the RISC-V controller in the design, a unified memory architecture was possible, where the data memory can be dynamically allocated to the neuron and synaptic parameters during run-time. In addition, efficiently mapping various types of neural network architectures (e.g., Depth Wise/3D Conv, Skip connections) with flexible data formats and sparse representations into the memory was possible. The RISC-V controller helps to optimize mapping to make the best use of different memory technologies.

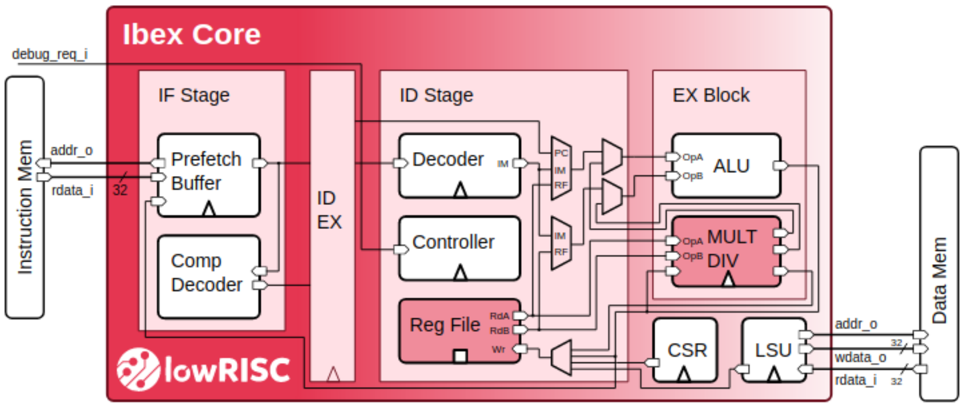


**Figure 2.3:** Internal structure of Ibex Core[19]

### 2.2.3. Shared memory pre-fetch unit (SMPU)

Apart from the dedicated data, the instruction memory per NCP as level 1 memories and registers, the FIFO and loop buffer as level 0 memories as shown in Figure 2.2, SENeCA comes with another level of memory hierarchy, which is shared among the few NCC can be seen in Figure 2.4. Since SENeCA was designed to be scalable, an external shared memory unit helps expand for applications that are memory intensive and goes beyond the area limitations of SRAM. Using shared memory improves area efficiency and shared memory can be implemented on the chip using denser memory technology such as eFlash, eDRAM or off-chip using 3D stacking memory technology [20] [21]. Shared-memory implementation in SENeCA is not implemented to share data among the processing cores (NCPs) like conventional GPU architectures, but is kept optional and will be used when applications are large and local memory on the NCP is not enough to store the data.

The shared memory prefetch unit (SMPU) is an optimized DMA (Direct Memory Accelerator) that accelerates access to shared memory through a direct link to the arbitrator of shared memory, as shown in Figure 2.5. To reduce the latency overhead, the SMPU unit pre-fetches the required parameters for processing an event from the shared memory while the event is queued in the NoC FIFO queue. Although the use of shared memory can be a good architectural choice, the energy and latency cost of data access is much higher than that of having data in local memory. One way to reduce the cost is to offload the variables that are less used in shared memory. Figure 2.4 shows the 64-core SENeCA instance with shared memory.

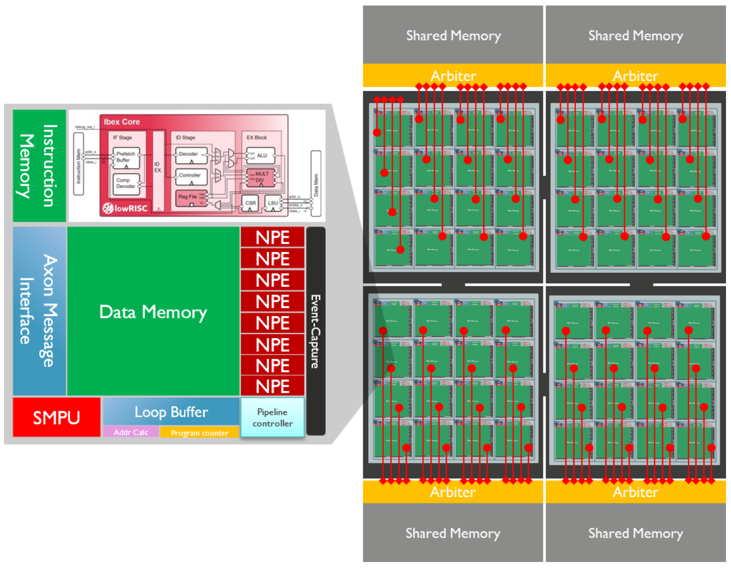


**Figure 2.4:** SENeCA Architecture with a level 2 shared memory [2] (internal IMEC)

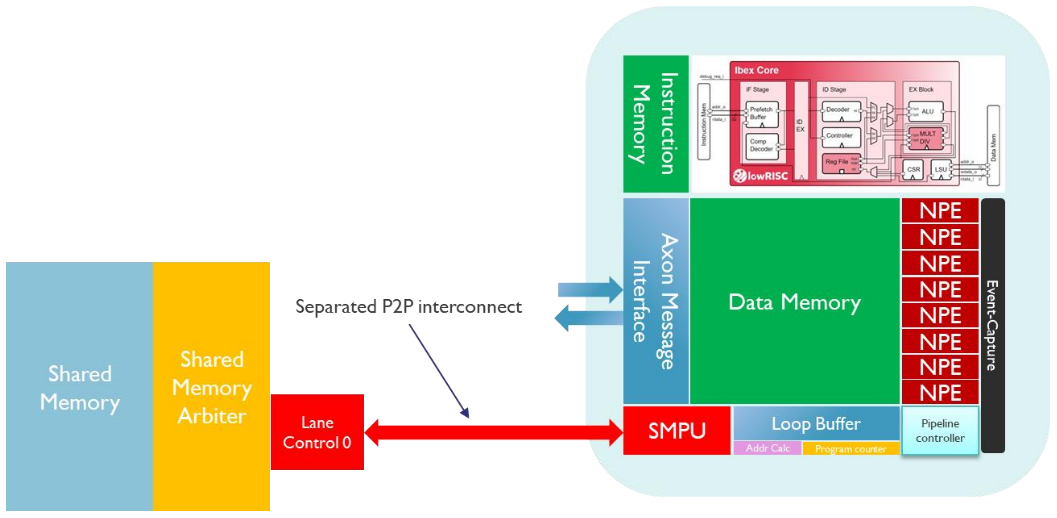


**Figure 2.5:** SENeCA Architecture with SMPU (Shared Memory Prefetch Unit) connected to the Shared Memory Arbiter [2] (internal IMEC)

### 2.2.4. Network on Chip (NoC)

To connect the NCPs and deliver the spike events, SENeCA implements a NoC that connects the NCPs. The NoC implemented in SENeCA consists of one router per NCP with a mesh-like architecture that connects each core with four other neighboring cores. The NoC is controlled by the Ibex controller. Communication in SENeCA is significantly reduced by using large cores to reduce intercommunication between the cores. SENeCA implements multicasting with source-based routing and variable-length packets for compression, which increases the operation density, reducing the latency.

In the implementation of SENeCA, the packets contain the source of the NCC instead of the destination of the NCC, which is different from the conventional implementation of NoCs on the chip. Routing occurs by maintaining a small routing table inside the routers. To avoid sending unwanted packets due to implementation, unwanted packets are filtered out by filters. These filters are stored in the register-based routing table inside each router; every filter implemented contains 3 fields (mask, key, and outports). When an event enters the NoC, the dot product (AND) of the event's Source Address and Mask will be compared with the key **(Source Address & Mask == Key)**. If the result is True, the event will be forwarded to the output ports. Routing tables can be reprogrammed by the Ibex core, allowing for synaptic plasticity. This process is repeated for all the filters implemented in the core. Increasing the number of filters increases the flexibility of routing. In structured connectivity, only a small set of filters is sufficient for efficient mapping. The routers in SENeCA do not have FIFOs that help reduce static energy and save space.

Data communication between NCCs can be expensive in terms of energy and latency; hence, the architecture tries to minimize the communication between the NCCs by implementing a SIMD array (single instruction, multiple data) of NPEs for parallel data processing within an NPE.

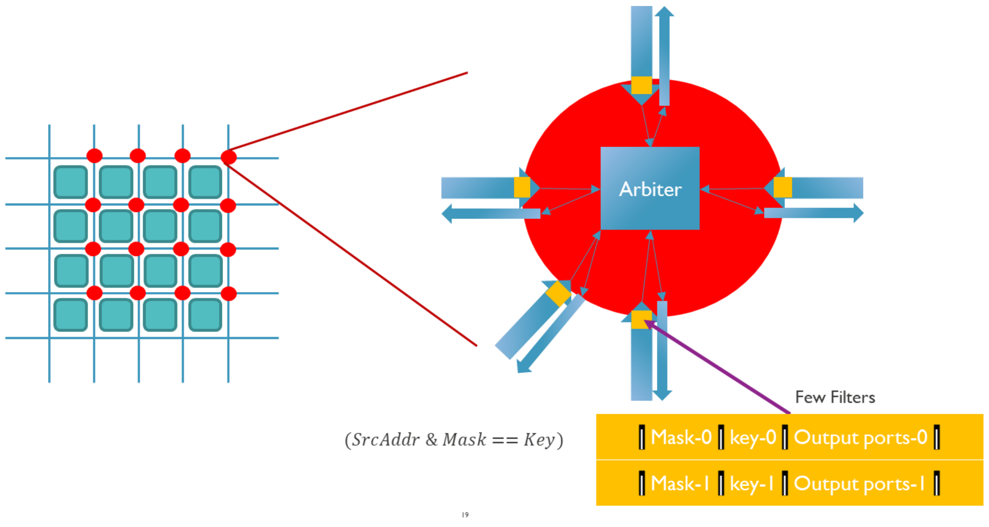


**Figure 2.6:** Network on a chip in SENeCA [2] (internal IMEC)

### 2.2.5. SENeCA SDK

SENeCA's SDK is mainly comprised of the following blocks.

- **DeltaDNN** is an open source tool for optimizing a trained DNN to improve its sparsity. The tool replaces the activation layer in the DNN with a delta activation layer, the cause of which the inference model only transmits and processes the changes in the output of each neuron. The temporal and spatial sparsity introduced in the process improves the energy and inference speed, which can be exploited by SENeCA's event-based execution model.
- **SNN Simulator** is GPU accelerated SNN simulator which reports the number of spikes firing in an SNN model and the number of operations executed in the SNN model. The simulator is used to perform fast hyperparameter tuning for pre-mapping. Although the simulator is fast, the execution of the model is not fast and there is a reasonable mismatch between the reported results and the event-based execution model in SENeCA.
- **SENSIM (Hardware-aware Simulator)** is a hardware-aware simulator that simulates post-mapping SNN models and gives an estimate of energy, latency, and accuracy. The tool executes models in a hybrid event-based and a time-driven mechanism designed to run on a conventional CPU not optimized for an asynchronous event-based processing. SENSIM provides a fairly accurate simulation compared to SNN and could be used for architectural exploration. SENSIM is described in detail in Section 2.3.
- **SENMap Hardware Mapper** A mapper that efficiently maps (in terms of latency and energy) SNN applications on SENSIM (a software model of SENeCA) through an iterative process. The design and implementation of the framework are detailed in Chapter 4.
- **Automatic Code Generator** converts the high-level neural network model and the hardware-mapped output to binaries and initializes the memory indices in SENeCA. The binaries output by the code generator are instruction files, data memory, and shared memory files for SENeCA before running any application.

Figure 2.7 shows the current SDK architecture. At the moment, the SENeCA SDK is in work in progress. While the other software framework is **DeltaDNN framework**, **SNN simulator** and **SENSIM** if the work is mainly completed, the development work for **Hardware Mapper** is covered in the later chapters of the thesis report. Although **DeltaDNN framework** is mainly inspired by the DeltaDNN DNN to SNN conversion model described above in Section 2.1.2, the SENSIM framework is described in detail in Section 2.3 later in this chapter.

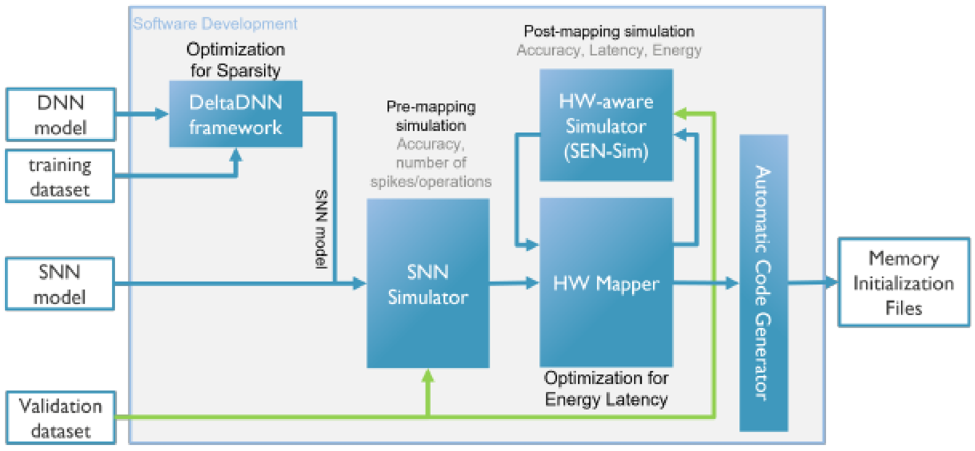


**Figure 2.7:** SENeCA Software Development Kit [2] (internal IMEC)

### 2.2.6. Comparison of SENeCA with other neuromorphic chips

SpiNNaker architecture [22] has many similarities with the SENeCA neuromorphic platform. In contrast to SENeCA, SpiNNaker is composed of multiple ARM cores connected through an advanced single-router star-type multicast asynchronous packet-switched network. SpiNNaker2 [23] added several accelerated arithmetic processing units and advanced power management techniques to the GF22nm technology node. In contrast, SENeCA uses the smallest open-source RISC-V processors as the controller (not used for event processing but for control operations), together with optimized accelerators. and a low-overhead mesh-type multicasting NoC (with reduced functionality compared to SpiNNaker) for sparse parallel event-based computation. Unlike SpiNNaker, which is designed for the simulation of brain-inspired research, the primary purpose of SENeCA is to open both hardware and software for optimizations and innovations in EdgeAI neuromorphic computation.

On the other hand, IBM TrueNorth [24] uses a plain mesh packet-switched network (unicast) but with optimized (inflexible) processing cores. Each core in the TrueNorth architecture emulates exactly 256 neurons. Each neuron has 256 input synapses organized in a cross-bar architecture, with a single output axon connected to 256 neurons in another core. This optimized processing core resulted in a power-efficient neuron update (approximately 26pJ). µBrain [25] goes further in optimizing the processing core and allows for application-specific IP with ultralow power (in contrast to the multipurpose neuromorphic processor).

In Intel Loihi [4], the processing cores are more flexible than in TrueNorth, and the interconnect is a simple unicast packet-switched mesh. Additionally, Loihi cores accelerate a bioinspired learning algorithm. The cost of this flexibility is having a higher neuron update energy (about 80pJ) compared to TrueNorth (while using a better technology node). Loihi2 [26] scaled the Loihi chip by packing more neurons and synapses in a die, using the Intel4 technology node. Additionally, it introduced microcode-programmable neurons, a feature that is also available in SENeCA. Both Loihi chips accelerate a specific kind of bioinspired learning mechanism on the chip.

Recently the spinnaker 2 [27] was released with a design that improves the energy by 45%. s The interplay between the two domains SNN and DNN by using DNoC and CNoC and several other modifications such as the GALS (Globally Asynchronous and locally Synchronous) and DVFS (Dynamic voltage and frequency scaling ) could also be used as a modification to SENeCA.

## 2.3. SENSIM framework

SENSIM is an open-source event-based neuromorphic simulator developed in Python designed by IMEC that assists the neuromorphic chip designer in simulated SNN models with hardware-aware parameters in software and provides an estimate of energy and latency. Although the SENSIM framework helped to realize and develop the initial SENeCA architecture, the framework can be scaled to support the development of other event-based neuromorphic architectures. SENSIM provides detailed temporal energy/latency measurements based on counting operations and memory access for a given application. To improve the execution time of SENSIM, parallel processing support with threads was also added. The current version of SENSIM models a GALS system in which reliable data transfer between independent clock domains. Although SENSIM and SENeCA are undergoing several changes, hence the model of SENSIM could change in the future. SENSIM can also be scaled to model other distributed systems on a large scale.

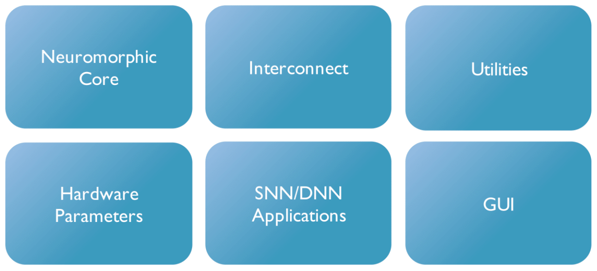


**Figure 2.8:** SENSIM software framework

### 2.3.1. Neuromorphic core

At the heart of the SENSIM simulator lies the neuromorphic core, which attempts to mimic the behavior of a neuromorphic core in the SENeCA architecture Section 2.2. A chip designer has the flexibility to vary the following parameters to his own convenience, but in order to maintain similarity with other cores on the chip, all the cores are initialized with the parameters except the location and name. Table 2.1 are the parameters with which the core is initialized. **Name** and **Location** of the core remain unique for every core on the chip.

| **Parameter** | **Description** |
|---|---|
| Name | The parameter is a source of identification for the Neuromorphic core |
| Location | The parameter decides the placement of the core on a 3D memory grid. |
| Number of NPE's | The parameter decides the number of NPEs in the core. |
| Queue Depths | The parameter decides the input and output queue size for the core. |

**Table 2.1:** Parameters for composition of a Neuromorphic core

Apart from the above parameters, a core is initialized with the following attributes, which may vary in a core for a given simulation.

| Attribute | Description |
|---|---|
| Energy | The attribute keeps track of the energy utilized by the core in processing data. The energy attribute also keeps track of the individual elements of the core, for instance, the NPE, FIFO, DMEM, and the controller. |
| Time | The attribute keeps track of the time in a given simulation. |
| Idle Time | The attribute keeps track of the time the core or idle and not processing any data. |
| Queue | The attribute keeps track of the 2 sub attributes - **In-Queue** and **Out-Queue**. The **In-Queue** keeps track of the data incoming to the core and being stored till the NPE's process it. Similarly, the **Out-Queue** stores the processed data before sending it to the next core. |
| Reset | The attribute is associated with the core reset. The attribute brings back the core to the initial state. |

**Table 2.2:** Attributes of the Neuromorphic Core (NCC) in SENSIM

To mimic the behavior of neuromorphic core in software, SENSIM comes with several functionalities.

**Mapping neurons to a core**
In order to process data from an SNN application, ideally, a core must know which neurons or a range of neurons will be assigned to the core. Before the simulation starts, the mapping of neurons to the core is achieved by mapping the mapped package, for example, `[(L1, range(0,600), CMR), (L2,range(600,1000), CMR)]` where the parameters of the mapped package are detailed in Table 2.3

| Contents | Description |
|---|---|
| Layers | A string which denotes the layer number from the SNN. For instance, *L1, L2.. etc.* |
| Range of Neurons | The range of neurons describes the range of neurons from an SNN which will be mapped to a core |
| Cache miss rate | A floating point/ rational value which lies between 0 and 1 which states the percentage/fraction of neurons and weights would be placed in the internal data memory of the NCP whereas the remaining fraction placed in the external/shared memory. |

**Table 2.3:** Contents of the mapped package

The core supports more than one or more than one mapped package being mapped to a core. It is possible to map a layer twice in one core as `[(L1,range(0,600), CMR1), (L1,range(800,1000), CMR2)]` where `CMR1` and `CMR2` are the cache miss rate for the respective range of neurons.

**Spikes package in SENSIM**
SENSIM can incorporate several spike compression schemes that have a variable spike to flit to event compression schemes. Following is a packet format Figure 2.9 used by the simulator. The address event representation is the same as that of SENeCA and only a timestamp is added in case of SENSIM.

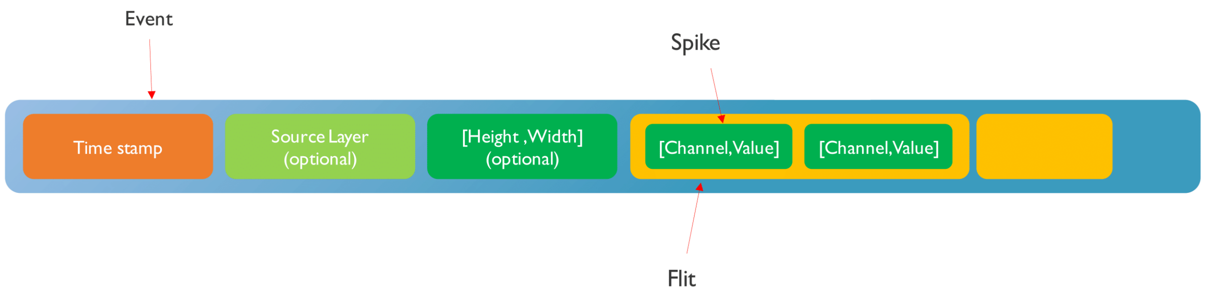


**Figure 2.9:** Event packet format for SENSIM

**Execution model of the neuromorphic core**
Every core in the multicore chip goes through the following execution stages.

***Note:*** *Every core maintains an* ***internal output queue*** *in addition to the* ***actual output queue***

**Step 1:** Check for events in the input queue. If there are no events in the core's input queue, the core will remain idle. When a core is idle, time units are added to the **Idle Time** attribute of the core for later analysis.

**Step 2:** Check if the internal output queue is filled or not. A core cannot generate new events when the internal output queue is filled, but it can still process incoming events. If there are events in the internal output queue, the events are copied to the actual output queue until it is full and deleted from the internal output queue. To copy certain events from the internal queue to the actual output queue, the energy and time are estimated and added to the core's attributes.

**Step 3:** After the above steps, in the case of time synchronization is **True** the core is interrupted flag (which occurs by interrupting the core timer) based on the evaluation time parameter from Section 2.3.4 the neurons are evaluated. This interrupt mechanism helps avoid over-utilization of the core. When the evaluation of neurons of a layer already mapped to a particular core is performed, and in this process, no new events are added; therefore, the neurons are not updated at this step.

**Step 4:** If the time-synchronization interrupt is not triggered and the above checks are passed means that there are events in the input queue to be processed by the core. Events are deleted from the input queue as they are processed by the core, and the neuron states are updated. As the events are fetched from the input queue and processed, the timestamp on the events is also updated by the time spent in the FIFO queue. ***Note:*** *The evaluation step only occurs when the consume and fire parameter is* ***False*** *and there are no events in the input and output queues.*

**Step 5:** If the layer for which the neurons and weights are external memory, The energy consumed to retrieve the data from the FIFO queue and the energy consumed by the controller to obtain the location of the pointer for the weights and bias values are estimated by Equation 2.4 and Equation 2.5.

**Step 6:** Based on the number of target layer/s in a given source layer assigned to the core and the type of layer, the layer processing is defined in Section 2.3.1. Additional checks are implemented to check if events from target layers are looped back to the same core.

**Step 7:** At the end of every time step, the total energy associated with the core is updated by different energy attributes of the core.

The energy consumption to read the data in and out of a FIFO queue can be expressed by the following equations. The energy consumed by the FIFO queue and the controller to send the data out of the fifo queue a for a given core can be expressed as the sum of events/flits written to the queue, as shown in Equation 2.1 and Equation 2.1. Similarly, the time consumed to write the events in the fifo queue is expressed as Equation 2.3. The energy consumed by a neuromorphic core to read the fifo data can be expressed as Equation 2.4. The energy consumed by the Ibex controller to obtain instructions for the location of the weight and the neuron pointer can be expressed as shown in Equation 2.5.

$$E_{fw}(C(x,y)) = \sum_{t=0}^{ts} E\_fifo\_wr \times cfe(ev) \times fw \tag{2.1}$$

$$E_{controller}(C(x,y)) = \sum_{t=0}^{ts} E\_CON \times cfe(ev) \tag{2.2}$$

$$T_{fifo} = \sum_{t=0}^{ts} cfe(ev) \tag{2.3}$$

$$E_{fr}(C(x,y)) = \sum_{t=0}^{ts} E\_fifo\_rd \times cfe(ev) \times fw \tag{2.4}$$

$$E_{controller\_PAI}(C(x,y)) = \sum_{t=0}^{ts} PAI \times E\_CON \tag{2.5}$$

$$E_{fifo}(C(x,y)) = E_{fr}(C(x,y)) + E_{fw}(C(x,y)) \tag{2.6}$$

where:

| | |
|---|---|
| $E_{controller\_PAI}$ | = energy of the controller to get the ponter location of the weights. |
| $E_{controller}$ | = energy consumed by the controller for reading and writing events |
| $E_{fifo}$ | = energy consumed by the core for reading and writing events |
| $E_{fr}$ | = energy consumed by the core to read from the fifo queue |
| $E_{fw}$ | = energy consumed by the core to write to a fifo queue |
| $cfe$ | = a function which counts the flits in the events |
| $ev$ | = events sent out after evaluation |
| $fw$ | = flit_width |
| $C(x,y)$ | = Core at (x,y) coordinate |

**Processing layer on the core**

The core processes every type of layer differently. Since the convolution layer and the dense layer are the most commonly implemented layers in a neural network, SENSIM implements the sigma-delta neuron processing model for the layers. The following are the steps involved in processing a dense layer via the sigma-delta model on the core.

***Note:*** *The processing of neurons in a convolution differs slightly due to the stride and the pooling parameters of a convolution process, and therefore there are some additional preprocessing steps (such as the calculation of the projection field to estimate the number of neurons to update) involved in the extraction of neurons, which have to be updated in a layer based on events from the previous layer. The rest of the process of updating neurons remains the same in dense and convolution layers. Therefore, for brevity, the steps involved in processing a dense layer will only be discussed in this Section.*

***Note:*** *The process of estimating the delta, updating the states, and sending non-zero events after Step 3 is detailed in the next subsection.* ***evaluation of neurons****. The evaluation of neurons is performed on the basis of the time-synchronization parameter 'Async' or 'Time-step Flag'. If the evaluation time-synchronization is set to 'Async' (Asynchronous), the output is evaluated in the same time step as they are stored in the registers of NPE for processing. If the evaluation time-synchronization is 'Time-Step Flag' based, the evaluation of neurons occurs at the evaluation time step* $t_{ev}$

**Step 1:** Read weights, bias, thresholds, and neuron states from an external file. The energy and time consumed to retrieve the weights and neurons from the data memory to the registers is then calculated.

**Step 2:** Calculate the new neuron states (new_state). Estimate the time and energy consumed at every step.

**Step 3:** Evaluate the results based on the quantization of the sigma-delta. `delta = quantize(f(new_state)) - quantize(f(old_state))`

**Step 4:** Update the states.

**Step 5:** Send nonzero events to the internal output queue.

Equation 2.7 gives the energy to read a neuron from the data memory or external memory. **E_Dmem_rd** and **E_Dmem_rd** are the energy parameters of Table 2.6 and **CMR(N)** represents the Cache miss rate for neurons in Equation 2.7.

$$E_{nr}(C(x,y)) = (1 - CMR(N)) \times E\_Dmem\_rd + CMR(N) \times E\_ext\_mem\_rd \quad (2.7)$$

Equation 2.8 gives the energy to write the updated neuron to the data memory or external memory. **E_Dmem_wr** and **E_Dmem_wr** are the energy parameters of Table 2.6 and **CMR(N)** represents the cache miss rate for neurons in Equation 2.8.

$$E_{nw}(C(x,y)) = (1 - CMR(N)) \times E\_Dmem\_wr + CMR(N) \times E\_ext\_mem\_wr \quad (2.8)$$

Equation 2.9 gives the energy to read a weight from the data memory or external memory. *E_Dmem_rd* and *E_Dmem_rd* are the energy parameters of Table 2.6 and **CMR(W)** represents the cache miss rate for the weights in Equation 2.9.

$$E_{wr}(C(x,y)) = (1 - CMR(W)) \times E\_Dmem\_rd + CMR(W) \times E\_ext\_mem\_rd \quad (2.9)$$

Equation 2.10 gives the time required to compute the output of a neuron over time once the weights and neurons are available in the data memory. $N_{weights}$ and $N_{spikes}$ represent the number of weights and spikes to be calculated in the NCP in Equation 2.10. **T_NPE** is taken from Table 2.7

$$T_{compute} = \sum_{0}^{t=ts} N_{spikes} \times \left\lceil \frac{N_{weights}}{N\_NPE} \right\rceil \times T\_NPE \times 2 \quad (2.10)$$

Equation 2.11 gives the time required to retrieve neurons from external memory to data memory. Equation 2.12 gives the time required to retrieve neurons from external memory to data memory. Equation 2.13 gives the total time required to access the weights and neurons of external memory. When estimating the time here, the assumption is taken that enough bandwidth is available to read and write 1 word (32b) per cycle.

$$T_{en} = \sum_{0}^{t=ts} CMR(N) \times N_{neurons} \times 2 \times \frac{BW\_states}{BW\_ext\_mem} \quad (2.11)$$

$$T_{ew} = \sum_{0}^{t=ts} CMR(W) \times N_{weights} \times N_{spikes} \times \frac{BW\_weights}{BW\_ext\_mem} \quad (2.12)$$

$$T_{ext} = T_{en} + T_{ew} \quad (2.13)$$

The final time to process the data in the NCP for a given time step is taken as the maximum time to retrieve the data from the external memory or the time to compute as expressed in Equation 2.14. The time computed at this step is used to time-stamp all the events at this stage.

$$T = max(T_{ext}, T_{compute}) \tag{2.14}$$

Equation 2.15 gives the energy consumption to read neurons from data memory. $N_{weights}$ and $N_{spikes}$ represent the total number of weights read and spikes fired in a given time step. **BW_weights** is the application parameters of Table 2.10. Energy estimation takes into account the reading of all weights for any number of spikes in the layer.

***Note:*** *The number of weights read can be reduced based on having more information on the weights that are used more in the SNN vs. the weights that are less used in an SNN*

$$E_{dn}(C(x,y)) = \sum_{0}^{t=ts} N_{weights} \times N_{spikes} \times E_{wr}(C(x,y)) \times BW\_weights \tag{2.15}$$

Equation 2.16 gives the energy consumption to read and write neurons from data memory. The reading and writing of the controller neurons depend on the number of spike registers in the controller.

$$E_{dw}(C(x,y)) = \sum_{0}^{t=ts} N_{neurons} \times \frac{N_{spikes}}{N_{SpikeRegister}} \times (E_{nr}(C(x,y)) + E_{nw}(C(x,y))) \times BW\_states \tag{2.16}$$

Total energy consumed by data memory in reading and writing neurons is given by Equation 2.17

$$E_d(C(x,y)) = E_{dw}(C(x,y)) + E_{dn}(C(x,y)) \tag{2.17}$$

Equation 2.18 and Equation 2.19 are the total energy of the NPE and controllers. The total energy of the controller is the sum of Equation 2.19 and Equation 2.5

$$E_{npe}(C(x,y)) = \sum_{0}^{t=ts} N_{neurons} \times N_{spikes} \times 2 \times E\_NPE \tag{2.18}$$

$$E_{controller}(C(x,y)) = \sum_{0}^{t=ts} \left\lceil \frac{N_{neurons}}{N\_NPE} \right\rceil \times N_{spikes} \times 2 \times E\_CON \tag{2.19}$$

Equation 2.20 gives the total energy for a time step by adding all the energy consumption for the controller, NPE, data memory, and FIFO in the core of Equation 2.19, Equation 2.4, Equation 2.17, Equation 2.18.

$$E_{total}(C(x,y)) = E_{controller}(C(x,y)) + E_{npe}(C(x,y)) + E_d(C(x,y)) + E_{fifo}(C(x,y)) \tag{2.20}$$

Once the state of the neurons is updated, the update flag of the neurons in the layer is set to True, which means the neurons are updated.

**Evaluation of neurons**
In case the time synchronization of the system is set to the 'Time-Step Flag' the evaluation of neurons does not happen at the same time as reading the weights, neuron states, and updating them. Evaluation of the output delta, updating the states, and sending non-zero events out of the output queue of the neuromorphic core. The following are the steps involved in the process.

**Step 1:** Evaluate outputs `delta = quantize(f(new_state)) - quantize(f(old_state))`
**Step 2:** Update states
**Step 3:** Send non-zero events out.
**Step 4:** Estimate the energy and time consumed at each step.

For the evaluation-type timestep synchronization flag, neurons that have been updated also maintain a flag called the update flag. The update flag is first evaluated for all mapped neurons. The energy consumed by the memory to read the flag in the hardware is accelerated by an event generation process to detect non-zero values, once for every 16 bits. Therefore, the energy consumed by the NPEs to read the update flag is given by equation . The energy consumed by the controller to read the update flag can be expressed as Equation 2.41. The time to read the update flag can be expressed with an Equation 2.41.

$$E_{npe}(C(x,y)) = \sum_{t_{ev}}^{t_s+t_{ev}} E\_NPE \times 2 \times \frac{N_{nuf}}{16} \tag{2.21}$$

$$E_d(C(x,y)) = \sum_{t_{ev}}^{t_s+t_{ev}} E\_Dmem\_rd \times N_{nuf} \tag{2.22}$$

$$E_{controller}(C(x,y)) = \sum_{t_{ev}}^{t_s+t_{ev}} E\_CON \times N_{nuf} + 2 \times \frac{\prod N_{nuf}(x)}{N_{npe}/16} \tag{2.23}$$

$$t_{ruf} = \sum_{t_{ev}}^{t_s+t_{ev}} T\_NPE \times N_{nuf} + 2 \times \frac{\prod N_{nuf}(x)}{N_{npe}/16} \tag{2.24}$$

where:

| | |
|---|---|
| $N_{nuf}$ | = number of neurons with the flag updated |
| $t_{ruf}$ | = time to read the update flag for all the neurons |
| $x$ | = dimensions of the vector |

Neurons for any given layer can be present in external memory or in local data memory. The estimation of energy consumed by neurons to read and write from and to external memory is given by the following equations, which are further used in later equations. ***Note:*** *The reading of neurons from data memory is repeated only in asynchronous mode, where neuronal evaluation is performed at each time step and not at each evaluation time step ($t_{ev}$)*

$$E_{nr}(C(x,y)) = (1 - CMR(N)) \times E\_Dmem\_rd + CMR(N) \times E\_ext\_mem\_rd \tag{2.25}$$

$$E_{nw}(C(x,y)) = (1 - CMR(N)) \times E\_Dmem\_wr + CMR(N) \times E\_ext\_mem\_wr \tag{2.26}$$

The energy consumed to read the updated neurons is given by Equation 2.27 . Equation 2.28 representing the energy consumed reading the old neuron output that needs to be updated. The energy consumed by NPEs for the evaluation and quantification of the activation function at every time step of the evaluation is given by Equation 2.29

$$E_d(C(x,y)) = \sum_{t_{ev}}^{t=t_s+t_{ev}} E_{nr} \times N_{ntu} \times BW\_States \tag{2.27}$$

$$E_d(C(x,y)) = \sum_{0}^{t=ts} E_{nr} \times N_{ntu} \times BW\_Outputs \tag{2.28}$$

$$E_{npe}(C(x,y)) = \sum_{0}^{t=ts} E\_NPE \times N_{ntu} \times 2 \tag{2.29}$$

where:

$N_{ntu}$ = Number of neurons that need to be updated.

The RISC-V controller needs to read the state of the neuron, old output of the neurons, and hence the energy consumed by the controller is twice that consumed by the NPEs. Since the neurons are updated channel-wise and the neurons in the channel are updated simultaneously on the basis of the number of NPEs, the equation is divided by the number of NPEs. The energy consumed by the controller is expressed as Equation 2.30

$$E_{controller}(C(x,y)) = \sum_{t_{ev}}^{t=t_s+t_{ev}} E\_CON \times \left\lceil \frac{N_{npc}}{N_{npe}} \right\rceil \times (3 + 1 \times F_{async}) \tag{2.30}$$

where:

| | |
|---|---|
| $N_{npc}$ | = Number of neurons per channel |
| $N_{npe}$ | = Number of neural processing elements |
| $F_{async}$ | = A flag which will take the 'true' value only if the mode is Time step synchronization. |

The processor can give the time to read the neurons, the old-neuron output, and execute the activation function Equation 2.31.

$$t_{rso} = T\_NPE \times N_{ntupc} \times \left\lceil \frac{N_{ntupc}}{N_{npe}} \right\rceil \times (2 + (1 \times t_{la}) + 1 \times F_{async}) \tag{2.31}$$

where:

| | |
|---|---|
| $E_{npe}$ | = energy consumed by neural processing elements to extract nonzero delta |
| $N_{ntupc}$ | = number of neurons to update per channel |
| $t_{la}$ | = activation time |
| $F_{async}$ | = a flag to control the synchronous mode and synchronization of time steps |

If the output of the neuron is stored in external memory, the time it takes to read the output can be expressed as follows. For reading and writing from external memory, it can be assumed that there is enough bandwidth available to read/write 1 word (32b) per cycle.

$$t_{eto} = CMR(W) \times N_{ntu} \times \frac{BW\_Outputs}{BWext\_mem} \tag{2.32}$$

$E_{npe}$ = energy consumed by neural processing elements to extract nonzero delta
$N_{ntupc}$ = number of neurons to update per channel
$t_{la}$ = activation time
$F_{async}$ = flag for synchronization.

The energy and time consumed for extracting the non-zero delta out can be expressed in the equation below. The energy consumed to extract non-zero events is carried out by Equation 2.33. The energy consumed by the controller to extract non-zero events can be expressed by Equation 2.34

$$E_{npe}(C(x,y)) = \sum_{0}^{t=ts} E\_NPE \times N_{ntu} \tag{2.33}$$

$$E_{controller}(C(x,y)) = \sum_{t_{ev}}^{t=t_s+t_{ev}} E\_CON \times \left\lceil \frac{N_{nupc}}{N_{npe}} \right\rceil + 3 \times F_{async} \tag{2.34}$$

where:

$N_{ntu}$ = a number of neurons to update.
$N_{nupc}$ = number of neurons per channel

The time consumed to extract non-zero events can be expressed as Equation 2.35

$$t_{nzd} = F_{async} \times T\_NPE \times N_{nupc} \times \left\lceil \frac{N_{nupc}}{N_{npe}} \right\rceil + 3 \times F_{flatten} \tag{2.35}$$

where:

$t_{nzd}$ = time to extract non-zero delta / changes
$N_{nupc}$ = neuron updates per channel.
$F_{flatten}$ = a Boolean flag that is true for flattening when the convolution layers are converted to the dense layers.
$F_{asunc}$ = a Boolean flag which is true for flattening in async mode.

After estimating the spike value with the threshold, the energy consumed to update the output of neurons can be given by the following equations Equation 2.36 and Equation 2.37 . The time to update the old state of the neurons is given by the following equation Equation 2.43. The time to send the non-zero delta out can be given by Equation 2.39

$$E_d(C(x,y)) = \sum_{0}^{t=ts} E_{nw} \times N_{snzd} \times BW\_outputs \tag{2.36}$$

$$E_{controller}(C(x,y)) = \sum_{t_{ev}}^{t=t_s+t_{ev}} E\_CON \times N_{snzd} \tag{2.37}$$

$$t_{nzd} = T\_NPE \times N_{snzd} \tag{2.38}$$

$$t_{ewo} = CMR(N) \times N_{snzd} \times \frac{BW\_outputs}{32} \tag{2.39}$$

$N_{snzd}$ = Number of non zero delta sent out
$t_{uos}$ = time to update old state
$t_{ewo}$ = time to write the output back to the external memory

Data memory consumes energy to reset the update flags of neurons. Since updating the neuron flags is equivalent to writing data back to the memory, the equation is expressed as Equation 2.40

$$E_d = \sum_{0}^{t=ts} E\_dmem\_wr \times N_{ntu} \tag{2.40}$$

$N_{ntu}$ = number of neurons to reset the update flag
$W_{dim}$ = dimensions of the weights

The energy consumed by the RISC-V controller to reset the update flags of neurons is given by Equation 2.41. The controller can reset at least one line of the kernel in one cycle; hence, the weights are divided by **min(**$W_{dim}$**, 32)**

$$E_{controller}(C(x, y)) = \sum_{0}^{t=ts} E\_CON \times \frac{N_{npe}}{min(W_{dim}, 32)} \tag{2.41}$$

where:

$t_{urf}$ = time to reset the update flag
$N_{ntu}$ = time to update the reset flag
$W_{dim}$ = dimensions of the weights

The time to reset the update flag in neurons is given by the equation below. Since the RISC-V controller can reset one row of the kernel in one clock cycle, the time is divided by the shape of the weights in one dimension.

$$t_{urf} = T\_NPE \times \left\lceil \frac{N_{ntu}}{min(W_{dim}, 32)} \right\rceil \tag{2.42}$$

where:

$t_{urf}$ = time to reset the update flag
$N_{ntu}$ = time to update the reset flag
$W_{dim}$ = dimensions of the weights

The total time for the evaluation of neurons is given by the equation below.

$$T_{ev} = max((t_{urf} + t_{rso} + t_{uos} + t_{nzd} + t_{rso}), (t_{ero}, t_{ewo})) \tag{2.43}$$

where:

$T_{ev}$ = total time to evaluate the neurons
$t_{ruf}$ = time to read the update flag
$t_{rso}$ = time to read the old output state and apply an activation potential
$t_{uos}$ = time to update old state
$t_{nzd}$ = time to extract non-zero delta/changes in frame
$t_{urf}$ = time to reset the update flag
$t_{ero}$ = time to read the output from the external memory
$t_{ewo}$ = time to write output back to the external memory

The time it takes for the processor to send every spike can be given by Equation 2.44. This time is timestamped to the packages at the end of the evaluation time step once the neurons are evaluated and a non-zero delta is sent out.

$$T_{sp} = \left\lceil \frac{T_{ev}}{N_{spnzd}} \right\rceil \tag{2.44}$$

where:

$T_{so}$ = time to send each spike out
$T_{ev}$ = total time to evaluate all the neurons
$N_{spnzd}$ = number of spikes with the non-zero delta / change

The energy consumed by the FIFO and the neuromorphic core controller to send events after evaluation can be expressed as Equation 2.45, Equation 2.46. The time consumed to send out the events is given by Equation 2.47

$$E_{fifo}(C(x,y)) = \sum_{0}^{t=tev} E\_fifo\_wr \times cfe(ev) \times fw \tag{2.45}$$

$$E_{controller}(C(x,y)) = \sum_{t_{ev}}^{t=t_s+t_{ev}} E\_CON \times cfe(ev) \tag{2.46}$$

$$T = \sum_{t_{ev}}^{t=t_s+t_{ev}} cfe(ev) \tag{2.47}$$

where:

$T$ = Total time to send the spikes out
$cfe$ = a function which counts the flits in the events
$ev$ = events sent out after evaluation
$fw$ = flit width

**Queuing model of the neurmorphic core**
Every NCP has 2 queues, the In queue and the Out queue. The In queue takes in events from the interconnect bus and stores them in the queue until the processor is ready to compute on them. Once, the processor computes the events and puts them to the out-queue. The out queue based on the source address and filters would multicast it to the destination cores according to the routing table.

In the queueing framework, every queue has a particular set of parameters.

| Parameter | Description |
|---|---|
| name | name of the queue |
| size | size of the queue |
| rd_pointer | A pointer which refers to the last read in the queue |
| wr_pointer | A pointer which refers to the last written data in the queue |
| n_events | Number of events to be processed in the queue |
| n_spikes | Number of spikes in an event queue |

**Table 2.4:** Attributes of a queue

Like any other queue, the queueing framework gives added and removing events functionality from and to the queue. The following steps explain how events are added and removed from the queue.

### 2.3.2. Interconnects framework

Although in SENeCA the NoC is designed to multicast packets, the interconnect framework gives options for unicast or multicast events to the destination core. The following are the steps that are executed in one step to process events in the interconnect.

***Note:*** *The design of the NoC in SENSIM resembles a segmented bus and attempts to mimic a multicast NoC.*

**Step 1:** Loop over all the output event queues for master cores and deliver their events to the input event queues of slave cores

**Step 2:** In the process, it guarantees that input event queues are sorted in time and returns the number of event-flits transferred in this time-step.

**Step 3:** If one of the slave input queues of the destination core is full, events will not be dropped or transferred to another destination if the parameter **flow control** is set to `strict`

**Step 4:** Depending on the communication parameter `multicast`, the total connection length is calculated, which is further used to calculate the total number of flits in the segment in one time step.

**Step 5:** Depending on the same communication parameter `multicast`, the average communication length is calculated, which is used to calculate the energy consumed by the bus segment to deliver the packets and also what the average delay would be to deliver the packets from the master core to the slave core.

The time and energy estimations are given by the following equations.

$$T = \sum_{t=0}^{ts} cfe(ev) \times T\_bus \times \frac{td(M_q, S_q)}{\sum S_q} \tag{2.48}$$

$$E(seg) = \sum_{t=0}^{ts} cfe(ev) \times E\_bus \times \frac{td(M_q, S_q)}{\sum S_q} \tag{2.49}$$

$$nf = \sum_{t=0}^{ts} cfe(ev) \times td(M_q, S_q) \tag{2.50}$$

where:

| | |
|---|---|
| $E$ | = total energy consumed by the interconnect framework |
| $T$ | = total time the flits spend in the queue |
| $nf$ | = number of flits processed in the queue |
| $cfe$ | = a function which counts the flits in the events |
| $ev$ | = events in the queue |
| $td$ | = total hops from master to slave queue |
| $M_q$ | = master core queue |
| $S_q$ | = slave core queue |
| $seg$ | = segment connecting the master core queue to slave core queues. |

**Total hops calculated for multicast event forwarding**

**Step 1:** Get the location of the master core from the location parameter of the core in Table 2.1

**Step 2:** Get the location of the slave cores in the list (since it is multicast) from the slave core location parameter as mentioned in Table 2.1

**Step 3:** Calculate the distance from the master to all slave cores by counting the positive hops and the negative hops.

**Step 4:** Get the total distance or a total number of hops by subtracting the negative hops from the positive hops.

**Total hops calculated for unicast event forwarding**

**Step 1:** Get the location of the master core from the location parameter of the core in Table 2.1

**Step 2:** Get the location of the slave cores from the location parameter of the slave core as mentioned in Table 2.1

**Step 3:** Calculate the distance from the master to the slave core by adding the total number of hops and taking the absolute value.

The interconnect framework takes into account the assumption that the output event queue is sorted in time and that no interspike distortion is caused by the queue.

### 2.3.3. Compositions of layers

In deep neural networks, there are several types of layer, and each type of layer needs to be processed differently in the NCP. To simulate the processing of a layer on the processor, SENSIM preprocesses layers from an SNN, which is further used to map the neurons on the core.

The following are the parameters used by the stage.

| **Parameter** | **Description** |
|---|---|
| layer_ID | Identification for the layer |
| name | Name of the layer. This identification is used to map the layers on cores. |
| shape | The shape of the layer |
| layer type | The type of operation the layer will perform `input`, `output`, `convolution`, `dense`, `flatten`. |
| pooling | Type of pooling layer. For instance, average pooling, max pooling, strided etc. |
| pooling size | In case of a stride layer, the stride value in [x,y] direction. For a stride value 2, the input to this parameter will be [2x2] |
| is_flatten | A boolean parameter which controls if the layer's data needs to be compressed in a single dimension. |
| neuron type | Type of data processing in the silicon neuron. For instance, in case of Delta inference model, the neuron type would be SigmaDelta. |
| threshold | Thresholds for the neurons to fire in an SNN. |
| activation function | Activation function the layer incorporates. For instance `Linear`, `ReLU`, `sigmoid` |
| weight tensor | The tensor with computed weight values for the layer which needs to be programmed on the simulated hardware. |
| bias tensor | The tensor with computed bias values for the layer which needs to be programmed on the simulated hardware. |
| output layer | The next layer to which the processed events will be delivered to. |

**Table 2.5:** Parameters used for the composition of a layer in SENSIM

### 2.3.4. Simulation parameters

To better understand the behavior, how each parameter affects the estimated end energy and latency, SENSIM can be configured with the following physical, energy, communication, flow control, and architectural parameters before simulation. Some of these parameters are chosen on the basis of the technology node, while others are initial architectural choices made.

The following are the energy parameters of the technology node. All default values taken for energy estimation are relative values measured through internal tools and not absolute values. If the technology node changes, the energy estimation of the node also changes.

| **Parameter** | **Value** | **Description** |
|---|---|---|
| E_CON | 3 | energy per each controller operation and instruction memory read |
| E_NPE | 1 | energy unit for each NPE operation |
| E_Dmem_rd | 3 | energy unit for each data memory read bit |
| E_Dmem_wr | 3 | energy per each data memory write bit |
| E_ext_mem_rd | 300 | energy per each shared memory read bit |
| E_ext_mem_wr | 300 | energy per each event queue read bit |
| E_fifo_rd | 1.5 | energy per each event queue read bit |
| E_fifo_wr | 1.5 | energy per each event queue write bit |
| E_bus | 6 | energy for sending a bit of data on a bus segment |

**Table 2.6:** Energy parameters for SENSIM

Table 2.7 are the time parameters of the technology node. All default values taken for time estimation are taken in terms of the clock period. The values mentioned in the table below are compared to one RISC-V cycle. Parameters can be varied on architectural exploration.

| Parameter | Value | Description |
|---|---|---|
| T_NPE | 1 | time per each NPE operation |
| T_fifo | 1 | time per each event queue access |
| T_bus | 1 | time for sending a flit of data for one bus leg |
| T_ext_mem | 100 | latency for accessing the shared memory |
| time_step | 1000 (cc) | The time step at which every NCC evaluated events |
| time_evaluation | 50000 (cc) | The time step at which neurons are evaluated with the threshold |

**Table 2.7:** Time parameters for SENSIM

Table 2.8 are the communication parameters of the NoC architecture. All default values are chosen on the basis of experiments with the SENSIM framework. Parameters can be varied on the basis of the communication framework designed by the neuromorphic architect.

| Parameter | Value | Description |
|---|---|---|
| max_event_flits | 9 | limit the max number of flits per event |
| spike_per_flits | 2 | number of [C,Value] per each flit |
| header_flits | 1 | number of flits for the header [Source Layer, H, W] –> can be 0 for single flit AER events |
| BW_ext_mem | 32 | shared memory bandwidth (bits per cycle) |
| flit_width | 32 | number of bits per flits |

**Table 2.8:** Communication parameters for SENSIM

Table 2.9 briefs the physical design parameters that can be varied at the architectural level. These parameters were also chosen on the basis of previous manual experiments with SENSIM.

| Parameter | Value | Description |
|---|---|---|
| queue_depths | [64,64] | A global parameter which sets the input and output queue depth of every core on the chip |
| N_NPE | 16 | A global parameter which sets the total number of NPEs on every neuromorphic core on the chip. |
| mesh_size | [64,64] | A parameter which decides to total number of NCCs on the neuromorphic chip in the simulation |
| clock_frequency | 200 Hz | clock frequency at which each processor is running |
| Core_Dmem | 1024*1024 | Data memory of each core (in bit) |
| N_Spike_Register | 8 | Number of spike register for storing events |

**Table 2.9:** Physical parameters of SENSIM

Table 2.10 the application parameters such as the bit width of the output, weights, and states are chosen by quantization experiments.

| Parameter | Value | Description |
|---|---|---|
| BW_outputs | 1 | Bit Width per state (in bits) to store temporary states of a Neuron in a simulation |
| BW_weights | 4 | Bit Width per weight (in bits) to store Weights of a Neural Network |
| BW_states | 16 | Bit Width per state (in bits) to store temporary states of a Neuron in a simulation |

**Table 2.10:** Application parameters for SEMSIM

Table 2.11 shows SENSIM simulation parameters. The parameters decide the number of physical cores needed to run the core process and the bus process.

| Parameter | Value | Description |
|---|---|---|
| NrPhysicalCore4Cores | 1 | Number of CPU cores dedicated for running the Neuromorphic core operations |
| NrPhysicalCore4Buses | 1 | Number of CPU cores dedicated for running the Interconnect operations |
| snapshot_log_time_step | 10000 | The time at which the snapshots of the simulator are taken. |

**Table 2.11:** Simulation parameters of SENSIM

In the application parameters Table 2.12. The parameters decide the type of communication between the cores, how the information is delivered between the cores, and also how the processing and execution takes place in the cores.

| Parameter | Value | Description |
|---|---|---|
| Sync type | timeStep flag | Sync type is the timestep flag sync by default. In the sync type Apart from timestep synchronization where neurons with an update flag are to be evaluated at every time step, there is an async mode where the neurons are fired anytime once they cross the threshold. |
| suppress bias wave | False | (by default the update flag for all the neurons is set to True) a flag which makes the initial update flag 0 and may result in inconsistent outcomes from the timestep flag. |
| consume then fire | False | First priority is to consume input events, so there will be no evaluation until the input queue is empty time. |

**Table 2.12:** Timing and synchronization parameters in SENSIM

### 2.3.5. Input output data processing

For each SNN model for the purpose of testing inference, the framework also implements functionality that buffers the input validation data set and relays it to the framework, which mimics the chip.

Similarly, the framework also captures the intermediate output state of the SNN simulation on SENSIM by capturing the data of the output membrane at regular intervals, which can be compared to the output when the SNN is executed on a Keras framework executed on a GPU/CPU. The time at which snapshots are captured can be changed by the simulator framework.

For PilotNet, the input buffer is of size $66 \times 200 \times 3$. PilotNet outputs steering angles; hence, the output buffer is 0. In the case of MNIST, the input buffer would be $28 \times 28$ and the output buffer would be of size 10.

After buffering the input, the framework converts `frames_input_events` and sends it to the input queue of the input layer. Once the output events are received on the output queue of the output layer, the events are combined again to form meaningful data and dumped into a file for evaluation and comparison.

The following are some parameters for editable input and output processing.

| Parameter | Description |
|---|---|
| plot_outputs | A Boolean value which would save plots of output values/data at intermediate step |
| input_frame_buffer | A tensor/array of zeros initialized to save intermediate output value. In the case of an MNIST, the input buffer is initialized with 28 x 28 |
| output_frame_buffer | A tensor/array of zeros initialized to save intermediate output value. In the case of MNIST, the buffer would be 10 for 10 class output |
| time_input_event | The time at which the input will be passed to the buffer after the last frame is converted to events has ended. |
| num_samples | Number of samples used from the whole dataset for inference purposes. The number of samples could vary varying the end energy and latency. |
| fps | frames per second are used in case the frames are captured at a particular rate. In the case of an event-based camera, the frames per second are 0 which means the capturing rate is as fast as the system can process the data. |
| last_sample_converted | Keeps a track of the last data sample converted in the complete set of samples. |
| input_image | *(optional)* A file pointer pointing to the input dataset. |
| threshold | A file pointer pointing to the file where threshold values are stored |

**Table 2.13:** Input output simulation parameters

**Frame to event conversion**
In order to process the input frame in the simulator, the frames must be converted to events. The steps to convert a frame to an event are listed below.

**Step 1:** Convert frames (in the case of image a 3D frame) from the input buffer and convert it 2D frame tensor

**Step 2:** Use the parameters `max_event_flits`, `spike_per_flit`, `flit_width` and `header_flit` to form an event from the 2D tensor.

**Step 3:** Add the time between events to reduce the rate.

**Step 4:** The pixels are shuffled to avoid top-to-bottom orders. The shuffling of pixels helps in event-based systems, as the pixels which have to be processed can be there at random.

**Step 5:** Generate delta events that are essentially the difference between the input and the frame buffer.

The following are some parameters for converting a frame to an event.

| Parameter | Description |
|---|---|
| input_frame | The input sample from the data set |
| frame_buffer | A previous input sample stored in the frame buffer which is used to calculate the delta and generate events. |
| threshold | Uses the thresholds for every frame for quantizing |
| parameters | Takes in `max_event_flits`, `spike_per_flit`, `flit_width` and `header_flit` parameters |
| input_queue | The Queue of the input layer to which the events will be sent to after conversion |
| time_between_events | An integer value which sets a time between events |
| shuffle_pixels | A boolean flag which gives control on whether to generate shuffle pixels or not |
| delta | A boolean flag which gives control on whether to generate delta events or not |
| input_layer | Input layer ID to which the input events will be sent to after conversion from frames |
| input_time | The simulation time at which the simulator will inject the frame data for frame to event conversion |

**Table 2.14:** Frame to event conversion parameters

**Event to Frame Conversion**
In order to get the output of the SNN after an iteration of execution on SENSIM, the events are accumulated from the input queue of the output layer, and based on the source address, they are arranged and the output is stored in the output buffer.

## 2.3.6. Application definition

For any SNN application to be implemented in the SENSIM framework, an application (for example, PilotNet, MNIST, etc.) must be manually assigned to the simulator. The following are the steps involved in defining an application in the SENSIM framework.

- **Initialize simulator parameters** The stage configures the simulator by setting parameters such as `Simulation end time`, `Simulation evaluation step`. `Snapshot log time`, etc. A complete list of the simulator parameters that can be varied can be found in Table 2.11.
- **Composing Layers** The stage creates layers from the SNN model that can be assigned to the hardware. It uses the parameters described in Table 2.5
- **Composing Layers Core Map** The stage maps the layers and neurons from a layer to a particular core. The details of the mapping are discussed in Chapter 4 under Section 4.1
- **Composing Interconnects** The stage creates segments by connecting the master cores to the slave cores to communicate between the cores. For example, if layer1 is assigned to core1 and layer2 is assigned to core2 and the data travel from core1 to core2, core1 is the master core, and core2 is the slave core. More information on the interconnection framework is described in Table 2.11

More details on the definition of the application will be discussed in Chapter 4 with the design and implementation of SENMap (Hardware mapper).

## 2.3.7. Additional utilities

**GUI setting file generation**
On every simulation, there are generated files that help configure the GUI framework for post-simulation analysis. There is a utility that generates a set of parameters and stores them in a csv file. The file stores the following parameters listed in Table 2.15

| Parameter | Description |
|---|---|
| type_layout | The type of layout the simulator will use to simulate the |
| no_of_rows | The number of rows in the mesh/layout. |
| no_of_colums | The number of column in the mesh/layout |
| core_name | The name of the core |
| core_x | The x coordinate of the core considering the first core in the mesh [0,0,0] |
| core_y | The y coordinate of the core considering the first core in the mesh [0,0,0] |
| num_mapped_layer | The number of layers mapped to the core |
| layer_name | Based on the number of layers mapped the parameter |

**Table 2.15:** Parameters for GUI setup

**Snapshot file generation**
Since SENSIM is an event-based simulator, snapshots were stored for all cores, interconnects and output values.

| Snapshot header | Description |
|---|---|
| snapshot_ID | The snapshot Identification Number |
| core_name | The name of the core for which the parametric numbers are presented |
| time(cc) | The time in clock cycles |
| internal_time | The internal time for any given core |
| peak_in_queue | In one time step, the peak value of the input queue for a given core |
| peak_out_queue | In one time step, the peak value of the output queue for a given core |
| packet_loss | The total packets lost in given time step |
| energy_total | The total energy consumed by the core in a given time step |
| idle_time(cc) | The time the core was idle in a given time step |
| processor_utilization(ratio) | The percentage the core was utilized in a given time step |
| in_queue_occupancy | The in queue occupancy at the end of the time step for a given core |
| out_queue_occupancy | The out queue occupancy at the end of the time step for a given core |
| energy_controller | The energy consumed by the controller in a given time step for a given core |
| energy_npe | The energy consumed by all the NPEs in a given time step for a given core |
| energy_dmem | The energy consumed by data memory read and writes in a given time step for a given core |
| energy_fifo | The energy consumed by fifo queue read and writes in a given time step for a given core |
| core_x | The x coordinate of the core considering the first core in the mesh [0,0,0] |
| core_y | The y coordinate of the core considering the first core in the mesh [0,0,0] |
| energy_total(ratio) | ask amir |
| peak_in_queue_occupancy (ratio) | The highest percentage (peak value), the In queue was occupied in a given time step |
| peak_out_queue_occupancy (ratio) | The highest percentage (peak value), the Out queue was occupied in a given snapshot time step |
| snapshot_time | The time duration of one snapshot in terms of clock cycles |

**Table 2.16:** Snapshot header for NCPs

| Snapshot header | Description |
|---|---|
| snapshot_ID | The snapshot Identification Number |
| time(cc) | Time in clock cycles |
| name | Name of the segment |
| total_flit | total flits in the bus segment |
| energy_bus | total energy in the bus segment |
| internal_time | internal time of the processor |

**Table 2.17:** Snapshot header for Interconnect

### 2.3.8. SENSIM execution model

SENSIM uses a Python standard library for multiprocessing to speed up the simulation execution process. The number of physical CPU cores used by the simulator is determined by the simulator parameters in Table 2.11. The processes to be run by the neuromorphic cores and segments are synchronized before executing the simulator. Regardless of the number of physical CPU cores, the simulator goes through the following steps.

**Step 1:** Before running the simulation, based on the mesh size, the number of cores is initialized by the simulator.

**Step 2:** Based on the number of layers in the application, the mapping package is created and neurons are mapped to the neuromorphic cores in the mesh.

**Step 3:** Based on the interconnection between the application layers and the mapping of the cores on the neuromorphic processor, the segments are initialized.

**Step 4:** Once the neurons are mapped on the cores, the input data is fetched for processing.

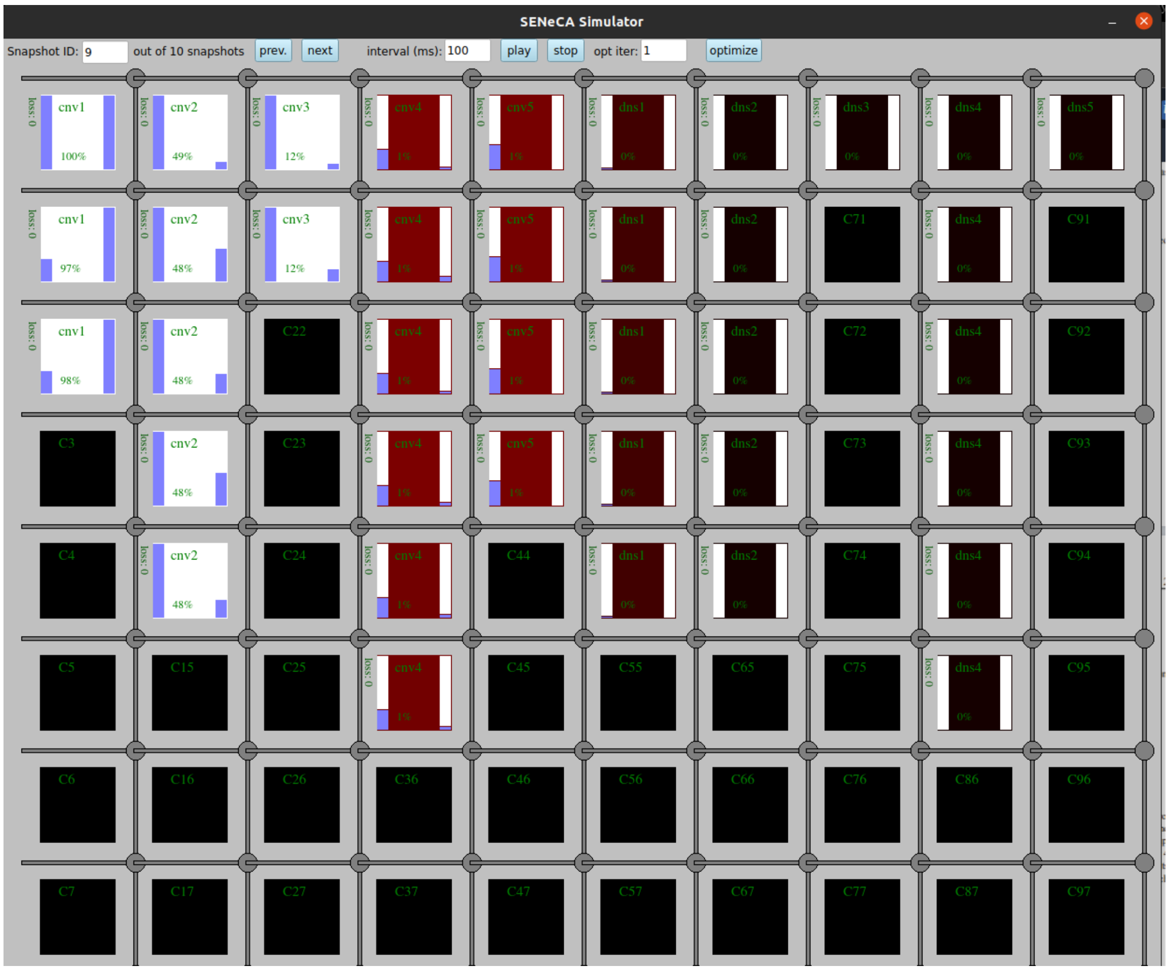
**Step 5:** The list of cores and segments starts its individual process in parallel, and the data is processed by the core and segment as detailed in Section 2.3.1 and Section 2.3.2.

**Step 6:** Snapshots are captured on the basis of the snapshot log-time parameters in Table 2.10.

### 2.3.9. SENSIM GUI

The GUI for SENSIM is implemented using the PyQt [28] and tkinter [29] libraries in the framework. The GUI essentially sprints through the snapshots saved during the simulation and gives a visual overview of how the application would execute on the Neuromorphic processor. The GUI gives a quick overview of processor utilization and input/output queue utilization of every core at every time step for the complete simulation. While the GUI does give a good overview of the processor post-simulation, it does not simulate the working of the neuromorphic core in real time.

Figure 2.10 shows PilotNet Mapped to SENSIM with white cores utilized the most, black cores the least, and red cores moderately used. The blue bars on either side of the GUI represent the input and output queues at any given time. The **snapshot ID** can be seen in the upper left corner, with the buttons **prev** and **next** to individually analyze each step in the simulation. **play** and **stop** buttons play and stop the simulation on offline execution of the GUI framework.


SENeCA Simulator

Snapshot ID: 9 out of 10 snapshots

interval (ms): 100

opt iter: 1

**Figure 2.10:** SENSIM GUI

### 2.3.10. Evaluation of SENSIM

#### Advantages of SENSIM

- SENSIM provides emulation of the scaled-up platform before silicon realization, which is not possible on FPGA and is slow on low-level software.
- SENSIM gives an opportunity to explore architecturally Interconnects, synchronization, flow-control, Neuron models, etc
- With SENSIM it is possible to benchmark emerging memory technologies such as, for example, non-volatile memory, high bandwidth memory [30].
- SENSIM provides early optimization and development of a scalable platform.
- SENSIM can provide iterative closed-loop mapping (after the implementation of the mapper).
- SENSIM provides relative power/energy/latency estimation by estimating memory access and operations on the chip.
- SENSIM provides the status of the cores, event queues, and idle times of the processor at each time step.

#### Limitation and opportunities in SENSIM

- The hardware mapper for efficient energy has not yet been implemented. The thesis work contributes to the development of the hardware Mapper.
- Mapping multiple layers to one core is supported by looping over events from the source layer to the destination layer by looping over the packets.
- Multicast does not imitate the hardware in the current version of SENSIM but is more like a segmented bus.

- The area is not modeled by the SENSIM software framework, which might pose a limitation in the final chip design.
- The communication framework between two layers assigned to the same core is currently implemented as a segmented bus that loops over the packets from the output queue of the master core and sends them to the slave core. **Note:**. The segmented bus averages the energy consumption based on events from the source core to the destination core.
- SENSIM does not incorporate support for other SNN models (Resonate and Fire [31], CUBO (current-based leaky integrate and Fire) [7], Integrate and Fire [32], [9] [8]), but is quite flexible for further addition.
- Other RTL, chip design simulators are required to get technology node parameters like energy consumption and area estimates.
- The data memory per core is not modeled in SENSIM, making it difficult to get a precise estimate of parameters such as the energy consumption for read and write.

## 2.4. Neural network applications

The mapping was tested with the following 2 applications, 10-Layered PilotNet SNN and 4-Layered MNIST DNN.

### 2.4.1. PilotNet-10 layered SNN applicaiton

PilotNet [33] [34] is a deep neural network developed by NVIDIA, as part of their complete software stack for autonomous driving, which takes live images from the camera installed in the front center of the car and outputs the angle at which the car should steer. PilotNet is a robust model trained with road images collected in different weather conditions - clear, cloudy, foggy, snowy and rainy - and performs well regardless of the presence of lane markings. The system is fed with input images captured from a front camera, and the angle of the steering wheel is predicted to be 1/r, where r is the turning radius of the car. The authors of the article reasoned that the use of 1/r makes the prediction more independent of the used car and the use of 1/r instead of r avoids infinity values when driving straight.

Figure 2.11 gives a pictorial representation of the Neural Network architecture of PilotNet. The neural networks consist of 9 layers (including a normalization layer), 5 convolutional layers, and 3 fully connected layers. The input image is of shape $66 \times 200$ with 3 channels divided into YUV. The normalization layer is hard coded and the parameters of the layers do not change while training. Following the normalization layer, convolutional layers, chosen empirically by a series of experiments that varied the layer configurations, were designed to perform feature extraction. The first three convolutional layers were chosen to have a stride of size $2 \times 2$ and a kernel of size $5 \times 5$. The next 2 layers are convolution layers with a stride of $1 \times 1$ with a kernel size of $3 \times 3$. Although the model is quite efficient and robust in predicting steering angles and gives autonomy up to 98%, the model does not predict when to accelerate or when to brake. It contains 108k neurons and 1.6M parameters. To convert DNN to SNN, the Sigma Delta model is used and an additional temporal delta activation layer [18] is added at each layer, which reduces the amount of information transferred from one layer to another by channel-wise thresholding in the case of PilotNet.

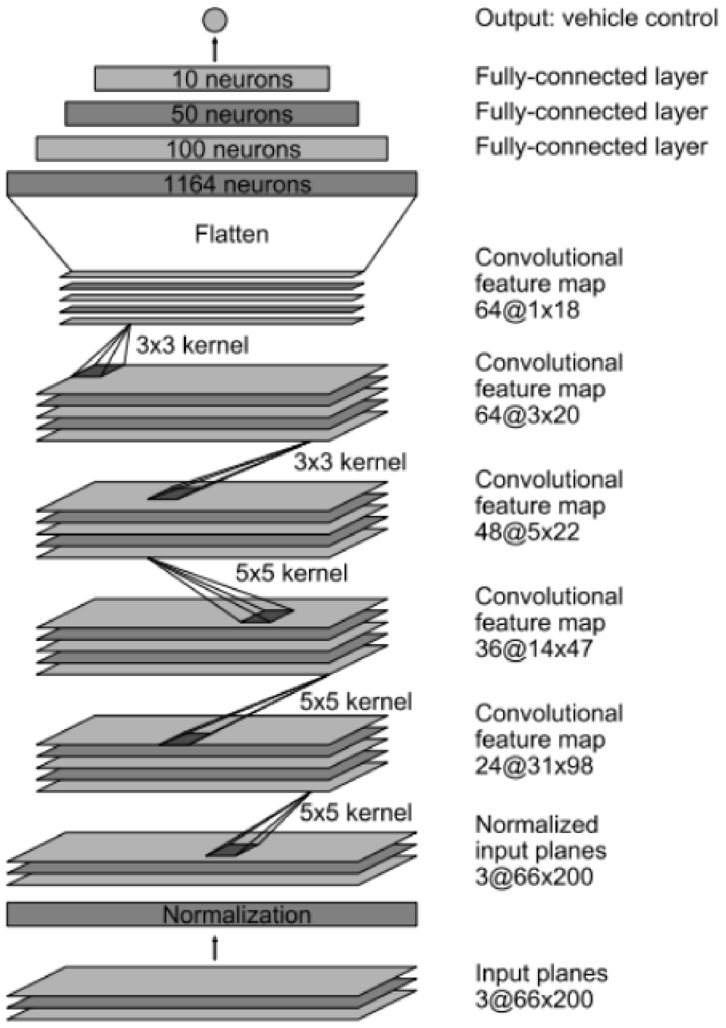
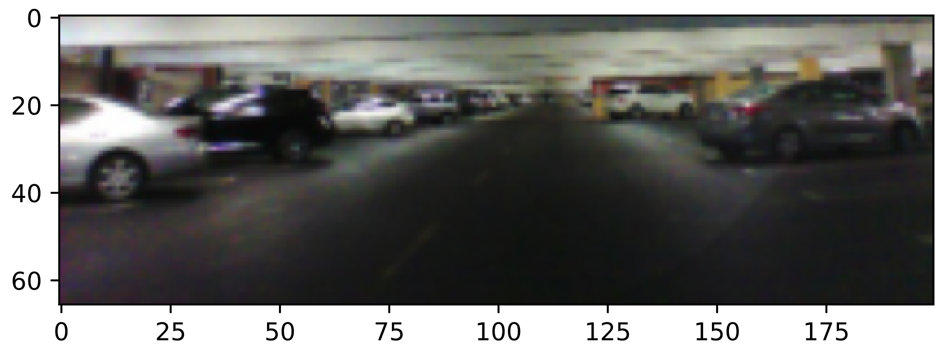
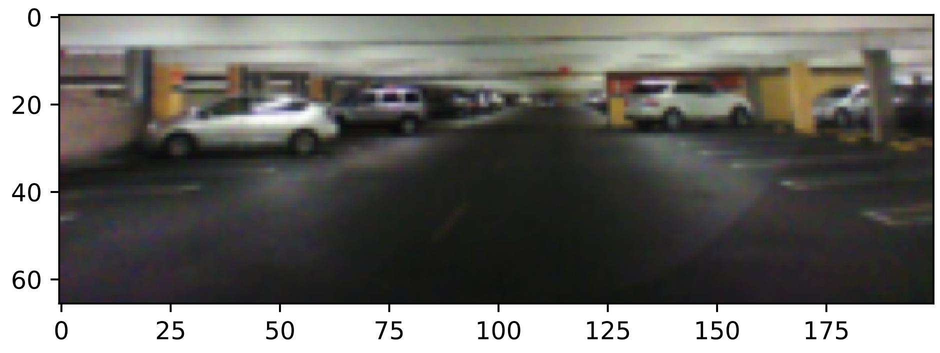


**Figure 2.11:** PilotNet architecture [34]

To test and validate the execution of the neural network, the pilotNet framework data set was used in the experiments. The validation data set has 2000 images, some of which are shown in Figure 2.12 and Figure 2.13.



**(a)** $1^{st}$ frame in the PilotNet Dataset[34]

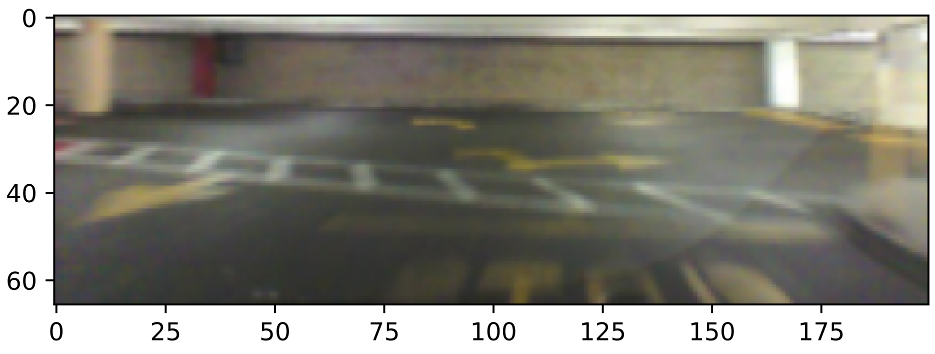


**(b)** $50^{th}$ frame in the PilotNet Dataset[34]

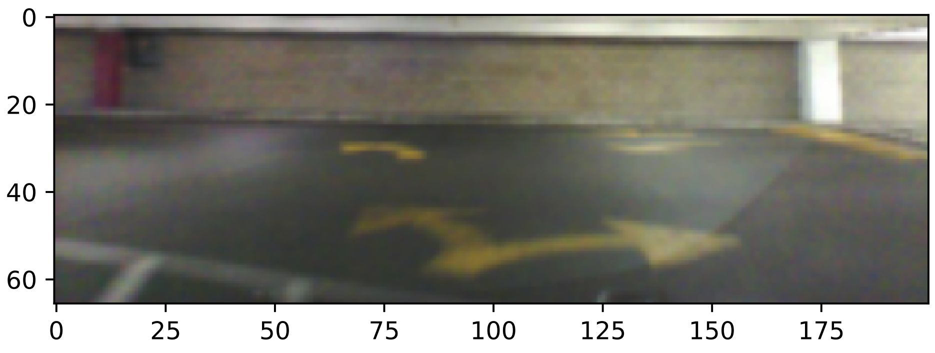
**(c)** $100^{th}$ frame in the PilotNet Dataset [34]

**Figure 2.12:** $1^{st}$, $50^{th}$ and $100^{th}$ frame in the PilotNet Dataset [33]

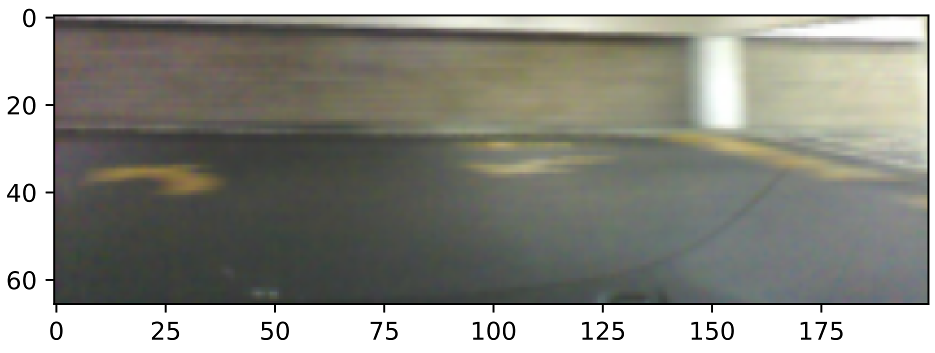
Figure 2.13 represents a right turn captured from a front-facing camera.



**(a)** $250^{st}$ frame in the PilotNet Dataset [34]



**(b)** $260^{th}$ frame in the PilotNet Dataset [34]



**(c)** $270^{th}$ frame in the PilotNet Dataset [34]

**Figure 2.13:** $250^{th}$, $260^{th}$ and $270^{th}$ frame in the PilotNet Dataset [33]

The validation set was executed in a TensorFlow framework and the predicted steering angles for all images were extracted from the 1 / r values using the following equation Equation 2.51 and compared with the TensorFlow keras output in Figure 2.14.

$$angle = \frac{1}{r} \times \frac{180}{\pi} \times 2 \tag{2.51}$$

In Figure 2.14, the positive value of the steering angle represents that the car is turning right, and the predicted negative value means that the car is turning left. The frames of Figure 2.13 are correlated with the positive steering angle of Figure 2.14.

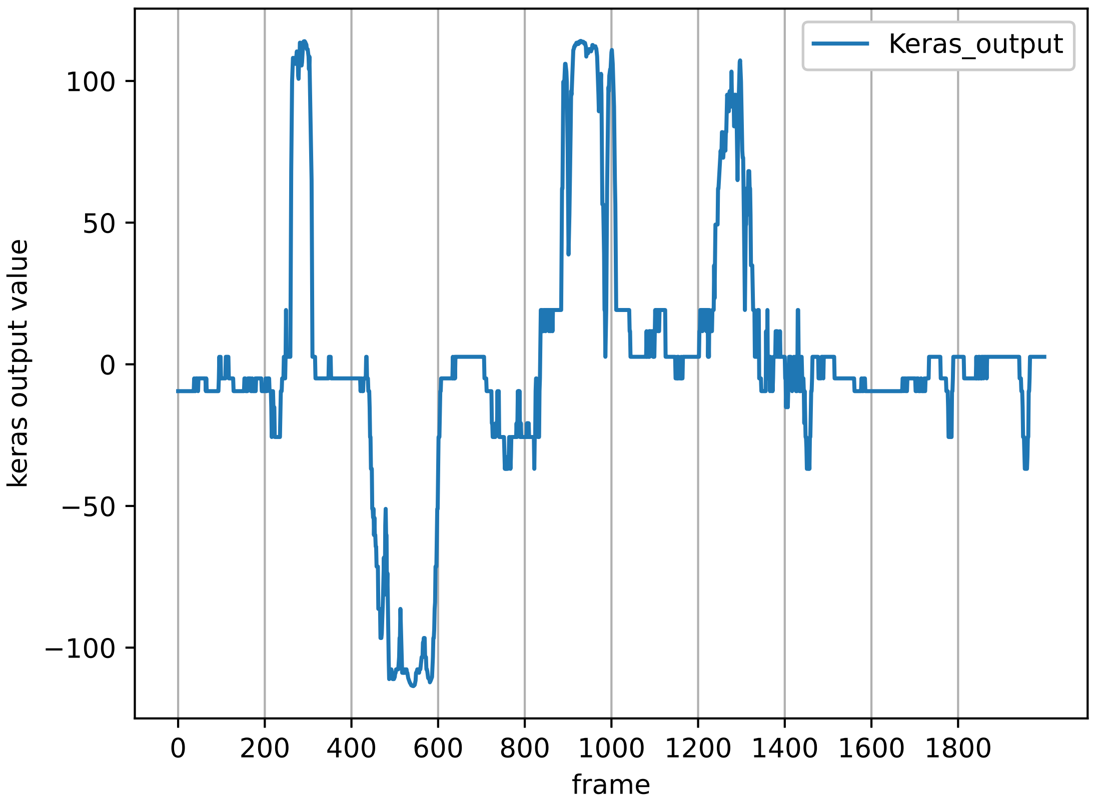


**Figure 2.14:** Keras output representing the steering angle in degrees [34]

### 2.4.2. MNIST- 4 layered DNN application

Mapping was also tested with an 4 layered DNN trained with MNIST. Compared to SNN PilotNet, 4 layered DNN MNIST has a relatively smaller number of layers and fewer neurons per layer, but it has been thoroughly evaluated, so MNIST DNN was also used for experimentation purposes. The architecture of the DNN used for the experiments can be Figure 2.15. The DNN output trained with MNIST dataset was captured for a set of input test images labeled **[7,2,1,0,4,1,4,9,5]** and processed with SENSIM, and the membrane potential was captured and analyzed as shown in figure Figure 2.16. The highest membrane potential that gives the output signal detected on the output will be the number detected at the output potential. The rate of the system and the data set input for the system can be controlled.

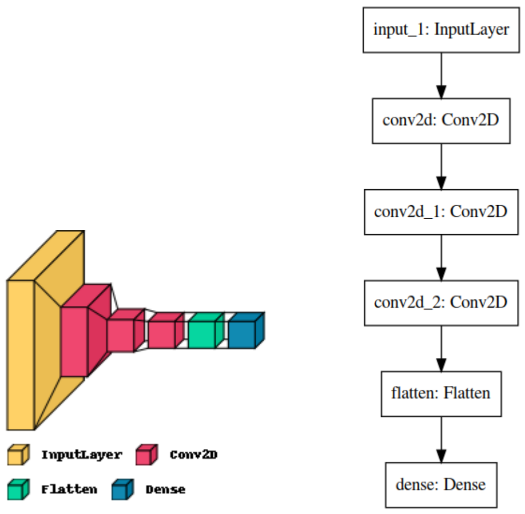


**Figure 2.15:** 4 layered DNN model trained with MNIST

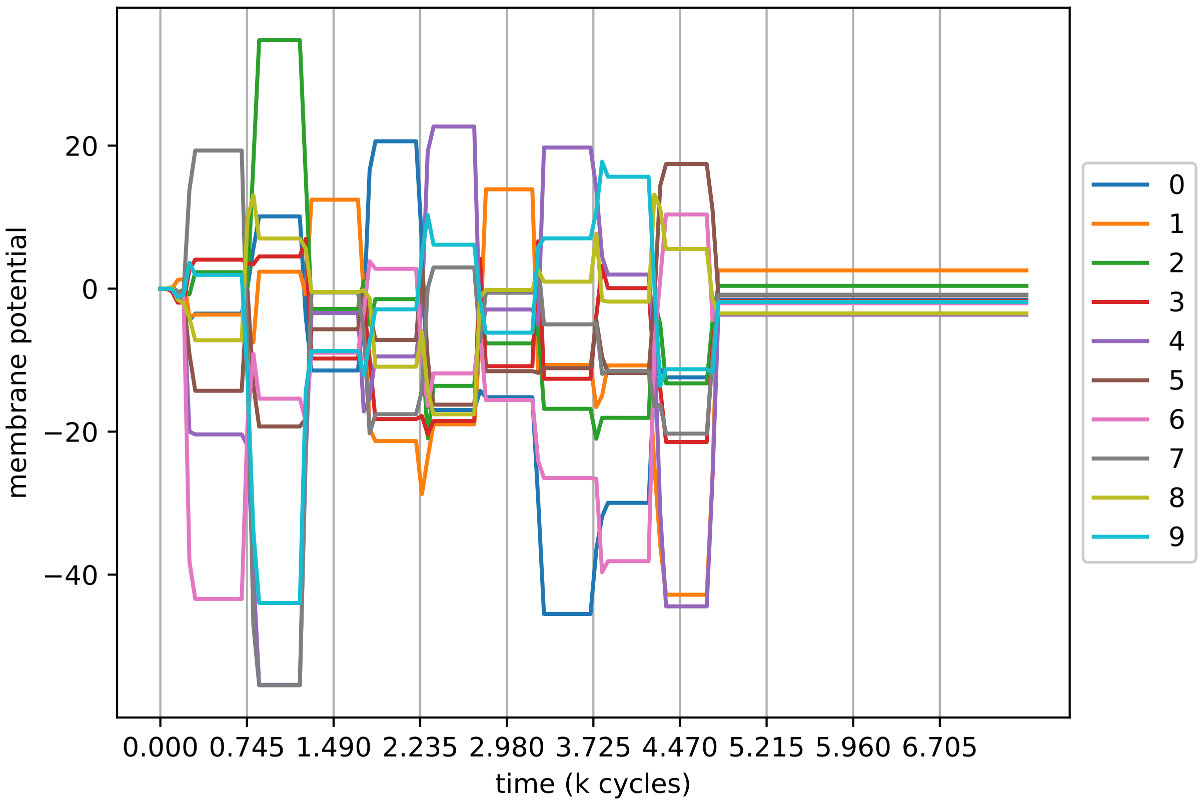


**Figure 2.16:** Output of the membrane potential of MNIST for series of images processed by SENSIM labeled as [7,2,1,0,4,1,4,9,5]

For the purpose of experimentation, the MNIST database [35] was used, which is a database of handwritten digits that has a training set of 60,000 examples and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. MNIST is ideal for trying learning techniques and pattern recognition methods on real-world data while spending minimal effort on preprocessing and formatting.

# 3

# Algorithmic research

This chapter will cover topics that provide a brief understanding of how neurmorphic SDKs are developed by other research communities. Section 3.2 briefs some common optimization strategies used by the research community to solve the efficient mapping problem. Followed by Section 3.2 giving a brief explanation of algorithms developed to solve optimization problems. Section 3.2.2 covers some important aspects that need to be considered when forming the optimization problem. Section 3.3 covers the brief exploration of the optimization tools and frameworks that could be integrated with the partition and clustering framework.

## 3.1. Software development frameworks and mapping algorithms

Lava is an open-source software framework, initially developed by Intel under the Lava Project **github.com/lava-nc**; an initiative of the Intel neuromorphic team for the development of neuromorphic computing. The framework emerges as a common framework to support the execution of applications on neuromorphic hardware systems, often involving multiple physical computing elements, ranging from special-purpose neural accelerators to conventional CPU/GPUs, sensors, or actuator devices. Lava Frameowrk tries to mimic this general, massively parallel, heterogeneous architecture. All applications built in lava are independent of the compute engine that runs underneath, and the messages are communicated via generalized message types. In the Lava framework, libraries such as the process library and application libraries are covered under the BSD license, the Magma framework, which contains the runtime, compiler, and lower-level libraries, is majorly proprietary, and some of the modules are currently under development. Although the compiler is proprietary where most likely the mapping algorithms are implemented, Intel details its implementation more in [36] for crossbar-based architectures. Since Loihi has a crossbar-based architecture, a mapping technique designed for SENeCA could not be used.

Carlsim [37] is a GPU-accelerated simulator for training and testing SNN-based applications. The simulator provides a PyNN-like programming interface in C / C++, which allows details and parameters to be specified at the synapse, neuron, and network level. CARLsim reports spike times for every synapse in the SNN. Although Carlsim provides a fair amount of detail in describing biological SNNs, the interface can only be programmed in C++ which does not prove to be programmer-friendly for SNN/ DNN programmers. Hence, later PyCARL was developed to integrate PyNN with Carlsim. The latest version of Carlsim can be found at `https://github.com/UCI-CARL/CARLsim6`.

Noxim++ [38] is a trace-driven and cycle-accurate interconnect simulator for multiprocessor systems. Noxim++ extracts parameters such as latency, throughput, power, ISI distortion, and disorder spike count. NOXIM also gives the complete spike trace for a detailed analysis of the spikes in the time frame. Noxim resembles the SENSIM framework to some extent, but SENSIM does not give a detailed trace of the spikes as spikes are packaged into events and delivered from one core. Inter-Spike Distortion or in the case of SENSIM Inter event/Spike Distortion cannot be analyzed in SENSIM which might pose a drawback in a detailed inspection of spikes and events/flits in the case of SENSIM but end signal's accuracy can be evaluated via correlation.

PyCARL [39]is a Python programming interface based on PyNN for hardware-software co-simulation of spiking neural networks (SNN). PyCARL integrates PyNN to CARLsim, a computationally efficient, GPU-accelerated, and biophysically detailed SNN simulator. PyCARL facilitates the joint development of machine learning models and code sharing between CARLsim and PyNN users, promoting a larger integrated neuromorphic community. PyCARL also integrates cycle-accurate models of state-of-the-art neuromorphic hardware, such as TrueNorth [24], Loihi [4], and DynapSE [40], to accurately model hardware latencies, which delay spikes between communicating neurons, degrading the performance of machine learning models. PyCARL allows users to analyze and optimize the performance difference between software-based simulation and hardware-oriented simulation. PyCARL performs design space exploration early in the product development stage, facilitating a faster time to market for neuromorphic products. PyCARL integrates hardware into the loop using NOXIM and extracts metrics such as latency, throughput, power, ISI (interspike interval) distortion, and disorder spike count. PyCARL resembles SENSIM to a good extent, except that it does not have a cycle-accurate trace-driven simulator, but the timing in SENSIM is estimated by packet time stamping.

SpiNeMap [38] consists of SpiNeCluster to partition SNNs into groups to minimize the total number of spikes for all of our evaluated applications in the shared interface and SpiNePlacer to optimize cluster placement on neuromorphic hardware crossbars to minimize energy consumption and latency in the shared interface. SpinePlacer uses Particle Swarm Optimization for placing the neurons on the crossbar and SpineCluster uses a greedy based approach to reduce the hops. Since the default setting in SENSIM uses multi-cast as the default communication method and the architecture of SENeCA is a NOC based architecture with default communication being multicast and not unicast. Estimation of hops and having it in the optimizaion strategy will be different for SENeCA as compared to SpiNeMap. While ideas such as calculation of interspike distortion and minimizing the number of hops to reduce the energy can be considered in the optimization strategy.

SNEAP [41] is a toolchain for mapping large-scale SNNs to neuromorphic hardware. The authors suggest a graph partitioning scheme that partitions the graph according to a multilevel graph partitioning scheme [42], followed by reducing energy by reducing the average hop count. SNEAP is a tool that can be used to map crossbar-based neuromorphic architectures that cannot be used directly with SENSIM because the SENeCA uses a generic memory node and maps a cumulative set of neurons and does not consider a single neuron and maps it. In [41] energy reduction is achieved by reducing the average hop count in the NoC for communication reduction, which also varies in SENSIM and SENeCA as the default communication is multicast. The optimization strategies used to minimize hop are particle swarm optimization, tabu search, and simultaneous annealing, which are commonly used black-box metaheuristics.

DNN mapping in Spinnaker2 [43] suggests that are partitioning to be done channel-wise followed by height-wise and width-wise and since SENeCA is similar to Spinnaker2 the partitioning strategy used in the given paper would also be the same.

The paper [44] suggests online learning approach with greedy based approach for clustering and hill climbing based approach placing the neurons. However, in the case of online learning, when the model is re-trained for better performance, the connections within the SNN are expected to change as new events are learned, and the weights of existing synaptic connections may undergo changes after every learning epoch. To ensure optimal hardware performance at all times, a run-time approach is required that remaps the SNN to the hardware after every learning epoch. The technique reduces the spikes communicated on the time-multiplexed interconnect, therefore, reducing energy consumption. For run-time mapping, the authors suggest a greedy approach [45] to partition the network followed by a hill-climbing-based heuristic [46] method that performs multiple local searches and moves to the optimum based on a custom cost function. [44] cannot be used directly with SENeCA, as the mapper is based on crossbar-type architectures with a fixed size of synaptic and neural memory. The paper only tests the algorithm on small-scale networks such as MLP-MNIST, and synthetic applications with 2,00,000 synapses, which are comparatively small- or medium-scale neural networks compared to PilotNet and large-scale spiking neural network applications such as RESNET. SENeCA does not transmit a spike, but rather is packed as events, and these events are multi-casted; hence the methods suggested in the paper cannot be directly applied.

## 3.2. Optimization strategies

All mapping tools have mainly used metaheuristics or heuristics to obtain neuronal mappings that minimize energy, latency, the distance between spikes, and or a combination of these in a cost function as the objectives are achieved. In the case of SENSIM the end result remains energy optimization hence similar optimization algorithms but with different objectives could be used.

Heuristics are problem-dependent techniques that adapt to the problem, whereas metaheuristics, on the other hand, are problem-independent techniques. Meta-heuristics do not take advantage of the problem, are not greedy, and instead look into the problem more thoroughly. In fact, they may even accept a temporary deterioration of the solution, which allows them to explore the solution space more thoroughly and thus get a better solution. Although a metaheuristic, is a problem-independent technique, it is nevertheless necessary to do some fine-tuning of its intrinsic parameters in order to adapt the technique to the problem at hand.

Metaheuristics inspired by nature and based on metaphors is a very active area of investigation because of its promising results in global optimization problems. Meta-heuristics are classified into algorithms simulated annealing, evolution strategy, ant colony optimization, and particle swarm optimization. SENSIM that emulates SENeCA is a system based on several parameters, and architectural bottlenecks, hence metaheuristics is the most ideal algorithms to find better mapping solutions.

### 3.2.1. Meta-heuristics

**Simulated annealing**

Simulated annealing [47]is a probabilistic method to approximate the total optimality of a given function. To briefly explain the workings of the algorithm, the simulated annealing heuristic takes into account some neighboring states s* of the current state, and taking into account some neighboring states s*, probability determines whether the system moves to s* or stays in state s at each step. Typically, this step is repeated until the system reaches a suitable state for the application or until a given computational budget is exhausted.

**Evolutionary algorithms optimization strategies**

Evolutionary algorithm [48] is an evolution-inspired algorithm that uses biological evolution-inspired phenomena such as reproduction, mutation, recombination, and selection to achieve an optimal solution. Candidate solutions for optimization problems play the role of individuals within the population, the function of fitness determines the quality of the solution. The evolution of the population then takes place after the repeated application of the above-mentioned operators. There are several implementations of evolutionary algorithms, such as [49] [50].

**Swarm intelligence optimization strategies**

Swarm intelligence (SI) [51] is the collective behavior of self-organized, decentralized systems, natural or artificial. The concept is employed in work on artificial intelligence. These properties make swarm intelligence make a successful design paradigm for algorithms that deal with increasingly complex problems. The most successful examples of optimization techniques inspired by swarm intelligence are ant colony optimization and particle swarm optimization.

In Ant Colony Optimization (ACO) [52], a set of software agents called artificial ants search for good solutions to a given optimization problem. To apply ACO, the optimization problem is transformed into the problem of finding the best path on a weighted graph. Artificial ants build solutions incrementally by moving on the graph. The solution construction process is stochastic and is biased by a pheromone model, that is, a set of parameters associated with graph components (either nodes or edges) whose values are modified at run-time by the ants.

Particle swarm optimization (PSO) [53] is a computational method that iteratively optimizes a problem by having a population of candidate solutions, here called particles. These particles move around in the search space according to a mathematical formula that updates the particle's position and velocity. Each particle's movements are influenced by its local best-known position, but are also guided towards best-known locations in the search space that are updated as other particles find better positions. However, PSO does not guarantee that an optimal solution will ever be found.

**Tabu search**
Tabu search [54] is a metaheuristic search method that employs local search methods used for optimization. Tabu Search improves the performance of local searches by relaxing the underlying rule that bad moves can be accepted at every step if no better move is available (for example, when the search is stuck at a strict local minimum), which increases the efficiency of the local search. In addition, prohibitions (hence the name Tabu) are implemented to discourage searching to return to previously visited solutions.

### 3.2.2. Heuristics and graph-based optimization strategies

There are few other heuritics approaches which possibly can be used for mapping cited below which were not experimented with in the thesis. [42] Multi-way partitioning for irregular graphs, [55] KL graph partitioning algorithm, [46] Hill climbing, [45] K way graph partitioning for graphs are some of the graph-based heuristics which are also used by several authors.

**Defining an optimization problem [56]**
Ideally, any optimization can be described by a set of generic mathematical equations as described in Equation 3.1 [56]

$$\begin{aligned} \text{min} \quad & f_i(x) & i = 1, 2, ..I \\ \text{s.t.} \quad & g_j(x) \leq 0 & j = 1, .., J \\ & h_k(x) = 0 & k = 1, .., K \\ & xL_m \leq x_m \leq xU_m & m = 1, .., M \\ & x \in \Omega \end{aligned} \tag{3.1}$$

where $x_m$ represents the *m-th* variable to be optimized, $xL_m$ and $xU_m$ its lower and upper bound, $f_i$ the $i-th$ objective function, $g_j$ the $j{-}th$ inequality constraint and $h_k$ the $k-th$ equality constraint. The objective function(s) f is supposed to be minimized by satisfying all the constraints of equality and inequality. If a specific objective function has to be maximized (*max* $f_i$), a maximization problem can be redefined to minimize its negative value (*min* $-f_i$). As a first step in solving an optimization problem, it is very important to think about the formulation of the mathematical problem. Performing so beforehand helps to identify the challenging aspects of the optimization problem and thus helps to select suitable optimization algorithms. The following section gives more details on the aspects to consider when evaluating an optimization problem.

**Aspects of evaluating an optimization problem [56]**

- **Variable Types** Different variable types, such as continuous, discrete/integer, binary, or permutation, define the characteristics of the search space. The variable type decides the solution space search time; hence, this makes the variable type an important parameter when defining the problem.
- **Number of Variables** Apart from the type of variables, the number of variables also plays an important role in deciding the search time in the search space. The choice of the algorithm also differs when solving a problem with 10 variables from when solving a problem with 1000 variables. Algorithms that require derivatives during solution computation time might increase exponentially with an increasing number of variables.
- **Number of Objectives** Some optimization problems have more than one conflicting optimization objective. To solve problems with more than one objective, the set of algorithms differs, where algorithms try to generalize the solution dominance relation by comparing the scalars in single-objective optimization. When the objective space has more than one dimension, there can be cases where the optimum lies in the non-dominant solution set; hence, the objectives play an important role while defining the problem.
- **Constraints** Optimization problems have inequality and equality constraints. It does not matter how good a solution is if it is considered infeasible if it violates only one constraint. Constraints can have a large impact on the complexity of the problem if the feasible solution is made up of small islands in the feasible search space due to a large number of constraints. Having a genetic algorithm to solve a problem with equality constraints can be a rather challenging task, which can be addressed by mapping the search space to a utility space where the equality constraints are always satisfied.

- **Multi-modality** In the case of multimodal fitness landscapes, optimization becomes more difficult due to the existence of local optima. Any solution found in the case of multimodal functions must have explored enough regions in the search space to maximize the probability of obtaining the global optimum. In case a local search method is used in a multi-model space, might result in a solution that easily gets stuck in local minima.
- **Differentiability** If a function is differentiable, gradient-based optimization methods can be used, which can be a great advantage over gradient-free methods. The gradient provides a good indication of what direction to use for the search. Most gradient-based algorithms are point-by-point-based and can be highly efficient for rather unimodal fitness landscapes. However, in practice, functions are often not differentiable, or a more complicated function requires a global search instead of a local search.
- **Evaluation Time** Optimization problems can consist of domain-specific software and can be lengthy, so the evaluation time of the problem must be considered when forming the optimization problem. In the case of third-party software, this often results in a computationally expensive and time-consuming function to assess objectives or constraints. For these types of problems, the overhead of the algorithm to determine the next solutions to be evaluated is often not recognizable.
- **Uncertainty** Objectives and constraints can be deterministic or non-deterministic. Noise can be introduced by multiple target functions, which can add uncertainty to the system. Optimization problems with underlying uncertainty are investigated in a research field and are called stochastic optimization. **Note**: From experimentation, there was minimal difference in energy estimations on executing the program several times; hence, the stochastic optimization and uncertainty were not considered in the scope of the project.

## 3.3. Optimization tools exploration

To solve any optimization problem compatible with the preexisting Python framework, a brief exploration of the tools that support the above algorithm was essential. Hence, the following are some tools explored, and from the following Pymoo (Python multi-objective optimization library) seems to be the most appropriate one, which could be integrated with SENeCA's SDK.

- **Genetic Algorithm Python Library** An open-source Python library that provides a basic implementation of the Genetic algorithm. The library provides a decent implementation, but very little modularity compared to other tools.
- **Google Or-tools** Google OR-Tools [57]is a free open-source software suite developed by Google to solve linear programming (LP), mixed integer programming (MIP), constraint programming (CP), vehicle routing (VRP) and related optimization problems. The tool being open source, does not give enough flexibility to optimize functional problems which cannot be modeled as mathematical equations, and its lack of support for nonlinear optimization problem solvers poses a big disadvantage for the library to be used for research work.
- **Scipy** Scipy [58] optimizes and minimizes Python libraries, providing a decent optimization frame to minimize the objective function subject to constraints. Scipy supports non-linear problems and global optimization algorithms along with linear, non-linear least squares, and root fitting, which makes it a decent choice as a tool for the implementation in the project. Support for algorithms like Nelder-Meed, Powell, Conjugate Gradient, BFGS, etc. which were some conventional optimization algorithms. A few disadvantages the library has is lack of support for multi- and many-objective optimization problem solvers, for which the library did not seem to be the most apt choice for the project.
- **jMetalPy** [59] An open source multi-objective optimization framework jMetal developed in Java ported to a Python version, namely jMetalPy. In addition to traditional optimization algorithms, jMetalPy also offers methods for dynamic optimization and supports various algorithms, including GA and simulated annealing. The tool also supports distributed frameworks like Spark, Dask, etc. The set of features and support for a wide variety of algorithms made this library an appropriate choice for the project.
- **PyGMO** PyGMO Parallel Global Multi-Objective Optimizer [60]is an optimization library for the easy distribution of massive optimization tasks on multiple CPUs. It uses the generalized island model paradigm for the coarse-grained parallelization of optimization algorithms and, therefore, allows users to develop asynchronous and distributed algorithms. PyGMO supports various basic local optimization algorithms and metaheuristic-based global optimization, which might prove to be an advantage where both local and global optimization problem solvers are needed.

- **Metaheuristics.jl** [61] is a high-performance metaheuristics for global optimization programmed in Julia that supports both single- and multi-objective global optimization algorithms such as PSO, GA, MCCGA, WOA, SA, GSA etc.
- **Open Beagle** Open BEAGLE [62] is a C++ Evolutionary Computation framework that provides a high-level software environment with support for tree-based genetic programming; bit string, integer and real-valued genetic algorithms; and evolution strategy.
- **Opt4J** Opt4J [63], Evolvica [**empty citation**] are open source Java-based frameworks for evolutionary computation. It contains a set of (multiobjective) optimization algorithms, such as evolutionary algorithms, differential evolution, particle swarm optimization, and simulated annealing.
- **JGAP** GAP (pronounced "jay-gap") is a genetic algorithm and a genetic programming component provided as a Java framework. Provides basic genetic mechanisms that can be easily used to apply evolutionary principles to problem solutions. See the examples for a demonstration or look at the graphical tree that can be created with JGAP for found solutions of genetically evolved programs.

Compared to several other optimization frameworks available and discussed above, Pymoo [56] was the library most widely used and accepted by the community.

### 3.3.1. Pymoo - Python Multi Objective Optimization

Pymoo [56] is an open source framework made to solve single, multiple, and many-objective optimization problems. The framework also comes with impressive analytical visualization and post-optimization decision-making tools. Since Python is one of the programming languages that the research community uses most frequently to analyze, improve, and research big data frameworks, its ease of integration with Python frameworks makes it the most appropriate optimization tool for our project. While implementing optimization algorithms can be time-consuming and prone to bugs, integrating several Python optimization libraries implemented by the research community can also be cumbersome. Having a comprehensive, well-tested, and widely accepted library can save time and avoid error-prone implementations.

Apart from having a huge set of algorithms from single multi- and many-optimization problems, the state-of-the-art framework also provides easy integration with multithreaded frameworks implemented in Python and recently quite popular distributed computing frameworks in Python, which helps improve the compute time of an optimization problem.

Compared to other open-source tools, Pymoo also provides in-depth information on optimization mathematics, such as gradients, for further development. Pymoo also comes with a set of optimization algorithm performance indicators that can be used to understand how well the algorithm performed for a given problem. Pymoo does not just give a basic naive implementation of optimization algorithms, but rather gives very functional plug-and-play implementations of submodules for parameters of an algorithm. This feature allows developers with domain knowledge to implement a customized version of the algorithms.

Apart from being open source, the framework is well documented with a starter guide and examples, which make it easy for beginners to get acquainted with the framework. Figure 3.1 gives a brief description of the Pymoo software architecture, suggesting that the framework can be classified principally as problem-solving, optimization, and analytics.

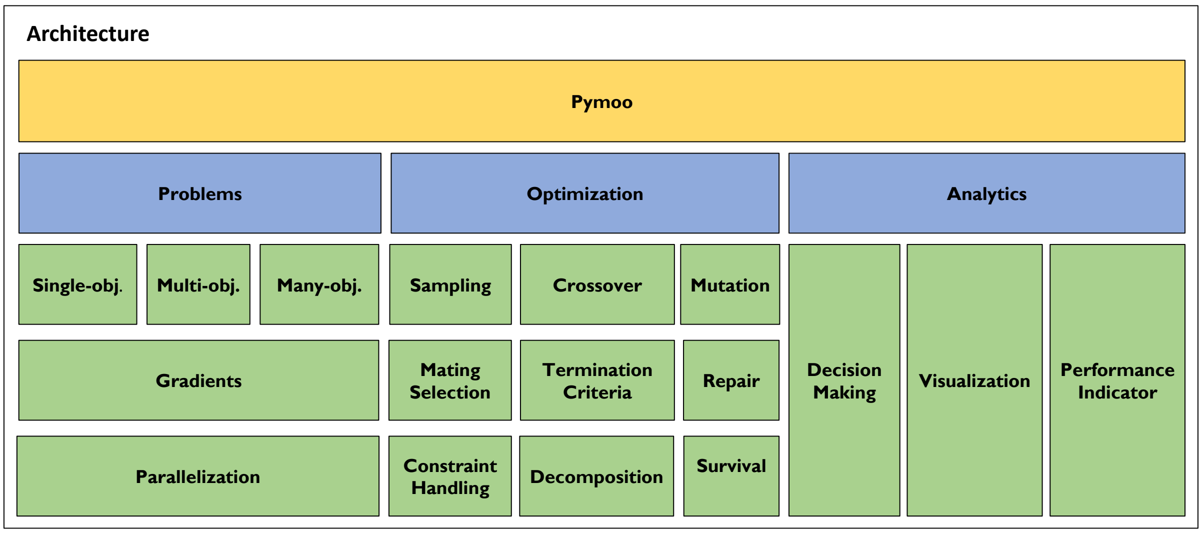


**Figure 3.1:** Pymoo software architecture [56]

To minimize or optimize a problem in Pymoo, the problem needs to be defined first. Pymoo's functional problem definition takes into account the parameters mentioned in the text.

**Pymoo - Problem definition**
Based on Equation 3.1, the parameters used by Pymoo are described in Table 3.1.

| Parameter | Description |
|---|---|
| n_var | Integer value representing the number of design variables. |
| n_obj | Integer value representing the number of objectives |
| n_constr | Integer value representing the number of constraints |
| xl | An array of length n_var representing the lower bounds of the design variables |
| xu | An array of length n_var representing the upper bounds of the design variables |
| type_var | *(optional)* A type hint for the user what variable should be optimized |

**Table 3.1:** Pymoo Problem definition Parameters

Based on how problems can be evaluated, Pymoo gives its users the option to choose from the following set of problem definitions.

- **Basic Problem definition** a set of solutions is evaluated at once.
- **Element wise Problem definition** single solution is evaluated at once, hence elementwise.
- **Functional Problem definition** A complete function is evaluated at once.

The definitions of the problem are such that **Functional Problem definition** is inherited from **Element-wise Problem definition**, which is further inherited from **Basic Problem definition** and gives high flexibility and usability to all its users. Pymoo's feature of forming an objective function to be evaluated as a function, or rather passing a function to minimize in the optimization problem, is one of the reasons which made it an apt choice in the design of the mapper.

Pymoo framework comes with a suite of the following algorithms for optimization to solve single multi-objective and many-objective problems.

| Algorithm | Description |
|---|---|
| Genetic Algorithm | Pymoo incorporates a modular implementation of genetic algorithm. The implementation of a genetic algorithm comes with the various operators like mutation, crossover, and selection. |
| Differential Evolution | Different variants of differential evolution which is a well-known concept for in continuous optimization especially for global optimization [64] |
| BRKGA | Baised Randomn key generation algorithm is a combinatorial optimization where instead of custom evolutionary operators the complexity is put into an advanced variable encoding. [49] |
| Nelder Mead | A point-by-point based algorithm which keeps track of a simplex with is either extended reflected or shrunk. [65] |
| Pattern Search | Pattern search is an iterative black box search based optimization method which [66] |
| CMAES | Well-known model-based algorithm sampling from a dynamically updated normal distribution in each iteration. [67] |
| ES | Evolutionary strategy algorithm proposed for real-valued optimization problems. [56] |
| SRES | SRES is an single objective evolutionary strategy which works well with constrained optimization problems [68] |
| ISRES | ISRES is an single objective improved version of SRES can deal with dependent variables efficiently. [69] |
| NSGA-II | NSGA-II multi-objective optimization algorithm based on non-dominated sorting and crowding. [50] |
| R-NSGA-II | An extension of NSGA-II where reference/aspiration points can be provided by the user. [70] |
| NSGA-III | An improvement of NSGA-II developed for multi-objective optimization problems with more than two objectives. [71] |
| U-NSGA-III | U-NSGA-III is a generalized version of NSGA-III to be more efficient for single and bi-objective optimization problems. [72] |
| R-NSGA-III | R-NSGA-III is a reference point based enhancement to NSGA-III which allows defining aspiration points for NSGA-III to incorporate the user's preference. The algorithm supports multi objective optimization [71] |
| MOEAD | MOEAD is a well-known multi-objective optimization algorithm which decomposes a multi-objective optimization problem into many scalar optimization sub-problems and optimizes them simultaneously. All sub-problems are optimized by using information from its several neighboring sub-problems, which makes MOEAD have lower computational complexity after each generation. [73] |
| AGE-MOEA | AGE-MOEA is similar to NSGA-II [50] but it estimates the shape of the Pareto-front to compute the score replacing the crowding distance. The non-dominated fronts are sorted using the non-dominated sorting procedure. Then the first front is used for normalization of the objective space and estimation of pareto front geometry. [74] |
| C-TAEA | A multi-objective algorithm which supports two archive constraint-handling by collaboration of convergence (Convergence Archive CA) which helps in converging to the optima and diversity (Diversity Archive DA) which helps in creating a variation in the solutions. [75] |

**Table 3.2:** Pymoo Optimization Algorithm Suit

Apart from having a huge variety of algorithms at our disposal, it provides options to its users to choose what type of variables it will operate on. Although Pymoo was developed for continuous-variable problems, it supports other variable types as well. Table 3.3 details the type of supported variable.

| Algorithm | Description |
|---|---|
| Binary Variable Problem | Supports a random search where the set of 0's and 1's |
| Discrete Variable Problem | Supports a random set of discrete integer values |
| Permutations | Support a random set of integer values where the integer is not repeated in the set |
| Mixed Variable Problem | Supports a mix of variables where some variable are integer while some are real valued variables. |
| Custom Variable Type | Supports set of objective functions to form problems with different variable types and variable length sets. |
| Biased Initialization | Supports a biased initialization of a problem by giving it a value or set of values which are to help the optimizer to come to a minimum is fewer iterations. |
| Subset Selection Problem | A special type of problem where the input variable type can be operated on, and the variable type is converted to another variable type to make the optimization simpler. |

**Table 3.3:** Variable type problems supported by Pymoo [56]

Since the number of cores, the number of neurons, and most of the parameters in SENSIM are discrete values, the discrete value support with Pymoo makes the tool extremely user-friendly. The CMR (Cache Miss Ratio) mentioned in Table 2.3 can take a mixed precision value which is also supported by Pymoo with Mixed Varaible problem formation. To add more to the above features, Pymoo also provides a visualization toolkit, which includes scattering plots, parallel coordinate plots, heat maps, etc.

## 3.4. Conclusion

It can be induced from the mathematical model of the neuromorphic core, and the architecture described in Section 2.3.1 that the scaled system is nonlinear. This nonlinearity is caused by functions like **max** as per 2.14 $T = max(T_{ext}, T_{compute})$ used to compare data copy time are stored at different levels of memory, as well as due to bottlenecks like queuing in the flexible architecture of SENeCA. As a result, the mapper was designed to match the flexible dynamic system, hence meta-heuristics (as detailed) seemed to be a better approach, as it has a better chance of approaching a global optimum. To scale up the design of changing applications (DNN and SNN), meta-heuristics were chosen over defined heuristics as meta-heuristics are problem independent, which gives more flexibility towards forming the problem. Since there could be more than 1 objective such as energy, latency, and accuracy, hence multi-objective optimization algorithms such as NSGA-2 were used. The choice of GA and NSGA2 was made based on factors such reliability of the implementation in Pymoo. Several papers compare different metaheuristics, but according to the no free lunch theorem, a universal strategy for optimization does not exist [76]. The only way one strategy can outperform another is if it is specialized to the structure of the specific problem under consideration. Since the problem is changing based on the changing architecture of SENeCA and different applications, the choice of algorithms was kept to GA and NSGA2 for all the experiments performed in Chapter 5.

# 4 SENMap: Design and implementation

This chapter will cover the step-by-step implementation of SENMap (the hardware mapper). Section 4.1 gives details on how neurons from an SNN are assigned to the neuromorphic cores of SENSIM. The mapping is rather direct and is not optimized in terms of memory, energy, latency, etc. Section 4.2 briefs about the analytics framework developed to understand how different mapping schemes affect the intrinsic parameters of the neuromorphic processor. Section 4.3 elaborates upon several utilities developed for meta-data extraction, neuron partitioning, mapping compression, clustering, and partitioning used to extract neuronal division before mapping and optimization. Section 4.4 covers various aspects to focus on before modeling the optimization problem and also illustrates some optimization problems. Chapter 4 is concluded by Section 4.7 evaluating the SENeCA SDK with SENMap as a whole.

## 4.1. Direct mapping

The definition of the application as stated in Section 2.3.6 consists of the following stages.

- Initialize the application with simulator parameters
- Compose the layers
- Compose layer core mapping
- Setup interconnect

Although the initialization of the application has basic steps to set up the application parameters from Table 2.11, it is followed by compiling the layers of the application with the parameters mentioned in Table 2.5. Without these basic steps, mapping of neurons from a layer is not possible, as the composition of the layer decides the type of operation the simulator will perform based on the parameters. Code Snippet 4.2 gives an example of the most basic PilotNet mapping in the SENSIM application.

In the mapping, a composed layer in PilotNet is assigned to a core in the simulator. The mapping is made up of **layer_ID** which is to be mapped. The range of neurons is expressed as **[range(0,66), range(0,200), range(0,3)]** for a convolution layer, where **range(0,66)** represents the height dimension in an input image with **[height, width, channel]**. Similarly, **range(0,200)** represents the width and **range(0,3)**, representing the channel in the example.

Due to the limitation of SENSIM's execution model, mapping more than one layer to a particular core is not possible, but one layer can be divided and mapped to more cores. The input and output layers labeled **I** and **O**, respectively, in Code Snippet 4.2 are not assigned to the physical cores, but to the virtual cores that are not part of the simulation.

For all other layers, a package of **[core, range of neurons, CMR[neurons, weights]]** is used to map a particular range of neurons from a layer to a core. For example, the first layer of Code Snippet 4.2, **(coresList[0][0],**

**[range(0,31), range(0,98), range(0,24)], [0.0,0.0])** maps **[range(0, 31), range(0, 98), range(0,24)]** to the core located at the location **[0][0]** from the complete corelist derived from the mesh parameter, which could possibly be a physical core in a simulation. The coordinates of the core, in the corelist, resemble the location of the core. For example, a core located at **[4,3,0]** will be in the corelist at **[4][3]**. The third content of the package is the CMR (cache Miss Ratio), which is the number between 0 and 1 and gives the ratio/percentage of weights and neurons that will be stored in external memory.

Due to the limitation of the SENSIM execution model, where more than one layer cannot be mapped to a particular core, it does pose a slight limitation in getting the most optimized mapped for energy, latency, and area when it comes to mapping large-scale neural networks, the expectation is that these networks are quite densely filled with neurons; hence, an assumption was taken that the most optimized energy and latency would not fall in a mapping with two layers mapped to a single core. The change in the execution model to support 2 layers per core will be discussed in future work Section 6.2

```


self.layers['O']  = layer(layer_type='output', name='output')

self.layers['L10']= layer(layer_type='dense', neuron_type='SigmaDelta', threshold=np.ones([
     1])*0,  act_fun='Tanh',  shape=[   1], name='dns5', weight_tensor=self.weights[9][0],
    bias_tensor=self.weights[9][1], outputLayer=(self.layers['O'],))

self.layers['L9'] = layer(layer_type='dense', neuron_type='SigmaDelta', threshold=np.ones([
    10])*self.thresholds[9],  act_fun='ReLU',  shape=[  10], name='dns4', weight_tensor=self.
    weights[8][0], bias_tensor=self.weights[8][1], outputLayer=(self.layers['L10'],))

self.layers['L8'] = layer(layer_type='dense', neuron_type='SigmaDelta', threshold=np.ones([
    50])*self.thresholds[8],  act_fun='ReLU',  shape=[  50], name='dns3', weight_tensor=self.
    weights[7][0], bias_tensor=self.weights[7][1], outputLayer=(self.layers['L9'],))

self.layers['L7'] = layer(layer_type='dense', neuron_type='SigmaDelta', threshold=np.ones([
    100])*self.thresholds[7],  act_fun='ReLU',  shape=[ 100], name='dns2', weight_tensor=self
    .weights[6][0], bias_tensor=self.weights[6][1], outputLayer=(self.layers['L8'],))

self.layers['L6'] = layer(layer_type='dense', neuron_type='SigmaDelta', threshold=np.ones
    ([1164])*self.thresholds[6],  act_fun='ReLU',  shape=[1164], name='dns1', weight_tensor=
    self.weights[5][0], bias_tensor=self.weights[5][1], outputLayer=(self.layers['L7'],))

self.layers['L5'] = layer(layer_type='conv', pooling='stride', pooling_size=[1,1], padding='
    valid', is_flatten=True,  neuron_type='SigmaDelta', threshold=np.ones([64])*self.
    thresholds[5],  act_fun='ReLU',  shape=[ 1,18,64], name='cnv5', weight_tensor=self.
    weights[4][0], bias_tensor=self.weights[4][1], outputLayer=(self.layers['L6'],))

self.layers['L4'] = layer(layer_type='conv', pooling='stride', pooling_size=[1,1], padding='
    valid', is_flatten=False, neuron_type='SigmaDelta', threshold=np.ones([64])*self.
    thresholds[4],  act_fun='ReLU',  shape=[ 3,20,64], name='cnv4', weight_tensor=self.
    weights[3][0], bias_tensor=self.weights[3][1], outputLayer=(self.layers['L5'],))

self.layers['L3'] = layer(layer_type='conv', pooling='stride', pooling_size=[2,2], padding='
    valid', is_flatten=False, neuron_type='SigmaDelta', threshold=np.ones([48])*self.
    thresholds[3],  act_fun='ReLU',  shape=[ 5,22,48], name='cnv3', weight_tensor=self.
    weights[2][0], bias_tensor=self.weights[2][1], outputLayer=(self.layers['L4'],))

self.layers['L2'] = layer(layer_type='conv', pooling='stride', pooling_size=[2,2], padding='
    valid', is_flatten=False, neuron_type='SigmaDelta', threshold=np.ones([36])*self.
    thresholds[2],  act_fun='ReLU',  shape=[14,47,36], name='cnv2', weight_tensor=self.
    weights[1][0], bias_tensor=self.weights[1][1], outputLayer=(self.layers['L3'],))

self.layers['L1'] = layer(layer_type='conv', pooling='stride', pooling_size=[2,2], padding='
    valid', is_flatten=False, neuron_type='SigmaDelta', threshold=np.ones([24])*self.
    thresholds[1],  act_fun='ReLU',  shape=[31,98,24], name='cnv1', weight_tensor=self.
    weights[0][0], bias_tensor=self.weights[0][1], outputLayer=(self.layers['L2'],))

self.layers['I']  = layer(layer_type='input', name='input', outputLayer=(self.layers['L1'],))
```

**Code Snippet 4.1:** Example composition of layers of PilotNet in SENSIM

```

self.layer_core_map = {
self.layers['I'] : ((self.sim.CI,[range(0,66), range(0,200),range(0,3)],[0.0, 0.0]),),
# L1 : shape=[31,98,24]
self.layers['L1']:((self.sim.coresList[0][0],[range(0, 31),range(0, 98),range(0,24)],[0.0,
    0.0]),),
# L2 : shape=[14,47,36]
self.layers['L2']:((self.sim.coresList[1][0],[range(0, 14),range(0,47),range(0,36)],[0.0,
    0.0]),),
# L3 : shape=[5,22,48]
self.layers['L3']:((self.sim.coresList[2][0],[range(0,5),range(0,22),range(0,48)],[0.0,0.0])
    ,),
# L4 : shape=[3,20,64]
self.layers['L4']:((self.sim.coresList[3][0],[range(0,3),range(0,20),range(0,64)],[0.0,0.0])
    ,),
# L5 : shape=[1,18,64]
self.layers['L5']:((self.sim.coresList[4][0],[range(0,1),range(0,18),range(0,64)],[0.0, 0.0])
    ,),
# L6 : shape=[1164]
self.layers['L6']:((self.sim.coresList[5][0],[range(0,1164)],[0.0, 0.0]),),
# L7 : shape=[100]
self.layers['L7']:((self.sim.coresList[6][0],[range(  0, 100)], [0.0, 0.0]),),
# L8 : shape=[50]
self.layers['L8']:((self.sim.coresList[7][0],[range(  0,  50)], [0.0, 0.0]),),
# L9 : shape=[10]
self.layers['L9']:((self.sim.coresList[8][0],[range(  0,  10)], [0.0, 0.0]),),
# L10: shape=[1]
self.layers['L10']:((self.sim.coresList[9][0],[range(  0, 1)], [0.0, 0.0]),),
# L out
self.layers['O']:((self.sim.CO,[range(0, 1)],[0.0, 0.0]),)}
```

**Code Snippet 4.2:** Direct mapping example of PilotNet on SENSIM

Once the neurons of a layer are assigned to a particular core, the cores must be connected to ensure that data flow between the cores is maintained. Code Snippet 4.42 gives an example of interconnect composition for a set of master to slave cores. **interconnect(master_cores = self.sim.coresList[0][0:1], slave_cores=self.sim.coresList[1][0:1], name='seg1')** connects **coresList[0][0:1]** master cores to slave cores **coresList[1][0:1]** where **[0][0:1]** gives the range of cores in the **[x][y]** axis. The name of the segment is also important to identify the interconnect and the connection.

```
segI = interconnect(master_cores=[self.sim.CI],                slave_cores=self.sim.
    coresList[0][0:1], name='segI')
seg1 = interconnect(master_cores=self.sim.coresList[0][0:1],    slave_cores=self.sim.
    coresList[1][0:1], name='seg1')
seg2 = interconnect(master_cores=self.sim.coresList[1][0:1],    slave_cores=self.sim.
    coresList[2][0:1], name='seg2')
seg3 = interconnect(master_cores=self.sim.coresList[2][0:1],    slave_cores=self.sim.
    coresList[3][0:1], name='seg3')
seg4 = interconnect(master_cores=self.sim.coresList[3][0:1],    slave_cores=self.sim.
    coresList[4][0:1], name='seg4')
seg5 = interconnect(master_cores=self.sim.coresList[4][0:1],    slave_cores=self.sim.
    coresList[5][0:1], name='seg5')
seg6 = interconnect(master_cores=self.sim.coresList[5][0:1],    slave_cores=self.sim.
    coresList[6][0:1], name='seg6')
seg7 = interconnect(master_cores=self.sim.coresList[6][0:1],    slave_cores=self.sim.
    coresList[7][0:1], name='seg7')
seg8 = interconnect(master_cores=self.sim.coresList[7][0:1],    slave_cores=self.sim.
    coresList[8][0:1], name='seg8')
seg9 = interconnect(master_cores=self.sim.coresList[8][0:1],    slave_cores=self.sim.
    coresList[9][0:1], name='seg9')
segO = interconnect(master_cores=self.sim.coresList[9][0:1],    slave_cores=[self.sim.CO],
     name='segO')
```

**Code Snippet 4.3:** Example of composing interconnects on SENSIM (PilotNet)

To obtain an accurate output for a given set of data processed on SENSIM, the cores had to be correctly mapped to a particular core; otherwise, there could be a situation in which the neural network is processing partial data on particular cores.

### 4.1.1. Correct vs incorrect mapping examples

Code Snippet 4.4, Code Snippet 4.5, Code Snippet 4.6, Code Snippet 4.7, Code Snippet 4.8 and Code Snippet 4.9 are different correct mapping schemes of layer 1 neurons to the cores in the first column.

```
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 31),range(0, 98),range(0,24)],[0.0, 0.0]),),
```

**Code Snippet 4.4:** All neurons in layer 1 mapped to core1

Code Snippet 4.5 is an example of a division of neurons by width and the first 31 x 50 x 24 neurons to core 1 and 31 x 48 x 24 to core 2.

```
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 31),range(0,50),range(0,24)],[0.0, 0.0]),
(self.sim.coresList[0][1],[range(0, 31),range(50, 98),range(0,24)],[0.0, 0.0]),
```

**Code Snippet 4.5:** first set of neurons 31 x 50 x 24 mapped to core 1 and 31 x 48 x 24 mapped to core 2

Code Snippet 4.6 is an example of the division of the height of neurons, and the first 20 x 98 x 24 neurons mapped to core 1 and 11 x 98 x 24 mapped to core 2.

```
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 31),range(0,50),range(0,24)],[0.0, 0.0]),
(self.sim.coresList[0][1],[range(0, 31),range(50, 98),range(0,24)],[0.0, 0.0]),
```

**Code Snippet 4.6:** first set of neurons 31 x 90 x 24 mapped to core 1 and 31 x 98 x 24 mapped to core 2

Code Snippet 4.7 is an example of channel-wise division, and the first 31 x 90 x 12 neurons to core 1 and 31 x 98 x 12 neurons to core 2.

```
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 31),range(0, 98),range(0,12)],[0.0, 0.0]),
(self.sim.coresList[0][1],[range(0, 31),range(0, 98),range(12,24)],[0.0, 0.0]),),
```

**Code Snippet 4.7:** first set of neurons 31 x 90 x 12 mapped to core 1 and 31 x 98 x 12 mapped to core 2

Code Snippet 4.7 is an example of channel-wise division into 4 cores, with each core with 31 x 98 x 6 neurons.

```
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 31),range(0, 98),range(0,6)],[0.0, 0.0]),
(self.sim.coresList[0][1],[range(0, 31),range(0, 98),range(6,12)],[0.0, 0.0]),
(self.sim.coresList[0][2],[range(0, 31),range(0, 98),range(12,18)],[0.0, 0.0]),
(self.sim.coresList[0][3],[range(0, 31),range(0, 98),range(18,24)],[0.0, 0.0]),),
```

**Code Snippet 4.8:** channelwise division of layer 1 into 4 cores

Code Snippet 4.7 is an example of channel-wise division into 4 cores, with the first 2 core with 31 x 50 x 12 neurons and the rest 31 x 48 x 12 neurons in the rest core.

```
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 31),range(0, 50),range(0,12)],[0.0, 0.0]),
(self.sim.coresList[0][1],[range(0, 31),range(0, 50),range(12,24)],[0.0, 0.0]),
(self.sim.coresList[0][2],[range(0, 31),range(50, 98),range(0,12)],[0.0, 0.0]),
(self.sim.coresList[0][3],[range(0, 31),range(50, 98),range(12,24)],[0.0, 0.0]),),
```

**Code Snippet 4.9:** channelwise division of layer 1 into 4 cores

Code Snippet 4.10, Code Snippet 4.11 and Code Snippet 4.12 are different incorrect mapping schemes of layer 1 neurons to the cores in the first column. Code Snippet 4.10 mapping is incorrect, since not all neurons are assigned to the core.

```
# L1 : shape=[31,98,24]
self.layers['L1']:((self.sim.coresList[0][0],[range(0, 20),range(0, 98),range(0,24)],[0.0,
    0.0]),),
```

**Code Snippet 4.10:** Incorrect Mapping for layer 1

The mapping shown in Code Snippet 4.11 is incorrect, as not all neurons are mapped to the core. The **[range(0, 31),range(0, 98),range(10,12)]** is missing because the processing of the data in SENSIM would be affected.

```
# L1 : shape=[31,98,24]
self.layers['L1']:((self.sim.coresList[0][0],[range(0, 31),range(0, 98),range(0,10)],[0.0,
    0.0]),),
(self.sim.coresList[0][1],[range(0, 31),range(0, 98),range(12,24)],[0.0, 0.0]),),
```

**Code Snippet 4.11:** Incorrect mapping example of neurons to the core

Code Snippet 4.11 mapping is incorrect since not all neurons are assigned to the core. The range of neurons **[range(0,31), range(50,98), range(0,12)]** is not assigned to any of the cores.

```
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 31),range(0, 50),range(0,12)],[0.0, 0.0]),),
((self.sim.coresList[0][1],[range(0, 15),range(0, 50),range(0,12)],[0.0, 0.0]),),
((self.sim.coresList[0][1],[range(0, 15),range(0, 98),range(12,24)],[0.0, 0.0]),),
((self.sim.coresList[0][1],[range(0, 15),range(0, 98),range(12,24)],[0.0, 0.0]),),
```

**Code Snippet 4.12:** incorrect mapping example of neurons to the core

### 4.1.2. Correct vs. incorrect interconnect examples

As easily one can miss mapping neurons to a core when manually mapping neurons of a layer to particular cores, similarly one can miss connecting mapped cores of a particular layer to the next layers mapped cores. Therefore, the section will cover a few correct and incorrect interconnect setups.

```
#layer mapping
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 15),range(0, 98),range(0,24)],[0.0, 0.0]),),
((self.sim.coresList[0][1],[range(15, 31),range(0, 98),range(0,24)],[0.0, 0.0]),),
# L2 : shape=[14,47,36]
self.layers['L2']:((self.sim.coresList[1][0],[range(0, 14),range(0,47),range(0,36)],[0.0,
    0.0]),),

# incorrect mapping
seg1 = interconnect(master_cores=self.sim.coresList[0][0:1],    slave_cores=self.sim.
    coresList[1][0:1], name='seg1')
```

**Code Snippet 4.13:** Incorrect interconnect setup

```
# layer mapping
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 15),range(0, 98),range(0,24)],[0.0, 0.0]),),
((self.sim.coresList[0][1],[range(15, 31),range(0, 98),range(0,24)],[0.0, 0.0]),),
# L2 : shape=[14,47,36]
self.layers['L2']:((self.sim.coresList[1][0],[range(0, 14),range(0,47),range(0,36)],[0.0,
    0.0]),),

# incorrect mapping
seg1 = interconnect(master_cores=self.sim.coresList[0][0:2],    slave_cores=self.sim.
    coresList[1][0:1], name='seg1')
```

**Code Snippet 4.14:** Correct interconnect setup

Although mapping and forming interconnections is a tedious and error-prone task in the simulator, several utilities were developed to handle this, detailed in Section 4.3.

## 4.2. Analysis framework

In order to understand the effect of different mappings on the various core attributes, an analysis framework was developed using the Jupyter Notebook and Pandas framework in Python. The framework helped to understand a brief description of the variety of different manual mappings. The parameter values for which this framework was designed were those mentioned in Table 2.16 and Table 2.17.

The analytics framework can be subdivided into the analytics framework for neuromorphic core analysis and interconnect analysis. In addition to having a comprehensive analytics framework for interconnects and core, the analytics framework for application output helps to understand the execution of the application on SENSIM and the accuracy of the output.

### 4.2.1. Analysis framework for neuromorphic cores

The analysis framework provides a detailed visual inspection of the following parameters.

- Relative energy per core
- Relative energy for every digital component per core
- Processor utilization per core
- Peak in/out queue occupancy for a given core
- Comparison of cumulative energy between the cores
- Comparison of processor utilization between the cores
- Comparison of total energy between the cores

Figure 4.1 gives a total energy consumption of the cores **C0_0**, **C4_0**, **C8_0**, **C9_0** respectively on SENSIM for the 2000 PiloNet images dataset. It can be seen that the energy consumption of **C8_0**, **C9_0** is $\approx 10^5$ magnitudes of order less, compared to the energy consumption of **C0_0** and $\approx 10^3$ magnitudes of order less, compared to **C0_0**.

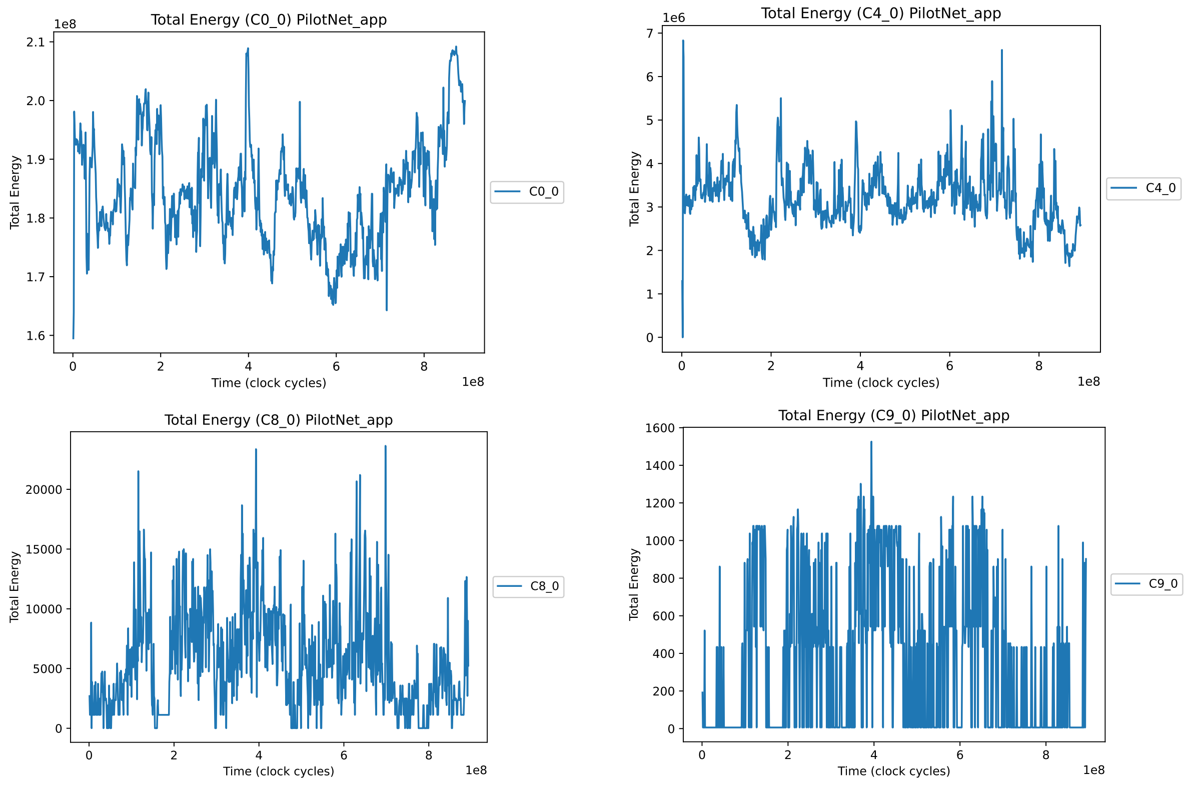


**Figure 4.1:** Energy consumption per core for C0_0, C4_0, C8_0, C9_0 with PilotNet mapped onto SENSIM and inferred for 2000 images

Figure 4.2 gives a total energy consumption for every component (FIFO, DMEM, NPE, controller) of the cores **C0_0**, **C4_0**, **C8_0**, **C9_0** respectively in SENSIM for the 2000 PiloNet images dataset. From Figure 4.2 one can observe that the energy consumption of reading and writing data from and to the memory is comparatively higher than the energy consumption of the data movement in FIFO queues and the energy consumption of the controller or NPE operations. From Figure 4.2 it can be seen that for the core **C9_0** with only one neuron, the energy consumption of FIFO could be relatively exceeded.

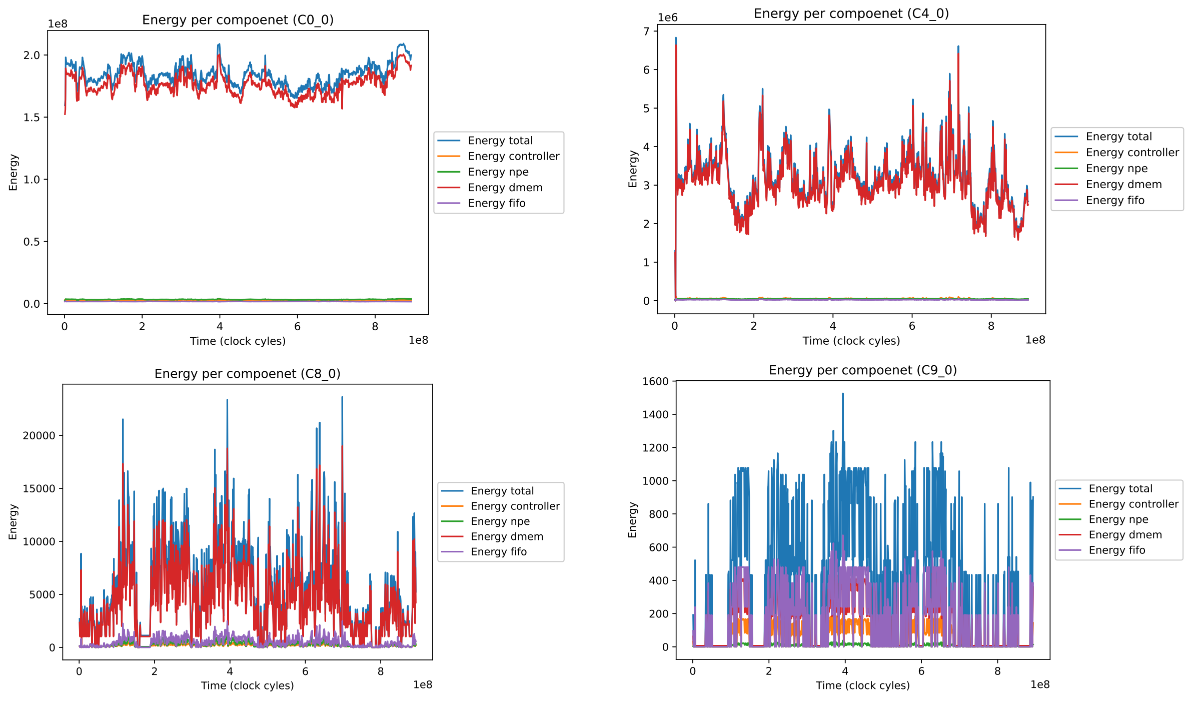


**Figure 4.2:** Energy consumption for different digital components (NPE, Controller, Data Memory, fifo and total) of C0_0, C4_0, C8_0, C9_0 cores with PilotNet mapped on SENSIM and inferred for 2000 images

Figure 4.3 gives a processor utilization of cores **C0_0**, **C4_0**, **C8_0**, **C9_0** respectively on SENSIM for the 2000 PiloNet images dataset. From Figure 4.3, observations can be made that **C0_0** is always used to 100% while the processor core **C4_0** is used only 3% at maximum and the rest of the time the processor is underutilized. The cores **C8_0** and **C9_0** are even more underutilized.

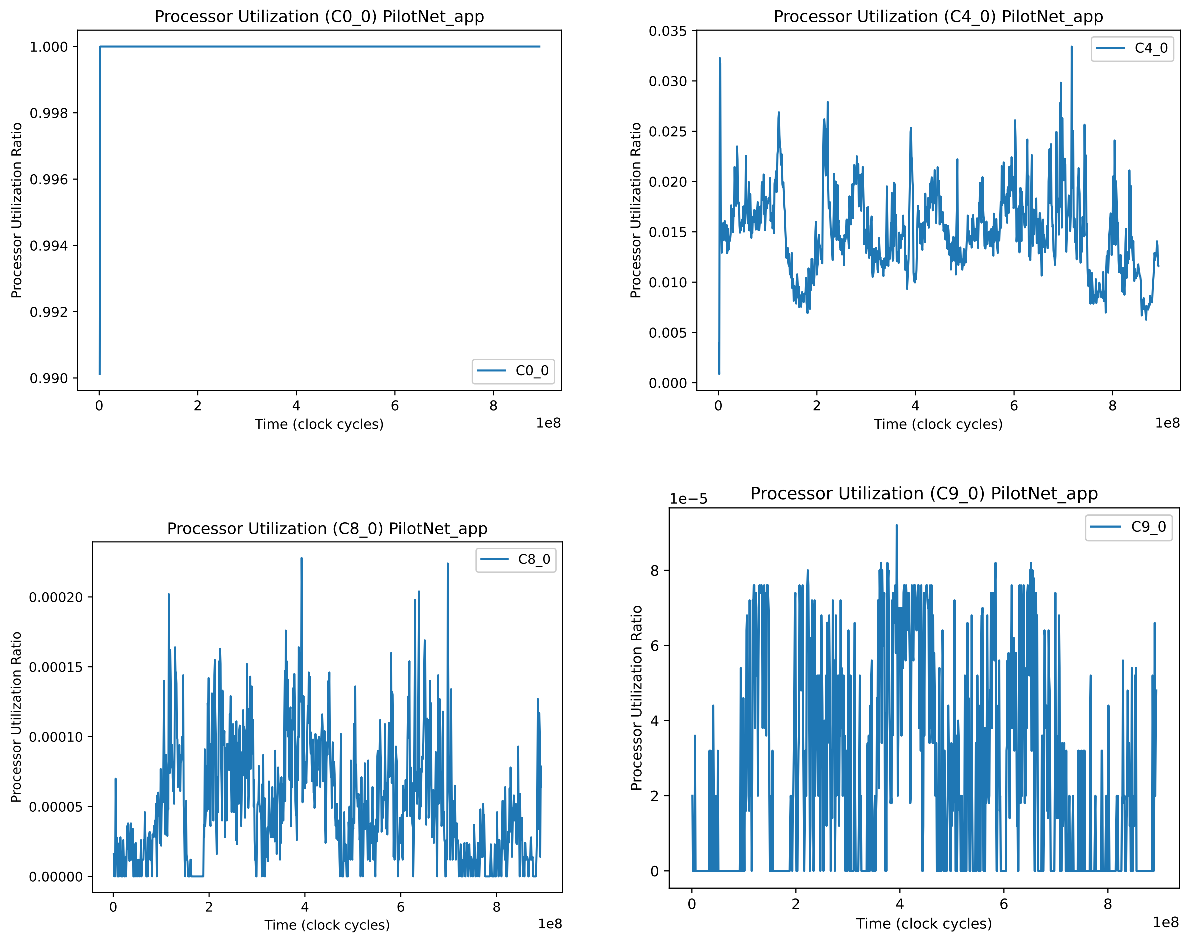


**Figure 4.3:** Processor utilization of C0_0, C4_0, C8_0, C9_0 core when PilotNet is mapped on SENSIM and inferred for 2000 images

Figure 4.4 gives the Peak In queue and Out queue occupancy for one snapshot time period of cores **C0_0**, **C4_0**, **C8_0**, **C9_0** respectively on SENSIM for the 2000 PiloNet images dataset. From Figure 4.3, observations can be made that the peak of **C0_0** in the queue varies between 1280 and 1287, while the outside queue remains stagnant for the complete simulation at 1280. For **C4_0**, **C8_0**, **C9_0**, the **peak in-queue** occupancy is relatively higher than the **peak out-queue** but compared to **C0_0** it is lower.

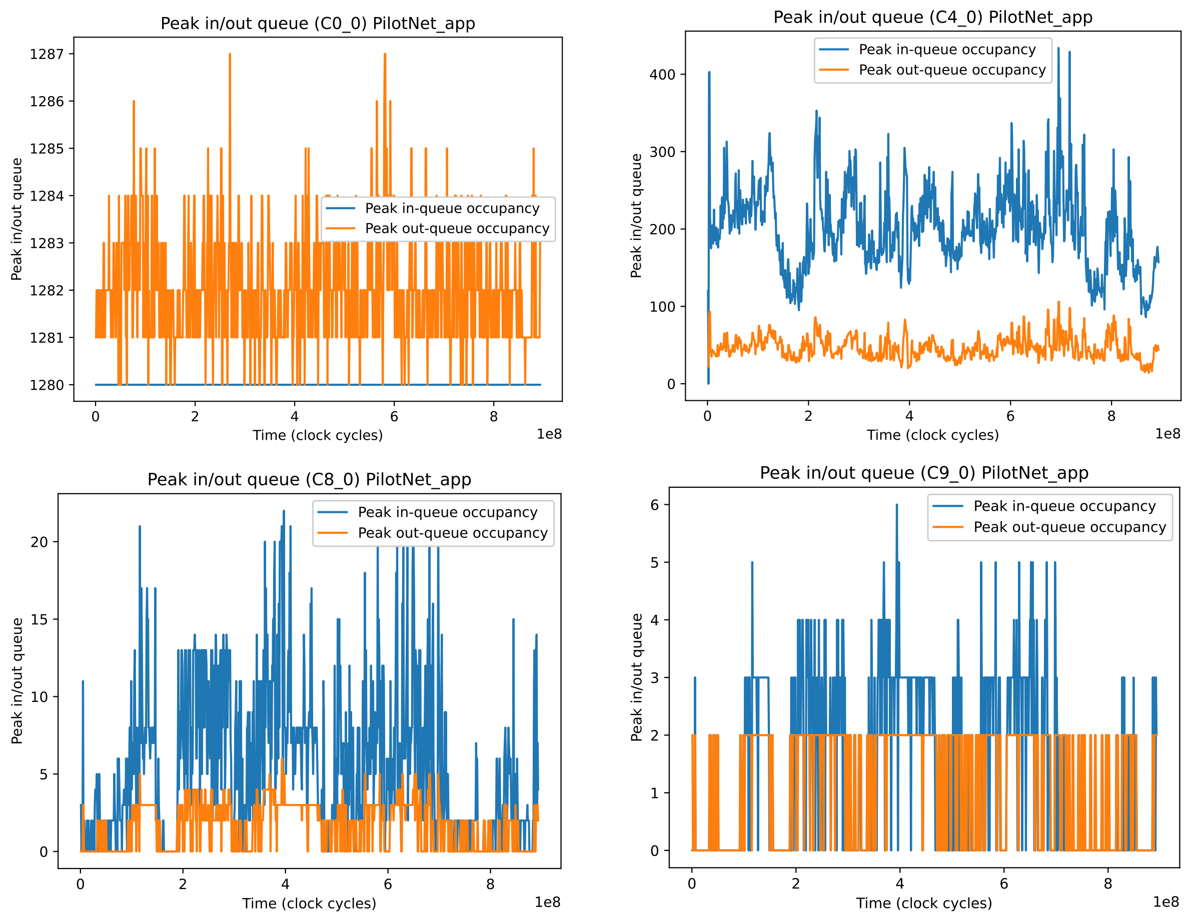


**Figure 4.4:** Peak in out queue occupancy of C0_0, C4_0, C8_0, C9_0 core when PilotNet is mapped on SENSIM and inferred for 2000 images

Figure 4.5 gives the cumulative energy of all the cores. Further observations can be made that the energy consumption of **C0_0** is greater than **C1_0**, but the increase in **C2_0** energy increases much slower than **C1_0**. With cumulative energy plots, observations can also be made when the processor is idle by observing the. Figure 4.6 gives the comparative plot of processor utilization of all cores for a complete simulation. Figure 4.7 gives the comparative plot of the energy consumption of all cores for a complete simulation. From processor utilitation and energy consumption plots, a visual comparison can be made when a particular core is utilized more or consumes more energy in the simulation.

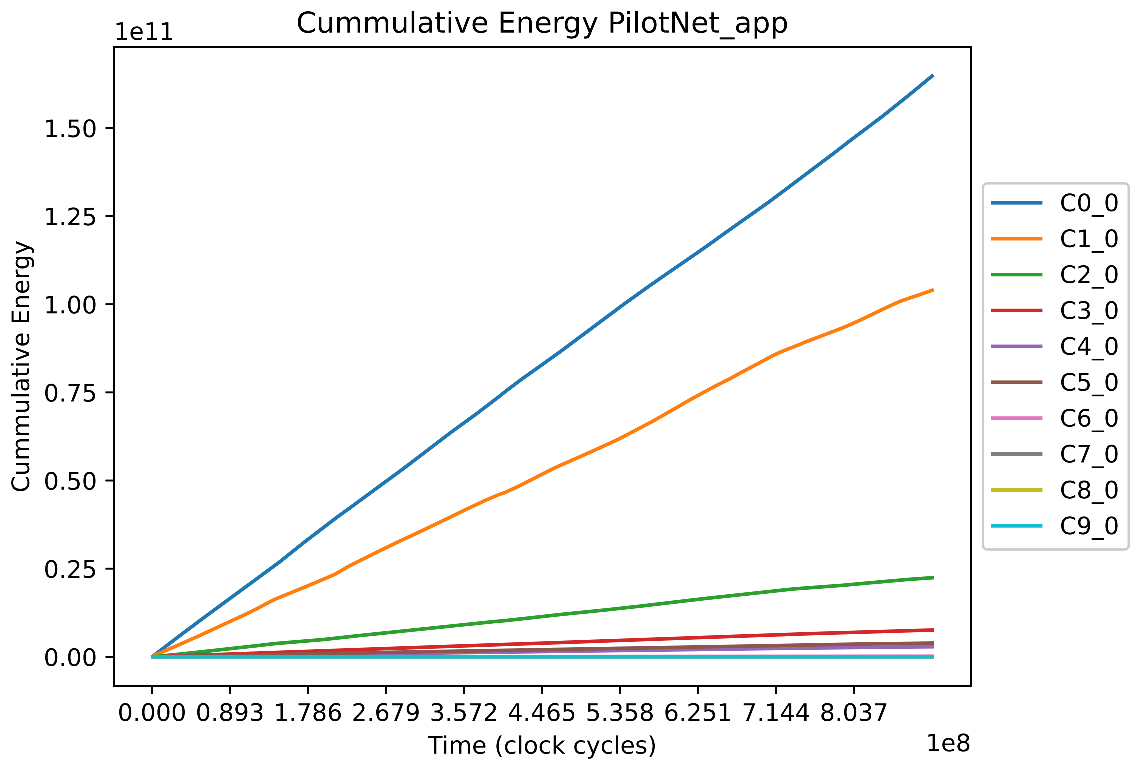


**Figure 4.5:** Cumulative energy for all cores with PilotNet mapped on SENSIM 2000 images were inferred

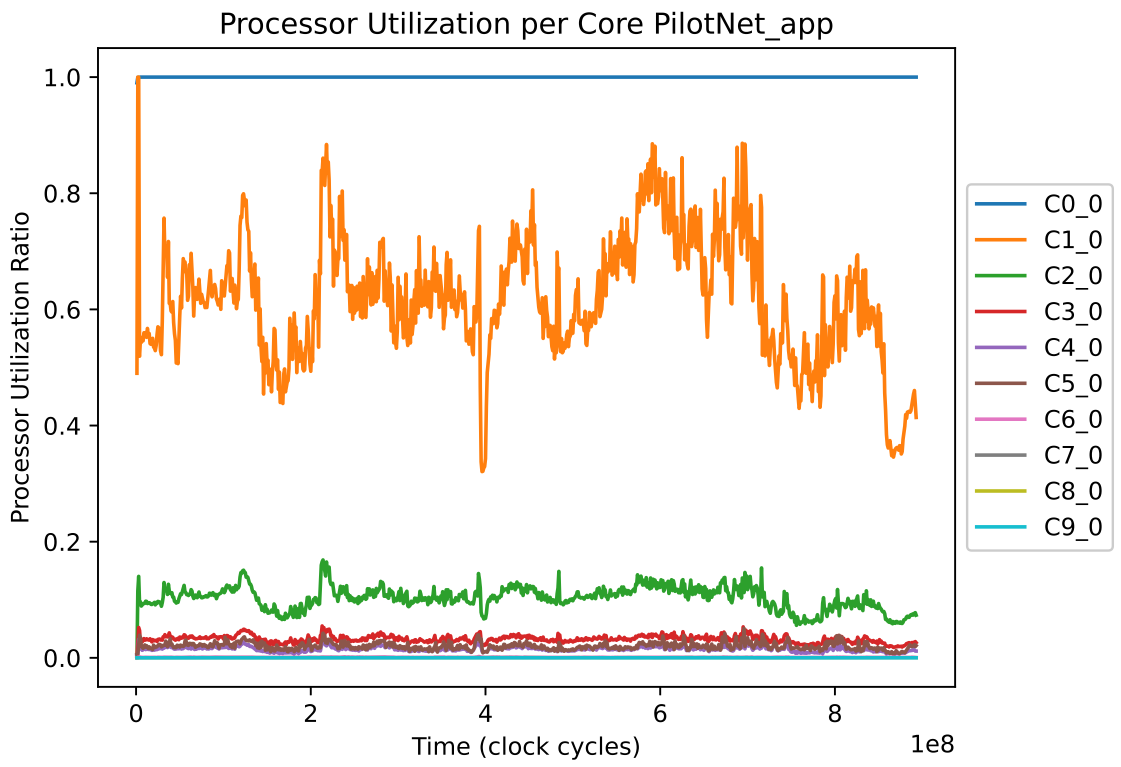


**Figure 4.6:** Processor utilization for all cores with PilotNet mapped on SENSIM 2000 images were inferred

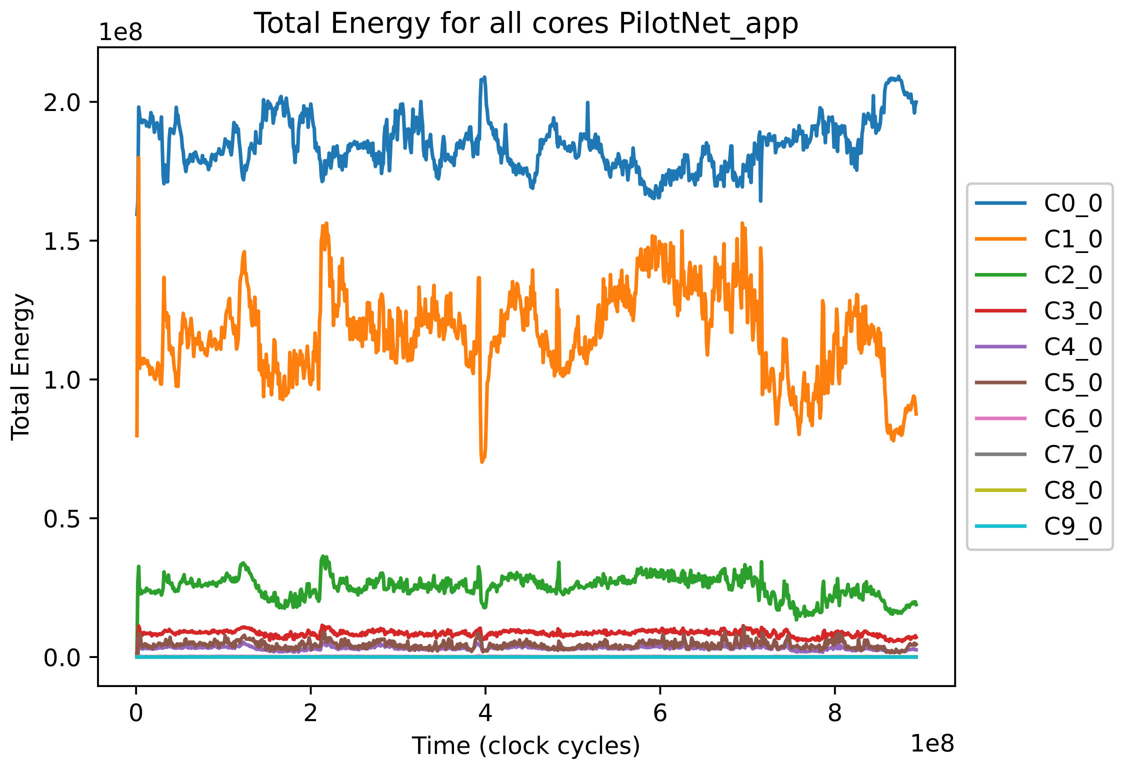


**Figure 4.7:** Total energy for all the cores for all cores with PilotNet mapped on SENSIM 2000 images were inferred

### 4.2.2. Analysis framework for interconnects

Figure 4.14 represents the relative energy consumption of the segments in the interconnect. The interconnect segment **seg** maps the virtual input core to the set of cores assigned to the first layer. The interconnect segment **seg1** represents the connection between a set of cores mapped from layer 1 to a set of cores mapped to layer 2. From Figure 4.14, observations can be made that the segments **seg5** and **seg9** consume less energy than the segments **seg** and **seg1**. Another observation that can be made is that the energy consumption of neuromorphic cores as compared to segments is on the order of magnitudes of 50 to 100 times more.

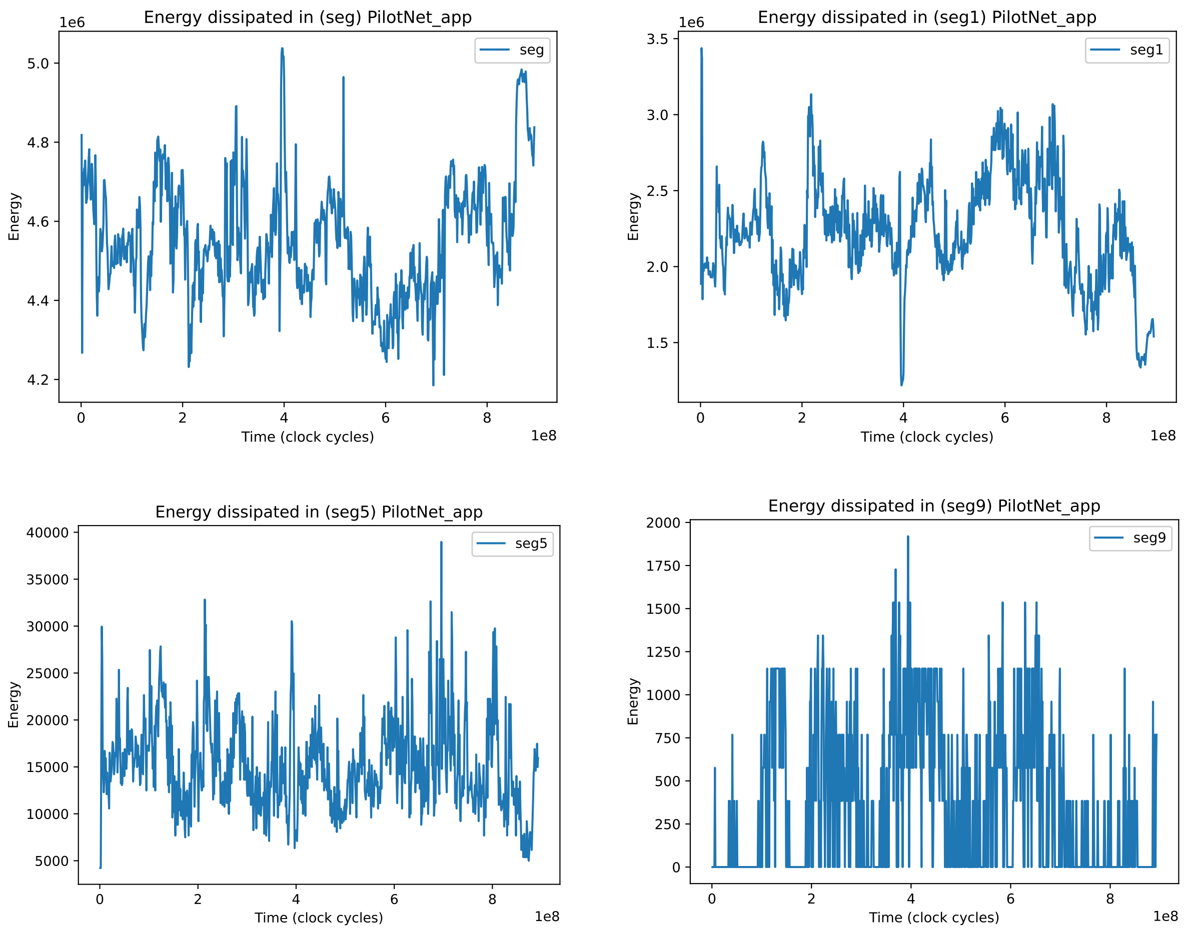


**Figure 4.8:** Energy consumption for segments (seg0,1,5,9) with PilotNet mapped on SENSIM 2000 images were inferred

Figure 4.9 gives the relative energy consumption of different segments in the network on the chip.

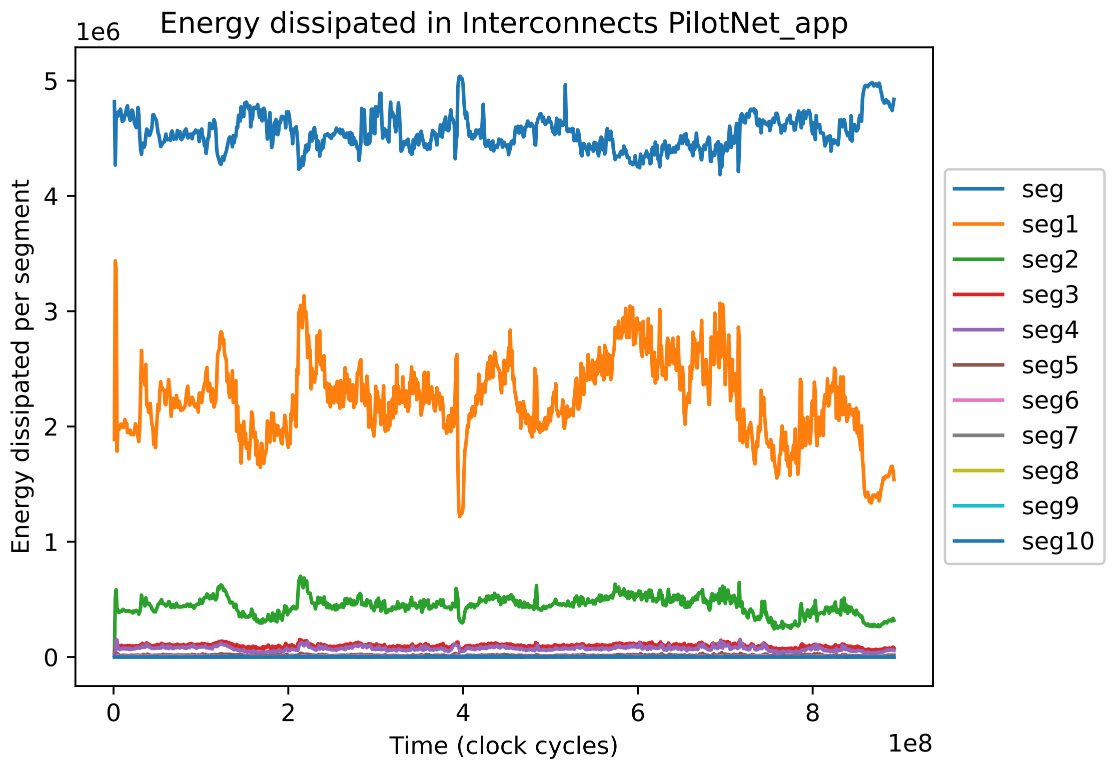
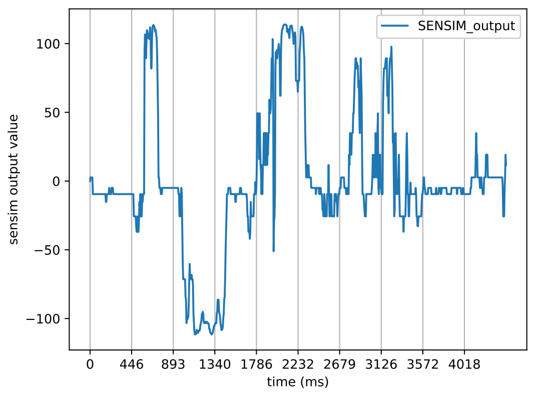


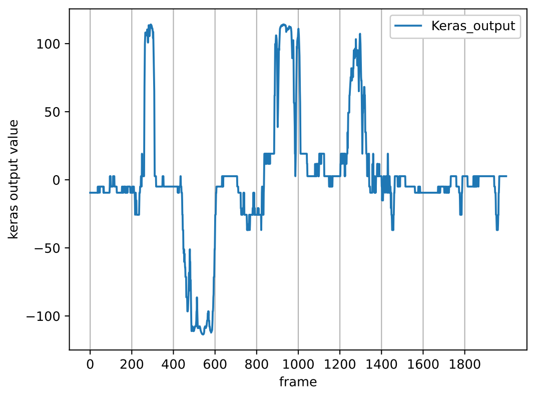
**Figure 4.9:** Energy comparison for all the segments with PilotNet mapped on SENSIM 2000 images were inferred

### 4.2.3. Output comparison framework

PilotNet's output is the steering angle. To ensure that SENSIM produces an inference result similar to TensorFlow Keras, timely snapshots of the output are taken and plotted on a graph Figure 4.10a. In a comparison of the output of SENSIM with Tensorflow Keras, the output seems to superimpose well, and by visual inspection, one can conclude if there is a drop in accuracy or a change in the end result due to mapping. Figure 4.10 gives a visual comparison of the output of the two systems.



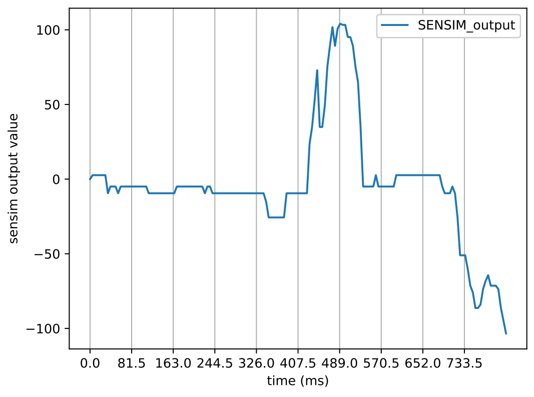
**(a)** SENSIM output for 2000 Pilot images



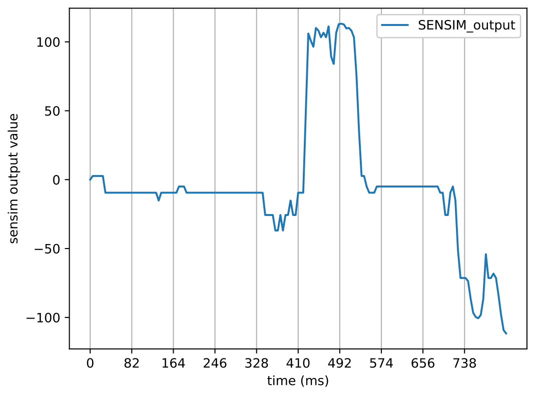
**(b)** Keras output for 2000 Pilot images

**Figure 4.10:** Comparison of SENSIM output with 2000 images

For experimentation purposes, instead of using the 2000 images, 500 images were used for inference for faster convergence time. Figure 4.11 gives a comparison of an accurate output versus an inaccurate output. Figure 4.13a would be the output when neurons are incorrectly mapped on the SENSIM framework. Hence, a visual inspection of the signal was performed at every step. For experimentation purposes, 10-20-30 images from the less sparse and more sparse segments were extracted.



**(a)** SENSIM inaccurate output for 500 Pilot images



**(b)** SENSIM accurate output for 500 Pilot images

**Figure 4.11:** Comparison of SENSIM output with the keras output with 500 images

## 4.3. SENMap utilities

To operate on layer core mapping, extract neurons, partition them, and model an optimization problem, several utilities were developed. The utilities made for SENMap were classified as follows.

- MetaData extraction utilities
- Partitioning utilities
- Mapping compression utilities
- Clustering utilities

### 4.3.1. Meta-data extraction utilities

#### Extracting the core Map and neuron Map

To simplify the mapping process, from the cumbersome mapping Code Snippet 4.2 it was essential to extract metadata; therefore, **layer to core map** also called **coreMap** and **layer to neuron map** also called **neuronMap** were extracted from Code Snippet 4.2. Code Snippet 4.15 and Code Snippet 4.16 are examples of **coreMap** and **neuronMap** extracted from the mapping, which will be used from now on for the mapping process.

```
coreMap = {'I': 'CI',
 'L1': ['C0_0'],
 'L10': ['C9_0'],
 'L2': ['C1_0'],
 'L3': ['C2_0'],
 'L4': ['C3_0'],
 'L5': ['C4_0'],
 'L6': ['C5_0'],
 'L7': ['C6_0'],
 'L8': ['C7_0'],
 'L9': ['C8_0'],
 'O': 'CO'}
```

**Code Snippet 4.15:** Core mapping example for direct mapping (PilotNet)

```
neuronMap = {
    'O':  [[range(0, 1)]],
    'L10':[[range(0, 1)]],
    'L9': [[range(0, 10)]],
    'L8': [[range(0, 50)]],
    'L7': [[range(0, 100)]],
    'L6': [[range(0, 1164)]],
    'L5': [[range(0, 1), range(0, 18), range(0, 64)]],
    'L4': [[range(0, 3), range(0, 20), range(0, 64)]],
    'L3': [[range(0, 5), range(0, 22), range(0, 48)]],
    'L2': [[range(0, 14), range(0, 47), range(0, 36)]],
    'L1': [[range(0, 31), range(0, 98), range(0, 24)]],
    'I':  [[range(0, 66), range(0, 200), range(0, 3)]]
  }
```

**Code Snippet 4.16:** Neuron mapping example for direct mapping (PIlotNet)

**Rebuilding the layer core map and interconnect**
Once the metadata **coreMap** and **layerMap** is extracted and modified, the core map of the original layer and the interconnection had to be recreated to make it compatible with the SENSIM framework. The utility does not create a new layer core mapping, but modifies the preexisting objects made at the time of application definition according to **neuronMap** and **coreMap** as shown in Code Snippet 4.18 and Code Snippet 4.17. By doing so, unnecessary memory wastage is avoided and software is made more memory efficient.

To understand further features, the coreMap in Code Snippet 4.17 and the neuronMap in Code Snippet 4.18 will be used.

```
coreMap = {'I': 'CI',
 'L1': ['C0_0', 'C0_1'],
 'L10': ['C9_0'],
 'L2': ['C1_0', 'C1_1', 'C1_2'],
 'L3': ['C2_0'],
 'L4': ['C3_0'],
 'L5': ['C4_0'],
 'L6': ['C5_0'],
 'L7': ['C6_0', 'C6_1', 'C6_2'],
 'L8': ['C7_0'],
 'L9': ['C8_0'],
 'O': 'CO'}
```

**Code Snippet 4.17:** Core mapping example (coreMap)(PilotNet)

```
neuronMap = {
    'O':  [[range(0, 1)]],
    'L10':[[range(0, 1)]],
    'L9': [[range(0, 10)]],
    'L8': [[range(0, 50)]],
    'L7': [[range(0, 100)]],
    'L6': [[range(0, 582)], [range(582, 1164)]],
    'L5': [[range(0, 1), range(0, 18), range(0, 64)]],
    'L4': [[range(0, 3), range(0, 20), range(0, 64)]],
    'L3': [[range(0, 5), range(0, 22), range(0, 48)]],
    'L2': [[range(0, 14), range(0, 47), range(0, 12)],[range(0, 14), range(0, 47), range(12,
    20)], [range(0, 14), range(0, 47), range(20, 36)]],
    'L1': [[range(0, 31), range(0, 98), range(0, 12)],[range(0, 31), range(0, 98), range(12,
    24)] ],
    'I':  [[range(0, 66), range(0, 200), range(0, 3)]]}
```

**Code Snippet 4.18:** Neuron mapping example (neuronMap) (PilotNet)

```
self.layer_core_map = {
self.layers['I'] :
((self.sim.CI,[range(0,66), range(0,200),range(0,3)],[0.0, 0.0]),),
# L1 : shape=[31,98,24]
self.layers['L1']:
((self.sim.coresList[0][0],[range(0, 31),range(0, 98),range(0,12)],[0.0, 0.0]),
((self.sim.coresList[0][1],[range(0, 31),range(0, 98),range(12,24)],[0.0, 0.0]),),
# L2 : shape=[14,47,36]
self.layers['L2']:
((self.sim.coresList[1][0],[range(0, 14),range(0,47),range(0,12)],[0.0, 0.0]),
(self.sim.coresList[1][1],[range(0, 14),range(0,47),range(12,20)],[0.0, 0.0]),
(self.sim.coresList[1][2],[range(0, 14),range(0,47),range(20,36)],[0.0, 0.0]),),
# L3 : shape=[5,22,48]
self.layers['L3']:
((self.sim.coresList[2][0],[range(0,5),range(0,22),range(0,48)],[0.0,0.0]),),
# L4 : shape=[3,20,64]
self.layers['L4']:
((self.sim.coresList[3][0],[range(0,3),range(0,20),range(0,64)],[0.0,0.0]),),
# L5 : shape=[1,18,64]
self.layers['L5']:
((self.sim.coresList[4][0],[range(0,1),range(0,18),range(0,64)],[0.0, 0.0]),),
# L6 : shape=[1164]
self.layers['L6']:
((self.sim.coresList[5][0],[range(0,582)],[0.0, 0.0]),
(self.sim.coresList[5][1],[range(582,1164)],[0.0, 0.0]),),
# L7 : shape=[100]
self.layers['L7']:
((self.sim.coresList[6][0],[range(  0, 100)], [0.0, 0.0]),),
# L8 : shape=[50]
self.layers['L8']:
((self.sim.coresList[7][0],[range(  0,  50)], [0.0, 0.0]),),
# L9 : shape=[10]
self.layers['L9']:
((self.sim.coresList[8][0],[range(  0,  10)], [0.0, 0.0]),),
# L10: shape=[1]
self.layers['L10']:
((self.sim.coresList[9][0],[range(  0, 1)], [0.0, 0.0]),),
# L out
self.layers['O']:((self.sim.CO,[range(0, 1)],[0.0, 0.0]),)}
```

**Code Snippet 4.19:** Layer core map composition from coreMap and neuronMap

```
segI = interconnect(master_cores=[self.sim.CI],                slave_cores=self.sim.
    coresList[0][0:2], name='segI')
seg1 = interconnect(master_cores=self.sim.coresList[0][0:2],    slave_cores=self.sim.
    coresList[1][0:3], name='seg1')
seg2 = interconnect(master_cores=self.sim.coresList[1][0:3],    slave_cores=self.sim.
    coresList[2][0:1], name='seg2')
seg3 = interconnect(master_cores=self.sim.coresList[2][0:1],    slave_cores=self.sim.
    coresList[3][0:1], name='seg3')
seg4 = interconnect(master_cores=self.sim.coresList[3][0:1],    slave_cores=self.sim.
    coresList[4][0:1], name='seg4')
seg5 = interconnect(master_cores=self.sim.coresList[4][0:1],    slave_cores=self.sim.
    coresList[5][0:2], name='seg5')
seg6 = interconnect(master_cores=self.sim.coresList[5][0:2],    slave_cores=self.sim.
    coresList[6][0:1], name='seg6')
seg7 = interconnect(master_cores=self.sim.coresList[6][0:1],    slave_cores=self.sim.
    coresList[7][0:1], name='seg7')
seg8 = interconnect(master_cores=self.sim.coresList[7][0:1],    slave_cores=self.sim.
    coresList[8][0:1], name='seg8')
seg9 = interconnect(master_cores=self.sim.coresList[8][0:1],    slave_cores=self.sim.
    coresList[9][0:1], name='seg9')
segO = interconnect(master_cores=self.sim.coresList[9][0:1],    slave_cores=[self.sim.CO],
     name='segO')
```

**Code Snippet 4.20:** Interconnects formation neuron map and core map

### neuronMap to neuronsPerLayer and neuronsPerCore Conversion

The utility essentially extracts **neuronsPerCore** and **neuronsPerLayer** which are used in more detail in other utilities and Neuron Partitioning and clustering. The utility takes **neuronMap** from Code Snippet 4.18 and gives the neurons processed by a given core as shown in Code Snippet 4.21 and the neurons processed by a layer as shown in Code Snippet 4.22.

```
neuronsPerCore =
{'I': [39600],
 'L1': [36456, 36456],
 'L10': [1],
 'L2': [7896, 5264, 10528],
 'L3': [5280],
 'L4': [3840],
 'L5': [1152],
 'L6': [582, 582],
 'L7': [100],
 'L8': [50],
 'L9': [10],
 'O': [1]})

```

**Code Snippet 4.21:** neurons per core (PilotNet)

```
neuronsPerLayer =
{'I': 39600,
  'L1': 72912,
  'L10': 1,
  'L2': 23688,
  'L3': 5280,
  'L4': 3840,
  'L5': 1152,
  'L6': 1164,
  'L7': 100,
  'L8': 50,
  'L9': 10,
  'O': 1},

```

**Code Snippet 4.22:** neurons per layer (PilotNet)

**neuronMap to channelMap**

The utility essentially extracts channel neurons from the neural network. In the case of a dense layer, the total number of neurons in the layer will be equal to the channel neurons, but with other layer types, such as convolution, the neurons are spread in 3 dimensions [Height, Width, Channel]. The utility is used for the division of neurons channel-wise. The utility takes **neuronMap** from Code Snippet 4.18 as input and gives **channelMap** Code Snippet 4.23 as output.

```
channelMap =
{'I': [3],
 'L1': [12, 12],
 'L10': [1],
 'L2': [12, 8, 16],
 'L3': [48],
 'L4': [64],
 'L5': [64],
 'L6': [582, 582],
 'L7': [100],
 'L8': [50],
 'L9': [10],
 'O': [1]}

```

**Code Snippet 4.23:** neurons in the channel per Core (channelMap) (PilotNet)

**neuronMap to neuronCountMap**

The utility takes neuronMap as input and extracts the count of neurons per layer, per core, and per dimension, which is also used for the partition of neurons. For example, the utility would convert **neuronMap** in Code Snippet 4.18 to **neuronMap** in Code Snippet 4.24

```
neuronCountMap =
{'I': [[66, 200, 3]],
 'L1': [[31, 98, 12], [31, 98, 12]],
 'L10': [[1]],
 'L2': [[14, 47, 12], [14, 47, 8], [14, 47, 16]],
 'L3': [[5, 22, 48]],
 'L4': [[3, 20, 64]],
 'L5': [[1, 18, 64]],
 'L6': [[582], [582]],
 'L7': [[100]],
 'L8': [[50]],
 'L9': [[10]],
 'O': [[1]]}
```

```

```

**Code Snippet 4.24:** neuron count per dimension (PilotNet)

**neuronMap from neuronCountMap**
The utility takes the **neuronCountMap** extracted per layer per core per dimension and rebuilds the neuron map. The utility is essential to rebuild **neuronMap** after applying the partition scheme.

### 4.3.2. Partitioning utilities

***Note:*** *Since clustering of neurons in a layer or clustering layers was not possible with the current SENSIM SDK, clustering schemes were not part of the current design and implementation, but will be discussed in the future in Section 6.2.*

For partitioning, neurons could be divided greedily, where the neurons of a core are filled up to their maximum, and the remaining neurons are used by the next core. The other partitioning strategy would be to balance the neurons between the cores in such a way that one core is not overutilized and there remains a balance between the cores. Balanced neuron division would be a better strategy to ensure that one processor is not overutilized and becomes a bottleneck in the processing pipeline. How many cores would be optimal for any given layer or application would be an optimization problem discussed in Section 4.4

**Balanced Neuron division Per Layer**
The utility essentially is a neuron partitioning scheme in which neurons per layer are divided among allotted cores in a balanced manner. Since the partitioning scheme. Utility taken in **neuronMap**, **corePerLayer** in the new mapping. The assignment of cores per layer was only limited to the cores assigned in the Y direction, hence also known as **YCoreMap**, but was extended further to the cores in any direction with compression of the mapping Section 4.3.3. The utility takes the input as **corePerLayer** as shown in Code Snippet 4.25 and gives **newCoreMap** in Code Snippet 4.26 and **newNeuronMap** in Code Snippet 4.27.

```
corePerLayer =
{'I': 0,
 'L1': 3,
 'L10': 1,
 'L2': 4,
 'L3': 2,
 'L4': 9,
 'L5': 2,
 'L6': 1,
 'L7': 3,
 'L8': 2,
 'L9': 3,
 'O': 0}

```

**Code Snippet 4.25:** Cores per Layer Mapping for creating balanced neuron division (PilotNet)

```
newCoreMap =
{'I': 'CI',
 'L1': ['C0_0', 'C0_1', 'C0_2'],
 'L10': ['C9_0'],
 'L2': ['C1_0', 'C1_1', 'C1_2', 'C1_3'],
 'L3': ['C2_0', 'C2_1'],
 'L4': ['C3_0', 'C3_1', 'C3_2', 'C3_3', 'C3_4', 'C3_5', 'C3_6', 'C3_7', 'C3_8'],
 'L5': ['C4_0', 'C4_1'],
 'L6': ['C5_0'],
 'L7': ['C6_0', 'C6_1', 'C6_2'],
 'L8': ['C7_0', 'C7_1'],
 'L9': ['C8_0', 'C8_1', 'C8_2'],
 'O': 'CO'}

```

**Code Snippet 4.26:** CoreMap extracted from number of cores per layer (PilotNet)

```
newNeuronMap =
{'I': [[range(0, 66), range(0, 200), range(0, 3)]],
 'L1': [[range(0, 31), range(0, 98), range(0, 8)],
        [range(0, 31), range(0, 98), range(8, 16)],
        [range(0, 31), range(0, 98), range(16, 24)]],
 'L10': [[range(0, 1)]],
 'L2': [[range(0, 14), range(0, 47), range(0, 9)],
        [range(0, 14), range(0, 47), range(9, 18)],
        [range(0, 14), range(0, 47), range(18, 27)],
        [range(0, 14), range(0, 47), range(27, 36)]],
 'L3': [[range(0, 5), range(0, 22), range(0, 24)],
        [range(0, 5), range(0, 22), range(24, 48)]],
 'L4': [[range(0, 3), range(0, 20), range(0, 8)],
        [range(0, 3), range(0, 20), range(8, 15)],
        [range(0, 3), range(0, 20), range(15, 22)],
        [range(0, 3), range(0, 20), range(22, 29)],
        [range(0, 3), range(0, 20), range(29, 36)],
        [range(0, 3), range(0, 20), range(36, 43)],
        [range(0, 3), range(0, 20), range(43, 50)],
        [range(0, 3), range(0, 20), range(50, 57)],
        [range(0, 3), range(0, 20), range(57, 64)]],
 'L5': [[range(0, 1), range(0, 18), range(0, 32)],
        [range(0, 1), range(0, 18), range(32, 64)]],
 'L6': [[range(0, 1164)]],
 'L7': [[range(0, 34)], [range(34, 67)], [range(67, 100)]],
 'L8': [[range(0, 25)], [range(25, 50)]],
 'L9': [[range(0, 4)], [range(4, 7)], [range(7, 10)]],
 'O': [[range(0, 1)]]}

```

**Code Snippet 4.27:** NeuronMap with balanced division of neurons channelwise (PilotNet)

**Memory utilization per core/layer estimate**

Before placing neurons in the core, it was essential to know what memory technology is used in the architecture to understand how many weights, neurons, and thresholds can be assigned to a particular core. Fetching data from external memory can be expensive in terms of energy and latency; hence, the main idea would be to obtain a minimum number of cores needed to store the data in local memory.

**Step 1:** Extract the number of weights and biases from the weights and biases stored in a <file>.npy file used by the simulator for inference. Code Snippet 4.28 gives the number of weights and biases in PilotNet

**Step 2:** Calculate the weights and biases to be stored per core (based on the division of neurons per core) from **corePerLayer** and the number of weights and biases calculated per layer.

**Step 3:** The number of thresholds used per core would be the same as the number of biases per core. ***Note:*** *The number of thresholds, bias, and channel neurons remains the same for the experiment since the thresholds in the experiment are chosen channel-wise and not neuron-wise or layer-wise.*

**Step 4:** Calculate the memory consumption per core by plugging it in Equation 4.1.

**Step 5:** Compare the estimated memory with the memory per core constraint.

**Step 6:** Extract the lower limit for **corePerLayer**. Code Snippet 4.33 is the lower limit of any application (PilotNet) if **Core_DMEM** is 1 Mb (1024 * 1024). The lower limit will change based on **Core_DMEM** and the weights, biases, thresholds, and states stored locally and not externally.

**Step 7:** Extract **neuronMap** and **coreMap** from **corePerLayer** mapping

$$M_{pc} = (N_{npc} + (N_{tpc} * F_{SNN}) \times (BW\_States + BW\_Outputs)) + (N_{wpc} + N_{bpc}) \times BW\_weights \quad (4.1)$$

$$M_{pl} = \sum_{C_{x,0}}^{C_{x,y}} M_{pc} \quad (4.2)$$

where:

$N_{npc}$ = number of neurons per core
$N_{wpc}$ = number of weights per core
$N_{bpc}$ = number of biases per core
$N_{npc}$ = number of thresholds per core
$M_{pl}$ = memory utilization per layer
$M_{pc}$ = memory utilization per core
$F_{snn}$ = a boolean flag which is taken in true value in case of an SNN
$x$ = a particular layer in the network
$y$ = cores assigned to a particular layer

```
weightsPerLayer=
[1800, 21600, 43200, 27648, 36864, 1340928, 116400, 5000, 500, 10]
biasPerLayer=
[24, 36, 48, 64, 64, 1164, 100, 50, 10, 1]
```

**Code Snippet 4.28:** Weights and biases per layer (PilotNet)

```
biasPerCore
{'I': 0,
 'L1': array([8., 8., 8.]),
 'L10': array([1.]),
 'L2': array([9., 9., 9., 9.]),
 'L3': array([24., 24.]),
 'L4': array([8., 7., 7., 7., 7., 7., 7., 7., 7.]),
 'L5': array([32., 32.]),
 'L6': array([1164.]),
 'L7': array([34., 33., 33.]),
 'L8': array([25., 25.]),
 'L9': array([4., 3., 3.]),
 'O': 0}
```

**Code Snippet 4.29:** Number of bias per core based on the core per layer division (PilotNet)

```
weightsPerCore
{'I': 0,
 'L1': array([600., 600., 600.]),
 'L10': array([10.]),
 'L2': array([5400., 5400., 5400., 5400.]),
 'L3': array([21600., 21600.]),
 'L4': array([3456., 3024., 3024., 3024., 3024., 3024., 3024., 3024., 3024.]),
 'L5': array([18432., 18432.]),
 'L6': array([1340928.]),
 'L7': array([39576., 38412., 38412.]),
 'L8': array([2500., 2500.]),
 'L9': array([200., 150., 150.]),
 'O': 0}

```

**Code Snippet 4.30:** Number of weights per core based on the core per layer division and weights per layer (PilotNet)

```
weightsPerCore =
{'I': 0,
  'L1': array([415736., 415736., 415736.]),
  'L10': array([78.]),
  'L2': array([122463., 122463., 122463., 122463.]),
  'L3': array([131784., 131784.]),
  'L4': array([22152., 19383., 19383., 19383., 19383., 19383., 19383., 19383.,19383.]),
  'L5': array([84192., 84192.]),
  'L6': array([5407944.]),
  'L7': array([159596., 154902., 154902.]),
  'L8': array([10950., 10950.]),
  'L9': array([952., 714., 714.]),
  'O': 0},
```

**Code Snippet 4.31:** NeuronMap with balanced division of neurons Channelwise (PilotNet)

```
memoryPerLayer (memory density) =
{'I': 0,
  'L1': 1247208.0,
  'L10': 78.0,
  'L2': 489852.0,
  'L3': 263568.0,
  'L4': 177216.0,
  'L5': 168384.0,
  'L6': 5407944.0,
  'L7': 469400.0,
  'L8': 21900.0,
  'L9': 2380.0,
  'O': 0},
```

**Code Snippet 4.32:** NeuronMap with balanced division of neurons Channelwise (PilotNet)

```
CorePerLayer =
{'I': 0,
  'L1': 2,
  'L10': 1,
  'L2': 1,
  'L3': 1,
  'L4': 1,
  'L5': 1,
  'L6': 6,
  'L7': 1,
  'L8': 1,
  'L9': 1,
  'O': 0})

```

**Code Snippet 4.33:** Minimum Cores per Layer (PilotNet)

### 4.3.3. Mapping compression utilities

The utility compresses the mapping so that energy consumption is saved in communication and the footprint of the area is also reduced. There are a couple of ways that mapping compression could work. Ideally, the physical structure of a chip could resemble either a square or a rectangle in 2D. Therefore, the mapping compression mechanisms implemented try to maintain the structure for easy packaging. The implementation offers the following kinds of the mapping compression mechanisms.

#### Strict area optimal mapping compression

In this mapping compression mechanism, the total number of cores is calculated by adding the total number of cores and the factors or the total cores are extracted and the factors that result in the least addition are used as rows and columns in the mesh size.

For example, if the total number of cores in mapping is 30 the factor pair would be **[(1,30), (2,15), (3,10), (5,6)]** and the pair that results in the least sum would be **(5,6)** therefore, the mesh size would be taken as **[5,6]** or **[6,5]**. For Code Snippet 4.27 compression of the network Code Snippet 4.26 would result in the mapping Code Snippet 4.34 as shown below.

```
compressedMapping =
{'I': 'CI',
'L1': ['C0_0', 'C0_1', 'C0_2'],
 'L10': ['C4_5'],
 'L2': ['C0_3', 'C0_4', 'C0_5', 'C1_0'],
 'L3': ['C1_1', 'C1_2'],
 'L4': ['C1_3', 'C1_4', 'C1_5', 'C2_0', 'C2_1', 'C2_2', 'C2_3', 'C2_4', 'C2_5'],
 'L5': ['C3_0', 'C3_1'],
 'L6': ['C3_2'],
 'L7': ['C3_3', 'C3_4', 'C3_5'],
 'L8': ['C4_0', 'C4_1'],
 'L9': ['C4_2', 'C4_3', 'C4_4']
 'O': 'CO' }
```

**Code Snippet 4.34:** NeuronMap with balanced division of neurons Channelwise (PilotNet)

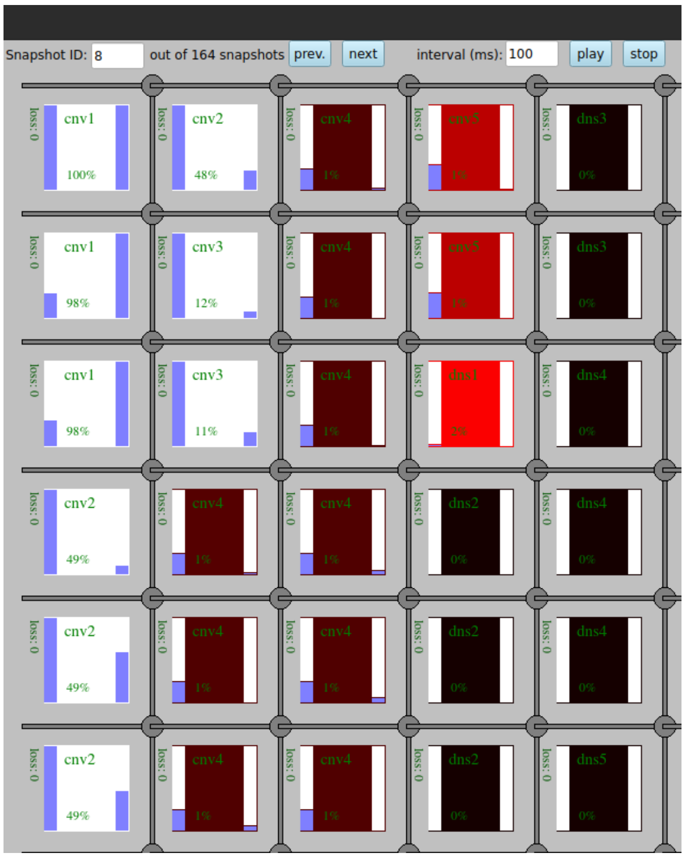


**Figure 4.12:** SENSIM GUI after network compression

**Loose area optimal mapping compression**
The drawback of having strict area optimal mapping would be when the total number of cores resulted in a prime number; for example, for 31 cores, the only possible factors would be **[(1,31)]** which would result in a chip with 31 cores aligned vertically or 31 cores aligned horizontally, which would increase the energy and latency between the cores rather than reduce it. Therefore, to keep compression flexible, a new scheme was introduced when the total number of cores resulted in a prime value, an additional core could be added, and factors would be extracted. For example, for 31 cores, an extra core would make the total cores 32 and the factors or 32 would be **[(1,32), (2,16), (4,8)]** and the chosen pair of factors would be **(4,8)** and hence the mesh size would be **[4,8]** or **[8,4]**.

**Strict square mapping compression**
While the above-mentioned compression schemes would result in a mapping that abruptly maps neurons even when the addition of cores would not result in a prime number, but only one pair of factors. For example, 26 cores would result in a mesh size of **[2,13]**. Therefore, the next compression strategy was to obtain a mesh in the shape of a square of s with the same number of columns and rows. The mesh size was calculated using the following equation.

$$mesh[rows, colums] = \lceil \sqrt{TotalCores} \rceil \tag{4.3}$$

**Flexible mapping compression**
The chip architect also has the option of giving a flexible mesh size (which could fit all cores) or a total flexible core, which the compression mechanism uses.

***Note:*** *The mapping compression mechanisms do not affect the processing of the data in the cores; therefore, the energy consumption by all cores and the processing latency are not expected to be reduced by mapping* *compression*.

### 4.3.4. Clustering utilities

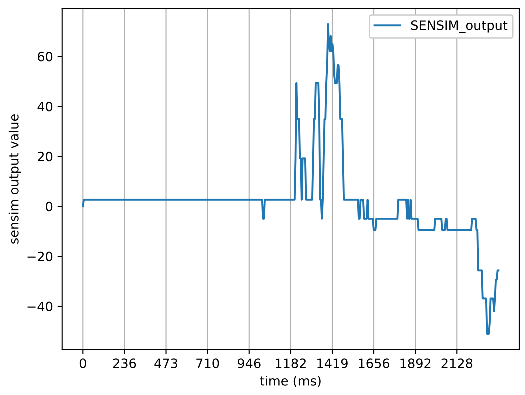
**How does clustering layers followed by partitioning layers reduce computation accuracy in SENSIM?** While combining the layers in the core is possible in SENSIM, partitioning neurons from a layer and combining layers to a similar core could land in a situation where events are updated incorrectly, resulting in incorrect output. In the layer core map Code Snippet 4.35 the layer 3 neurons are partitioned into cores located in **[2][0]** and **[2][1]** and, at the same time, layer 4 neurons are assigned to the core (**[**2] [0]). This possible situation could arise when different layers are clustered and partitioned at the same time when heuristics are used to achieve optimal mapping. Figure 4.13 shows the deviation in the SENSIM output when this configuration was tried.

```
self.layer_core_map =
{ self.layers['I']:((self.sim.CI,[range(0,66), range(0,200), range(0,3)], [0.0, 0.0]),),
# L1 : shape=[31,98,24]
self.layers['L1']:((self.sim.coresList[0][0],[range(  0, 31), range(  0, 98), range(  0, 24)
    ], [0.0, 0.0]), ),
# L2 : shape=[14,47,36]
self.layers['L2']:((self.sim.coresList[1][0],[range(  0, 14), range(  0, 47), range(  0, 36)
    ], [0.0, 0.0]), ),
# L3 : shape=[ 5,22,48]
self.layers['L3']:(
(self.sim.coresList[2][0],[range(  0,  5), range(  0, 22), range(  0, 24)], [0.0, 0.0]),
(self.sim.coresList[2][1],[range(  0,  5), range(  0, 22), range( 24,48)], [0.0, 0.0])),
# L4 : shape=[ 3,20,64]
self.layers['L4']:((self.sim.coresList[2][0],[range(  0,  3), range(  0, 20), range(  0, 64)
    ], [0.0, 0.0]), ),
# L5 : shape=[ 1,18,64]
self.layers['L5']:((self.sim.coresList[3][0],[range(  0,  1), range(  0, 18), range(  0, 64)
    ], [0.0, 0.0]), ),
# L6 : shape=[1164]
self.layers['L6']:((self.sim.coresList[3][0],[range(  0,1164)], [0.0, 0.0]), ),
# L7 : shape=[100]
self.layers['L7']:((self.sim.coresList[3][0],[range(  0, 100)], [0.0, 0.0]), ),
# L8 : shape=[50]
self.layers['L8']:((self.sim.coresList[3][0],[range(  0,  50)], [0.0, 0.0]), ),
# L9 : shape=[10]
self.layers['L9']:((self.sim.coresList[3][0],[range(  0,  10)], [0.0, 0.0]), ),
# L10: shape=[1]
self.layers['L10']:((self.sim.coresList[3][0],[range(  0, 1)], [0.0, 0.0]), ),
# L out
self.layers['O']:((self.sim.CO              ,[range(  0, 1)], [0.0, 0.0]), )}
```

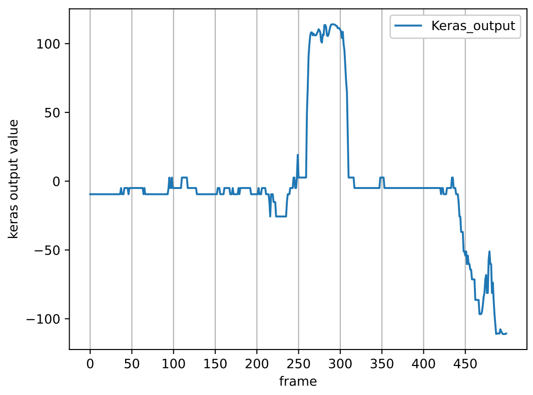
**Code Snippet 4.35:** Clustering and partitioning combined layer core map example

```
segI = interconnect(master_cores=[self.sim.CI],                 slave_cores=self.sim.
    coresList[0][0:1], name='segI')
seg1 = interconnect(master_cores=self.sim.coresList[0][0:1],    slave_cores=self.sim.
    coresList[1][0:1], name='seg1')
seg2 = interconnect(master_cores=self.sim.coresList[1][0:1],    slave_cores=self.sim.
    coresList[2][0:1], name='seg2')
seg3 = interconnect(master_cores=self.sim.coresList[2][0],    slave_cores=[self.sim.coresList
    [2][0], self.sim.coresList[3][0]], name='seg3')
seg4 = interconnect(master_cores=self.sim.coresList[3][0:1],    slave_cores=[self.sim.
    coresList[3][0], self.sim.CO ], name='seg5')
```

**Code Snippet 4.36:** Clustering and partitioning combined interconnect example



**(a)** SENSIM inaccurate output for 500 Pilot images



**(b)** Keras output for 500 Pilot images

**Figure 4.13:** Comparison of the output of SENSIM for PilotNet (500 images) with keras on clustering and partitioning layers at the same time

### Cluster lower bound memory estimation

Clustering of layers such that 2 or more layers can be clustered and area efficacy can be achieved for layers that have a small number of neurons. Initially, the lower bound was estimated on the basis of the number of cores Per Layer. After the layers were clustered, the lower bound for the optimization problem would not remain the same; therefore, the lower bound is estimated after the layers are clustered according to the constraint **Core_Dmem**. The memory calculated per core for PilotNet

```
clusterLayerMap=
{'C0_0': ['L1'],
 'C0_1': ['L1'],
 'C1_0': ['L2', 'L3'],
 'C3_0': ['L4', 'L5'],
 'C5_0': ['L6'],
 'C5_1': ['L6'],
 'C5_2': ['L6'],
 'C5_3': ['L6'],
 'C5_4': ['L6'],
 'C5_5': ['L6'],
 'C6_0': ['L7', 'L8', 'L9', 'L10']}
```

**Code Snippet 4.37:** Clustering layer core map (PilotNet)

```
clusterMemoryMap=
{'C0_0': 623604.0,
 'C0_1': 623604.0,
 'C1_0': 753420.0,
 'C3_0': 345600.0,
 'C5_0': 901324.0,
 'C5_1': 901324.0,
 'C5_2': 901324.0,
 'C5_3': 901324.0,
 'C5_4': 901324.0,
 'C5_5': 901324.0,
 'C6_0': 493758.0}
```

### clusterLayerMap to clusterNeuronMap utility

Once the lower bound of the cluster is satisfied, neurons of the particular layer need to be assigned to the core

```
clusterNeuronMap=
{'C0_0': [{'L1': [range(0, 31), range(0, 98), range(0, 12)]}],
 'C0_1': [{'L1': [range(0, 31), range(0, 98), range(12, 24)]}],
 'C1_0': [{'L2': [range(0, 14), range(0, 47), range(0, 36)]},
          {'L3': [range(0, 5), range(0, 22), range(0, 48)]}],
 'C3_0': [{'L4': [range(0, 3), range(0, 20), range(0, 64)]},
          {'L5': [range(0, 1), range(0, 18), range(0, 64)]}],
 'C5_0': [{'L6': [range(0, 194)]}],
 'C5_1': [{'L6': [range(194, 388)]}],
 'C5_2': [{'L6': [range(388, 582)]}],
 'C5_3': [{'L6': [range(582, 776)]}],
 'C5_4': [{'L6': [range(776, 970)]}],
 'C5_5': [{'L6': [range(970, 1164)]}],
 'C6_0': [{'L7': [range(0, 100)]},
          {'L8': [range(0, 50)]},
          {'L9': [range(0, 10)]},
          {'L10': [range(0, 1)]}]}
```

**Code Snippet 4.38:** Clustering and partitioning combined interconnect example

```
layerCoreMap=
{'L1': [['C0_0', [range(0, 31), range(0, 98), range(0, 12)]],
        ['C1_0', [range(0, 31), range(0, 98), range(12, 24)]]],
 'L2': [['C2_0', [range(0, 14), range(0, 47), range(0, 36)]]],
 'L3': [['C2_0', [range(0, 5), range(0, 22), range(0, 48)]]],
 'L4': [['C3_0', [range(0, 3), range(0, 20), range(0, 64)]]],
 'L5': [['C3_0', [range(0, 1), range(0, 18), range(0, 64)]]],
 'L6': [['C4_0', [range(0, 194)]],
        ['C5_0', [range(194, 388)]],
        ['C6_0', [range(388, 582)]],
        ['C7_0', [range(582, 776)]],
        ['C8_0', [range(776, 970)]],
        ['C9_0', [range(970, 1164)]]],
 'L7': [['C10_0', [range(0, 100)]]],
 'L8': [['C10_0', [range(0, 50)]]],
 'L9': [['C10_0', [range(0, 10)]]],
 'L10': [['C10_0', [range(0, 1)]]]}

```

**Code Snippet 4.39:** Clustering and partitioning combined interconnect example

### Cluster network compression

The neurons in a layer after clustering can be randomly assigned to any number of cores; hence, a network compression mechanism was also developed to optimize area efficiency and energy efficiency in segments. The cores from **LayerCoreClusterMap** are replaced with the cores placed consecutively. Similar network compression schemes as discussed in Section 4.3.4

```
layerCoreMapCompressed=
{'L1': [['C0_0', [range(0, 31), range(0, 98), range(0, 12)]],
        ['C0_2', [range(0, 31), range(0, 98), range(12, 24)]]],
 'L2': [['C0_3', [range(0, 14), range(0, 47), range(0, 36)]]],
 'L3': [['C0_3', [range(0, 5), range(0, 22), range(0, 48)]]],
 'L4': [['C1_0', [range(0, 3), range(0, 20), range(0, 64)]]],
 'L5': [['C1_0', [range(0, 1), range(0, 18), range(0, 64)]]],
 'L6': [['C1_1', [range(0, 194)]],
        ['C1_2', [range(194, 388)]],
        ['C1_3', [range(388, 582)]],
        ['C2_0', [range(582, 776)]],
        ['C2_1', [range(776, 970)]],
        ['C2_2', [range(970, 1164)]]],
 'L7': [['C0_1', [range(0, 100)]]],
 'L8': [['C0_1', [range(0, 50)]]],
 'L9': [['C0_1', [range(0, 10)]]],
 'L10': [['C0_1', [range(0, 1)]]]}
```

**Code Snippet 4.40:** Clustering and partitioning combined interconnect example

### Cluster partitioning (equal neuron division)

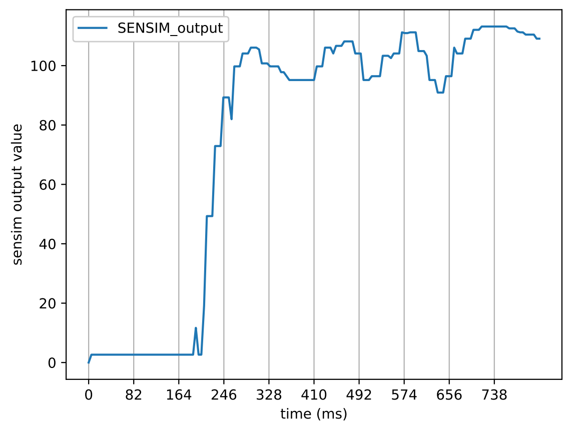
With the clustering of neurons of different layers, the lower limit is set by how many clusters can be formed as variables in the optimization problem and by which layers can be clustered together based on the physical constraints of the chip. For a design space exploration of the newly formed layer clustered problem where each cluster gets a core on the chip, there can still be a possibility that for energy efficiency more cores are needed.

Hence, a cluster partitioning problem was developed in which in a given cluster neurons would be equally divided for the given number of cores per cluster. For example, Code Snippet 4.41 the neurons in Code Snippet 4.40 are equally divided and new **layerCoreMap** were created. The lower limit was set to be 1 core per cluster, but the upper limit per cluster can be decided by the architect. The **divideArray** decides the partition of neurons in different clusters. To maintain simplicity in the system, the neurons for any given layer are divided equally. There can be a few more ways of partitioning the neurons with clustering combined, but adding them would further complicate the problem. Therefore, for simplicity, equal partitioning after clustering is implemented.

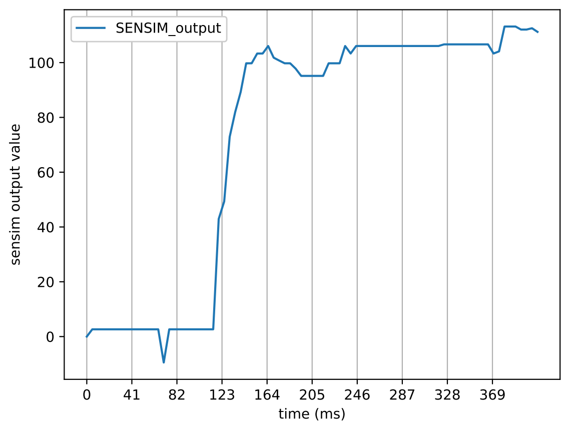
```
divideArray= [4,2,1,2,3,1,2,1,2,4,2]
layerCoreMap =
{'L1': [['C0_0', [range(0, 31), range(0, 98), range(0, 3)]],
        ['C0_1', [range(0, 31), range(0, 98), range(3, 6)]],
        ['C0_2', [range(0, 31), range(0, 98), range(6, 9)]],
        ['C0_3', [range(0, 31), range(0, 98), range(9, 12)]],
        ['C1_0', [range(0, 31), range(0, 98), range(12, 18)]],
        ['C1_1', [range(0, 31), range(0, 98), range(18, 24)]]],
 'L2': [['C2_0', [range(0, 14), range(0, 47), range(0, 36)]]],
 'L3': [['C2_0', [range(0, 5), range(0, 22), range(0, 48)]]],
 'L4': [['C3_0', [range(0, 3), range(0, 20), range(0, 32)]],
        ['C3_1', [range(0, 3), range(0, 20), range(32, 64)]]],
 'L5': [['C3_0', [range(0, 1), range(0, 18), range(0, 32)]],
        ['C3_1', [range(0, 1), range(0, 18), range(32, 64)]]],
 'L6': [['C4_0', [range(0, 65)]],
        ['C4_1', [range(65, 130)]],
        ['C4_2', [range(130, 194)]],
        ['C5_0', [range(194, 388)]],
        ['C6_0', [range(388, 485)]],
        ['C6_1', [range(485, 582)]],
        ['C7_0', [range(582, 776)]],
        ['C8_0', [range(776, 873)]],
        ['C8_1', [range(873, 970)]],
        ['C9_0', [range(970, 1019)]],
        ['C9_1', [range(1019, 1068)]],
        ['C9_2', [range(1068, 1116)]],
        ['C9_3', [range(1116, 1164)]]],
 'L7': [['C10_0', [range(0, 50)]], ['C10_1', [range(50, 100)]]],
 'L8': [['C10_0', [range(0, 25)]], ['C10_1', [range(25, 50)]]],
 'L9': [['C10_0', [range(0, 5)]], ['C10_1', [range(5, 10)]]],
 'L10': [['C10_0', [range(0, 1)]]]}

```

**Code Snippet 4.41:** Clustering and partitioning combined interconnect example

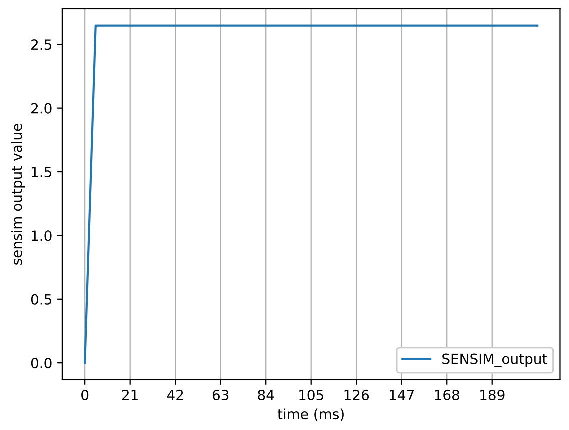
The clustering of layers can affect the accuracy of the system in the case of an event-based system.



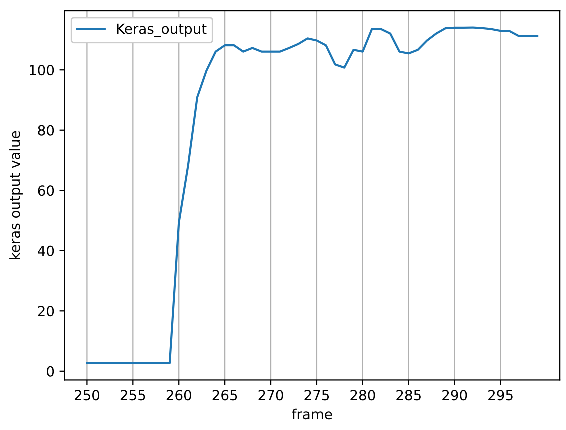
**(a)** SENSIM output with processing rate 60 fps



**(b)** SENSIM output with processing rate 120 fps



**(c)** SENSIM output with processing rate 240 fps



**(d)** Keras output

**Figure 4.14:** PilotNet output comparison keras for different rate

## 4.4. SENMap optimization

SENMap uses meta-heuristics to get to a mapping that results in a minimum mapping. To use the meta-heuristic algorithms efficiently, the optimization strategy must be decided. Section 4.4.1 details the decision making for optimization

### 4.4.1. Optimization strategies and problem formation

Once the partition of neurons in the neural network is complete, solving the optimization problem is the next step involved in obtaining efficient mapping. Forming the optimization problem is the most important step when solving the mapping problem because if the optimization problem is formed incorrectly, the time to solution for the problem can take either more time to converge or not converge to the global minimum within a given time frame. The formation of optimization problems was formed on the basis of the aspects mentioned in the various optimization problems Section 3.2.2.

#### Variable for the problem

To make sure the convergence is much faster, the variables for the problem were chosen in such a way that the iterative algorithm has to choose from a smaller set of combinations, but also not so small that the architect cannot make the correct decisions. Below are the options on which the variable for the optimization problem could be decided.

- **P** Divide the total number of neurons in a layer, divide them into cores, and make the total cores a single variable.

- **P** Divide the total number of neurons in a layer and make the number of neurons per core as variable.
- **P** Divide the total number of neurons per layer equally among the cores so that all cores have almost the same number of neurons per layer and keep the cores per layer as the variable.
- **CP** Cluster the neuron from adjacent layers and partition the clustered neurons into given number of cores.

The drawback in the first scheme is that the variable on which the optimization algorithm has to operate is comparatively small, and it would take more computational cycles for the optimizers to learn how the total number of cores results in a minimum energy. Therefore, the first strategy was rejected.

The second strategy suggests getting the set of neurons per core for all cores. The cores in a scalable neuromorphic processor can scale up, and the number of cores the optimizer will have to evaluate would increase, and the number of neurons to fit in a core would create a computationally intense problem. Therefore, the second strategy was avoided. If a problem needs to be solved in a manner that each neuron is evaluated, might give the exact neuron mapping coming to a possibly more energy-efficient solution, but the computational time outweighs the benefits of having a numerically intense problem.

The third strategy suggests that neurons are equally divided between the cores. For example, there are 10 neurons in a layer and 5 cores available for the layer; each core gets 2 neurons from the layer to process. With this strategy, the CPU utilization of the core is balanced and the workload is shared among the cores. Also, by doing so, the variable limit would be the number of cores per layer at max, which cannot exceed the number of cores on the chip. Ideally, a large-scale neural network is expected to have more than one layer, so the limit would be further restricted. The total number of variables with which the optimizer will operate will be equivalent to the number of layers in the neural network.

The last strategy would improve the optimal area of the whole system by reducing the number of cores in general in the system. ***Note:*** *The current version of SENSIM software does not fully support layer combining without loss of accuracy at the end signal at higher rates; therefore, the combination of cores per layer as variables for the optimizer is the ideal choice. Once SENSIM supports the combination of layers in SENSIM, the optimization problems would change and therefore the choice of variables taken as cores per layer would be clustered per core.*

After choosing the variable type, the division of the neuron strategy by layer type would change. Since SENSIM allows flexibility to work with different types of layers (for example, dense and convoluted layers) when composing layers as shown in Section 2.3.3, SENMap is currently supported for the division of neurons into dense and convoluted layers. For dense layers, the division of neurons is rather simple when done equally, compared to the division of neurons in convolution layers, since they have only one dimension, which is the channel. For convolution layers, the division can be height, width, or channel. The choice of the channel-wise division was promoted over the division of neurons by height or width to maximize the use of data memory for weight, bias, and threshold storage for any processor. Dividing the height and width of neurons would require the processor to store the complete set of weights, biases, and thresholds for any layer, as the receptive field varies in dimension depending on the pool and stride length, which makes it rather complicated and memory intensive.

**Bounds for the problem**

After choosing the variables, the limits of these variables are important to restrict the search space. These variable bounds could be taken from physical restrictions by making reasonable architectural choices. For example, a bound to the number of cores per layer could be the total number of neuromorphic cores in the chip or neuromorphic cores on the 1 axis in a square mesh matrix.

While the upper boundaries can be kept scalable, the lower boundaries could be set by SENSIMs execution model restrictions of at least giving one core to be assigned per layer.

It is impossible to fit neurons into data memory beyond the limitations of memory; hence, the lower boundaries for the variable would be based on the memory density of the layer and how many cores would at least be required to fit the neurons of the layer onto the core. In such a situation, the assumption is taken that shared memory is not used and all weights, biases, and thresholds would be stored in the local memory of the neuromorphic processor.

The assumption taken here that neurons, weights, bias, and thresholds are stored in local memory and not shared memory helps reduce energy and latency.

On the basis of the physical restrictions and assumptions mentioned above, the lower and upper bounds were decided and various experiments were conducted. As the variables in the optimization problem changed, the lower and upper bounds of the problem would also change.

**Constraints for the problem**
While bounds for the problem add restrictions to the variables chosen, there could be other constraints to the problem that could be added according to the architect. The following are some possible constraints that the architect may add. An architect could possibly add constraints on time, energy, total cores, and much more.

**Objectives for the problem**
The main objective was to minimize energy and latency, but various time-objective functions can be formed with the support of the multi-objective optimization problem, and different objectives could also be combined to form a multi-objective optimization problem.

### 4.4.2. Types of optimization problem

To obtain an optimal mapping based on different objectives and constraints, variables, and constraints to construct the optimization problem.

Equation 4.4 expresses the optimization problem to minimize latency for the i layers (in the case of PilotNet layers = 10) assigned to the neuromorphic processor. $xL_m$ is the lower bound, which in this case is the optimal memory mapping. $xU_m$ is the upper bound, which in this case is the row or column of the mesh.

$$\begin{aligned} \text{min} \quad & f_{latency}(x_i) && i = 1, 2, ..layers \\ \text{s.t.} \quad & xL_m \leq x_m \leq xU_m && m = 1, ., layers \\ & x \in I \\ & xL \leq min(memOpt, channelNeurons) \\ & xU \leq row \times column \end{aligned} \tag{4.4}$$

Equation 4.5 expresses the optimization problem to minimize the energy of i layers (in the case of PilotNet - layers = 10) mapped in the neuromorphic processor. $xL_m$ is the lower bound, which in this case is the optimal memory mapping. $xU_m$ is the upper bound, which in this case is the row or column of the mesh.

$$\begin{aligned} \text{min} \quad & f_{energy}(x_i) && i = 1, 2, ..layers \\ \text{s.t.} \quad & xL_m \leq x_m \leq xU_m && m = 1, ., layers \\ & x \in I \\ & xL \leq min(memOpt, channelNeurons) \\ & xU \leq row \times column \end{aligned} \tag{4.5}$$

Equation 4.6 expresses the optimization problem that minimizes the energy of the i layers (in the case of PilotNet layers = 10) mapped on the neuromorphic processor. $xL_m$ is the lower bound, which in this case is the optimal memory mapping. $xU_m$ is the upper bound, which in this case is the row or column of the mesh.

$$\begin{array}{rll} \text{min} & f_{energy}(x_i) & i = 1, 2, ..layers \\ \text{s.t.} & sum(x_i) \leq total\_cores & I = 1, .., layers \\ & xL_m \leq x_m \leq xU_m & m = 1, ., layers \\ & x \in I & \\ & xL \leq min(memOpt, channelNeurons) & \\ & xU \leq row \times column & \end{array} \tag{4.6}$$

Equation 4.7 expresses the optimization problem that minimizes the energy and latency of the i-layers (in the case of PilotNet layers = 10) mapped in the neuromorphic processor. $xL_m$ is the lower bound, which in this case is the optimal memory mapping. $xU_m$ is the upper bound, which in this case is the row or column of the mesh.

$$\begin{array}{rll} \text{min} & f_{energy}(x_i) & i = 1, 2, ..layers \\ \text{min} & f_{latency}(x_j) & j = 1, 2, ..layers \\ \text{s.t.} & xL_m \leq x_m \leq xU_m & m = 1, ., layers \\ & x \in I & \\ & xL \leq min(memOpt, channelNeurons) & \\ & xU \leq row \times column & \end{array} \tag{4.7}$$

Equation 4.8 expresses the optimization problem that minimizes the energy, latency, and total neuromorphic cores for i-layers (in the case of PilotNet layers = 10) mapped in the neuromorphic processor. $xL_m$ is the lower bound, which in this case is the optimal memory mapping. $xU_m$ is the upper bound, which in this case is the row or column of the mesh.

$$\begin{array}{rll} \text{min} & f_{energy}(x_i) & i = 1, 2, ..layers \\ \text{min} & f_{latency}(x_j) & j = 1, 2, ..layers \\ \text{min} & f_{cores}(x_k) & k = 1, 2...layers \\ \text{s.t.} & xL_m \leq x_m \leq xU_m & m = 1, ., layers \\ & x \in I & \\ & xL \leq min(memOpt, channelNeurons) & \\ & xU \leq row \times column & \end{array} \tag{4.8}$$

Equation 4.9 expresses the optimization problem that minimizes the energy and total neuromorphic cores for i-clusters (in the case of PilotNet, the clusters formed are 11 clusters in a case where the core Data Memory is equal to 1 Mb and neurons from the last layer are combined) mapped in the neuromorphic processor. $xL_m$ is the lower bound, which in this case is 1 core per cluster. $xU_m$ is the upper limit on which the neuromorphic architect can decide. The output error can be

$$\begin{array}{rll} \text{min} & f_{energy}(x_i) & i = 1, 2, ..clusters \\ \text{min} & f_{cores}(x_k) & k = 1, 2...clusters \\ \text{s.t.} & error \leq threshold & \\ & xL_m \leq x_m \leq xU_m & m = 1, ., layers \\ & x \in I & \\ & xL \leq 1 & \\ & xU \leq coresPerCluster & \end{array} \tag{4.9}$$

To solve optimization problems, optimization algorithms suggested in Chapter 3 could be used. For experimentation purposes, the algorithm GA for a single objective and NSGA2 for a multi-objective were used with the following fixed parameters.

**Basic algorithmic parameters used for experimentation**
The experiment uses the basic genetic algorithm (GA) version with the following algorithmic parameters.

| Algorithmic parameter (GA) | Value |
|---|---|
| population size | 30 |
| mutation type | integer polynomial mutation (eta=3.0) |
| crossover type | integer SBX (eta=3.0) |
| sampling type | integer (random selection) |

The experiment uses the basic version of the NSGA2 with the following algorithmic parameters.

| Algorithmic parameter (NSGA2) | Value |
|---|---|
| population size | 40 |
| number of off-springs | 10 |
| mutation type | integer polynomial mutation (eta=3.0) |
| crossover type | integer SBX (eta=3.0) |
| sampling type | integer (random selection) |

## 4.5. SENMap framework enhancements

To improve the SENMap framework, there were a couple of additions that improved the flexibility of the framework, improved the computation speed, and gave flexibility to the chip architect to vary parameters and perform experiments.

### 4.5.1. Flexible application replacement

A neuromorphic chip can be designed to work with various applications; therefore, SENMap must be designed in such a way that it supports different applications by making minor changes.

The utilities inherit the application definition class, where the simulator parameters, the composition of the layer, the composition of the layer core map and the interconnect as defined in Section 4.1 are defined, causing the utilities to use them as wrappers for any application class it inherits. Based on the application the input dataset and the output captured would also vary.

```python
# for PilotNet Mapping
from appdefs import PilotNet_app as App
from iodefs import PilotNet_IO as IO
# for Mnist Mapping
from appdefs import mnist_App as App
from iodefs import minist_IO as IO

# Class defination of SENMap Utilities
class mapperUtility(App):
```

**Code Snippet 4.42:** Application replacement in SENMap

To ensure consistency between different applications, the parameters used to define the problem Table 3.1 are generated using the application parameters of the neural network.

### 4.5.2. Parallel processing

Parallel metaheuristics are popular techniques developed in parallel programming to run multiple metaheuristic searches in parallel; these can range from simple distributed schemes to concurrent search runs that interact to improve the overall solution. Parallel metaheuristics works well with population-based algorithms like particle swarm optimization or evolutionary algorithms instead of local search algorithms like Nelder Mead. Integrating the Pymoo framework gave the flexibility to use the multiprocessing Python standard library and Dask (distributed framework). To process 1 iteration of SENSIM with 2000 images, the computational time exceeds 1 hour. Executing various mapping schemes in parallel in a generation reduces the total time by approximately the factor in which parallelization is implemented. For example, if a compute node has 30 cores for parallel processing, a genetic algorithm will compute its 1 generation with 30 genomes in parallel considering that there is enough memory ($\sim$ 6GB)for each thread.

**Multiprocessing Python standard library**
SENMap integrates Pymoo, which allows parallelization with the Python multiprocessing standard library, which allows passing a starmap object to be used for parallelization. The starmap interface is defined in the Python standard library function *multiprocessing.Pool.starmap*. Serialization delay might become an overhead in the event that the problem is not computationally expensive; multiprocessing is recommended if the problem is computationally expensive. The multiprocessing library gives the option to parallelize by using threads or processes. While threads are input-output bound and processes are more flexible, threading is much more efficient, and Pymoo supports threading with most of its algorithms; hence, to speed up the computation time, multithreading was used.

**Distributed framework Dask**
For SENMap to support more dense and large applications with a computationally expensive data set to process and simulate, the framework also supports easy integration with the advanced distributed framework Dask. Dask.distributed is a lightweight library for distributed computing in Python. It extends both the concurrent features and Dask APIs to moderate-sized server clusters.

***Note:*** *The Dask framework was not used in the experimentation, but could be used in the event that the computational workload on each compute server in a cluster of servers is higher than the time it takes to copy the data between these servers*

### 4.5.3. Experimentation framework

The experimentation framework was required to obtain an in-depth analysis of several iterations of SENSIM with different mappings proposed by the mapper. The experimentation framework helps maintain a database that can be used later for chip design analysis. The framework can also be used later for the SENMap GUI for post-optimization analysis. The framework considers the future development of algorithms based on hyperheuristics, such as machine learning-based mappers or online learning maps for better performance.

Below is an example of how the experimentation directory expands. A time-stamp-based framework helps the researcher identify when the experiment was performed and also helps with easy data analysis. The framework begins with the first level of division of the application, which needs to be assigned to SENSIM. The second level of division is based on the algorithm in use, followed by additional levels depending on whether the problem is a single-objective optimization problem or a multiple-objective optimization problem. For an optimization problem with various objectives, the directory is divided by the objectives to be achieved. In the following example, for the NSGA2 algorithm, the objectives chosen were energy minimization and latency minimization. For every objective, the algorithm takes certain iterative steps to reach the most optimal point. Recording these iterations helps in multiple ways, including understanding why certain inefficient mappings did not map well to SENSIM and creating a database to model SENSIM. Once the experimentation is complete, the framework stores certain files that could give a summary of how the experimentation works. In the case of population-based optimizers, such as particle swarm optimizers and evolution-based optimizers, which can evaluate candidates in parallel, the framework adds an identity to the time-stamped iteration folder, which in the case of multithreading is a thread ID. The identifier can vary according to the parallelism added to the optimizer. Table 4.1 details the contents of the experimentation summary folder. Table 4.2 details the contents of the iteration folder. Figure 4.15 is an example of the structure of the experimentation folder.

| Snapshot header | Description |
|---|---|
| algo.prm | The file contains the algorithm name and all the algorithmic hyperparameters the experiment is tuned to. |
| BaselineRunDebug.txt | The file contains the output of SENSIM debugs when turned on |
| engOpt.csv | The file contains input variables chosen for a problem (in our case the cores per layer) used for energy optimization. |
| latOpt.csv | The Ifile contains input variables chosen for a problem (in our case the cores per layer) used for latency optimization and the output variables for every iteration |
| gui_setting.csv | The setting file for a GUI initialization |
| output_sensim.eps | The output of the Neural Network on the execution of data on SENSIM |
| output_sensim_keras.eps | The output values of keras predicted values for post inference analysis and comparison of SENSIM output |
| output_snapshot.csv | The file captures the output of SENSIM at regular intervals and stores |
| results.txt | Additional analytics offered by Pymoo |
| sim.prm | The file contains simulation parameters (technology, architectural) |
| snapshots_cores.csv | The file contains the snapshot of the processor taken at different intervals |
| snapshots_interconnects.csv | The file contains snapshots of various interconnect parameters at different intervals |
| plot_cores | The folder contains plots of energy dissipation, processor utilization, etc of the core for the baseline Run |
| plot_interconnect | The folder contains the plots of energy dissipation of the interconnect for the baseline Run |

**Table 4.1:** Contents of experimentation summary folder

| File name | Description |
|---|---|
| gui_setting.csv | The setting file for a GUI initialization |
| output_sensim.eps | The output of the Neural Network on the execution of data on SENSIM |
| output_sensim_keras.eps | The output values of keras predicted values for post inference analysis and comparison of SENSIM output |
| output_snapshot.csv | The file captures the output of SENSIM at regular intervals and stores |
| snapshots_cores.csv | The file contains the snapshot of the processor taken at different intervals |
| snapshots_interconnects.csv | The file contains snapshots of various interconnect parameters at different intervals |
| plot_cores | The folder contains plots of energy dissipation, processor utilization, etc of the core |
| plot_interconnect | The folder contains the plots of energy dissipation of the interconnect. |

**Table 4.2:** Contents of iteration folder

```
experiments/
├── Mnist_app/
└── PilotNet_app/
    ├── BRKGA_0_2022_07_23_23:22:35/
    ├── DE_0_2022_07_23_23:32:46/
    ├── ES_0_2022_07_23_23:32:20/
    ├── NSGA2_1_2022_08_05_14:20:58/
    │   ├── Energy/
    │   │   ├── 2022_08_05_14:21:44_140477164676864
    │   │   │   ├── gui_setting.csv
    │   │   │   ├── output_snapshot.csv
    │   │   │   ├── snapshots_cores.csv
    │   │   │   ├── snapshots_interconnects.csv
    │   │   │   └── summary.txt
    │   │   └── 2022_08_05_14:21:44
    │   ├── Latency/
    │   └── NSGA2_summary_2022_08_05_14:20:58/
    │       ├── algo.prm
    │       ├── BaselineRunDebug.txt
    │       ├── engOpt.csv
    │       ├── latOpt.csv
    │       ├── gui_setting.csv
    │       ├── output_sensim.eps
    │       ├── output_sensim_keras.eps
    │       ├── output_snapshot.csv
    │       ├── results.txt
    │       ├── sim.prm
    │       ├── snapshots_cores.csv
    │       ├── snapshots_interconnects.csv
    │       ├── plot_cores
    │       └── plot_interconnect
    └── PSO_0_2022_08_01_05:28:39
        ├── 2022_08_01_06:59:13_140477164676764
        ├── 2022_08_01_12:43:14_140477164676064
        └── PSO_summary_2022_08_01_05:28:39
```

**Figure 4.15:** Experimentation file structure for SENMap

### 4.5.4. Flexible architectural and algorithmic parameters

**Optimization with architectural parameters**

The simulator takes into account the architectural, simulation, and technology parameters, which can be varied as shown in Section 2.3.4. Although some of these parameters are extracted from the simulation of the technology node, some architectural parameters, such as **N_NPE** (number of neural processing elements per core), **Core_DMEM** (data memory per neuromorphic core), can be varied in experimentation. Some of these architectural parameters can also be made variables in an optimization problem. For example, the number of neural processing elements **N_NPE** could be varied in the optimization problem with the number of cores per layer and the most optimal neuromorphic chip could be manufactured with the exact number of neural processing elements in a processing core. Adding **Core_DMEM** to the optimization problem could give a better indication of what memory technology and size would be the best for a given application.

```
python senMap_v1.py -a GA -s 400 -n 16 -m 1024*1024 -t 10 -o latency &
python senMap_v1.py -a GA -s 400 -n 32 -m 1024*1024 -t 10 -o energy &
python senMap_v1.py -a GA -s 400 -n 8 -m 1024*1024 -t 10 -o energy-latency &
```

**Code Snippet 4.43:** Example for varying the (-n) number of NPE per core in SENMap

**Algorithmic choice and hyperparameter tuning**

Since Pymoo provides a great deal of flexibility in terms of choosing hyperparameters for every algorithm and performing computationally better for every problem, the hyperparameters need to be tuned. SENMap also takes these algorithm hyperparameters as arguments and saves them in **algo.prm** for the reproducibility of results. The experimentation does not take stochastic optimization into account.

## 4.6. Post analysis framework

A post-experimentation analysis framework was also developed to speed up the process of understanding the results gathered. All the parameters for the experimentation are the same as described in Chapter 5. The results are for 0-10 images for PilotNet.

### 4.6.1. Energy-Latency correlation plots for different NPE's

Although by the laws of physics, the calculation of the energy of the system is done by calculating the energy per processor node by giving different read and write values from a memory, and the system time is estimated by adding time units as the data pass through the neuromorphic processing system. Therefore, these graphs help to understand the correlation between the two parameters as the algorithms optimize the mapping.

$$energy = power \times time \tag{4.10}$$

For every experiment and optimization strategy (in this case, energy and latency), the correlation for energy and latency was observed.

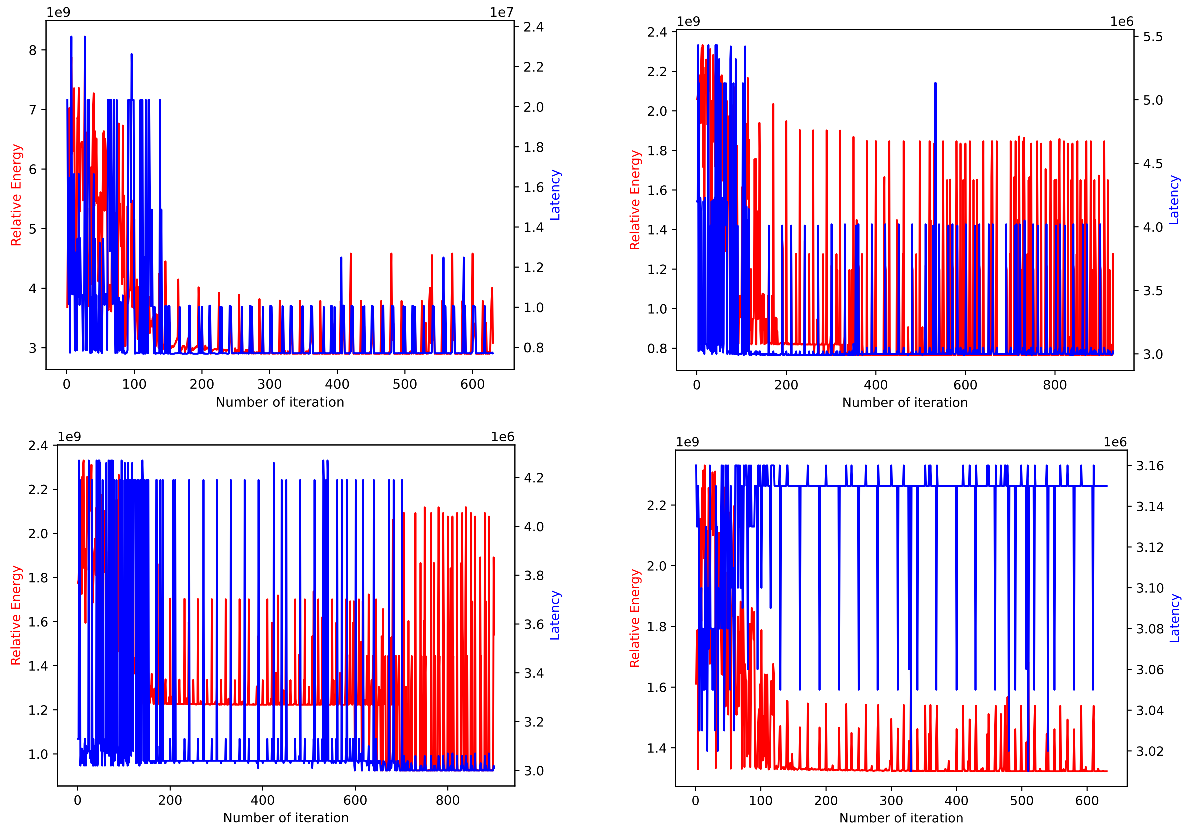


**Figure 4.16:** Energy Latency correlation plots with latency optimization for different number of NPE's

### 4.6.2. Energy-Mapping and Latency-Mapping correlation plots for different NPE's

By looking at the trend in the reduction of energy/latency of bar graphs, one can come to a conclusion about how the mapping changes with the reduction in energy. One can also observe that changing the cores in the initial layers affects the energy to a greater extent as compared to the cores from the last layer, which is probably because of the number of reads and writes in the initial layer and the operation density of the initial convolution layers is higher in case of PilotNet which results in more cores for lower NPE values.

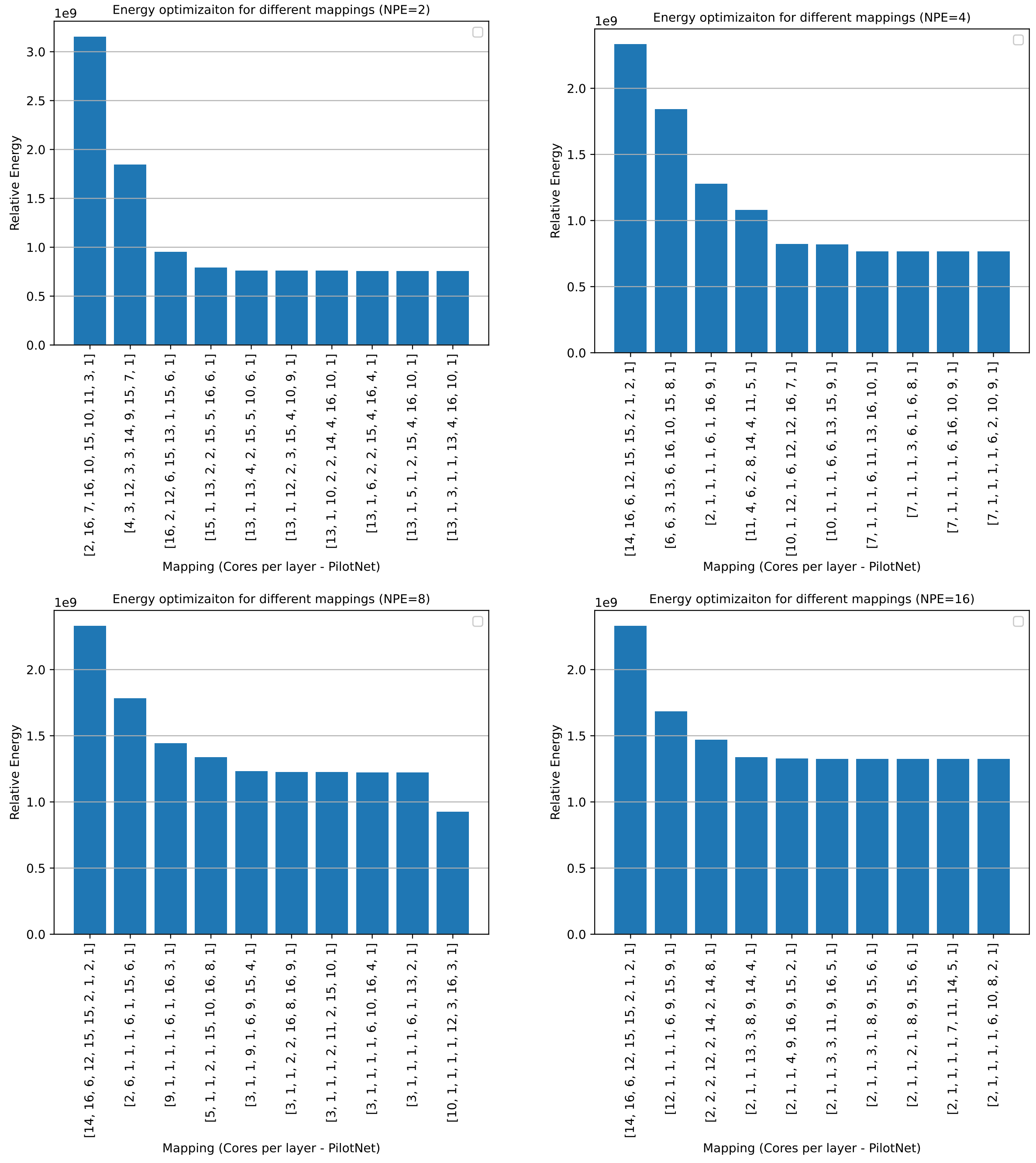


**Figure 4.17:** Energy optimization for different mappings for different NPE's

### 4.6.3. Baseline, maximum, and minimum comparison plots

By looking at the trend of the baseline, maximum, and minimum for different architectural parameters it was easy to compare the reduction of energy Figure 4.18 provides a brief description of how the cores per layer vary with the energy for different NPE values. The other objectives like latency and area were also analyzed in a similar way.

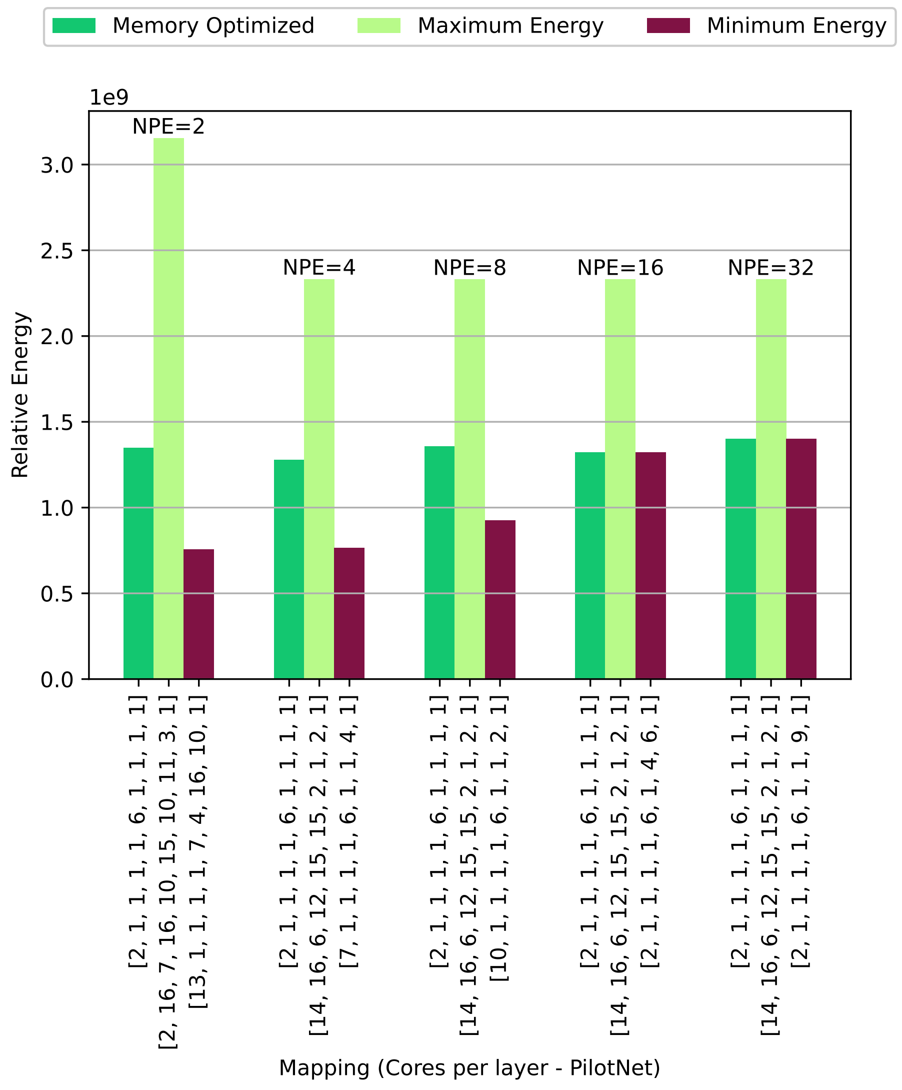


**Figure 4.18:** Baseline maximum and minimum energy comparison with different optimal mappings for FPS=0 video stream=(0 10 images) PilotNet

## 4.7. Conclusion

### Evaluation of SENMap

- Accuracy measurement in the current version of SENSIM and SENMap is performed by visual inspection of the application output. A numerical measure would help to better understand how much the deviation is for any given mapping or architectural change. The accuracy parameter can be used in the hardware-aware closed-loop simulator while changing architectural parameters. A suggested method would measure the accuracy of a signal through simple signal correlation techniques.
- SENSIM in a single iteration uses around $6 \sim 8$ GB of memory, SENMap uses population-based algorithms that execute SENSIM iterations in parallel, increasing RAM use to a great extent. SENMap is not a lightweight software.
- The time to solution depends on the computational resources available to the designer, the application (SNN/DNN), the optimization strategy, and the hyper-parameters.
- Although there is a significant improvement in energy and latency when partitioning neurons in the core,

there remains more room for improvement if the clustering of multiple shallow layers into a single core could be made feasible, and with clustering and partitioning, there remains further room for improvement in any particular rate. The accuracy of the mapper needs to be kept in the loop at higher rates.

- A combined approach where clustering and partitioning are possible would be the ideal hardware mapper to attain maximum improvements for any given objective (energy, latency, area, accuracy) for any given rates.
- There is a minor error in the memory calculation. The memory required would be twice for the neuron states (since both the old and new states of the neuron need to be saved in the memory location). The result and the mapping will not be affected to a great extent. In case the weights or neurons are stored in the external memory, the CMR (Cache Miss Ratio) factor needs to be added to the equation as well. In the case of PilotNet and MNIST, both neurons and weights were in local memory, and therefore the CMR factor was not added. The new estimate of memory would be Equation 4.11.

$$\begin{aligned} M_{pc} = &(N_{npc} + (N_{tpc} * F_{snn}) \times (2 \times BW\_States + BW\_Outputs))(1 - CMR(N)) + \\ &(N_{wpc} + N_{bpc}) \times BW\_weights(1 - CMR(W)) \end{aligned} \tag{4.11}$$

where:

| | |
|---|---|
| $N_{npc}$ | $=$ number of neurons per core |
| $N_{wpc}$ | $=$ number of weights per core |
| $N_{bpc}$ | $=$ number of biases per core |
| $N_{npc}$ | $=$ number of thresholds per core |
| $M_{pl}$ | $=$ memory utilization per layer |
| $M_{pc}$ | $=$ memory utilization per core |
| $F_{snn}$ | $=$ a boolean flag which is taken in true value in case of an SNN |
| $x$ | $=$ a particular layer in the network |
| $y$ | $=$ cores assigned to a particular layer |

# 5 Experimentation and results

This chapter covers several experiments performed with SENMap designed in Chapter 4 meant to validate the hypotheses discussed in the thesis. In the following subsections, all experiments were carried out by varying various architectural and rate-based parameters for PilotNet SNN and 4-layered DNN trained to understand the SNN/DNN mapping and its correlation with SENeCA. These experiments were carried out using different sizes of PilotNet data sets (ranging from 10 to 30 images). It was noted that in our case, the relative energy efficiency and the size of the data set is not very influential. The same can also be justified with the observation by performing several other experiments where the PilotNet data set was varied with 10, 50, 100 images, which showed that the mapping was not affected.

## 5.1. Experiments

### 5.1.1. Experiment 1: Relative energy optimization for different number of NPEs (video stream 0-10 images) fps=0

Figure 5.1 shows the results of an experiment measuring the relative energy consumption simulated using SENSIM for the first 10 images of PilotNet. The y-axis of the figure shows the relative energy consumption of all 10 images. The x-axis shows how the mapping (number of cores per PilotNet layer) is done using various SENSIM configurations while changing the number of NPEs active per core (ranging from 2 to 32 NPEs). For each of these configurations, three different measurements are taken, one that shows the minimum, maximum, and memory-optimized energy consumption.

Inspection of the minimum energy bar in the figure shows that reducing the number of NPEs used per core results in lower total energy consumption. This is caused by having fewer NPEs giving more space to use more cores in the system, which allows us to process many parts of an image in parallel. The energy consumption for memory-optimized mappings is rather the same. The maximum energy consumption also remains the same for 4 to 32 NPEs per core and only increases for the case of NPE=2.

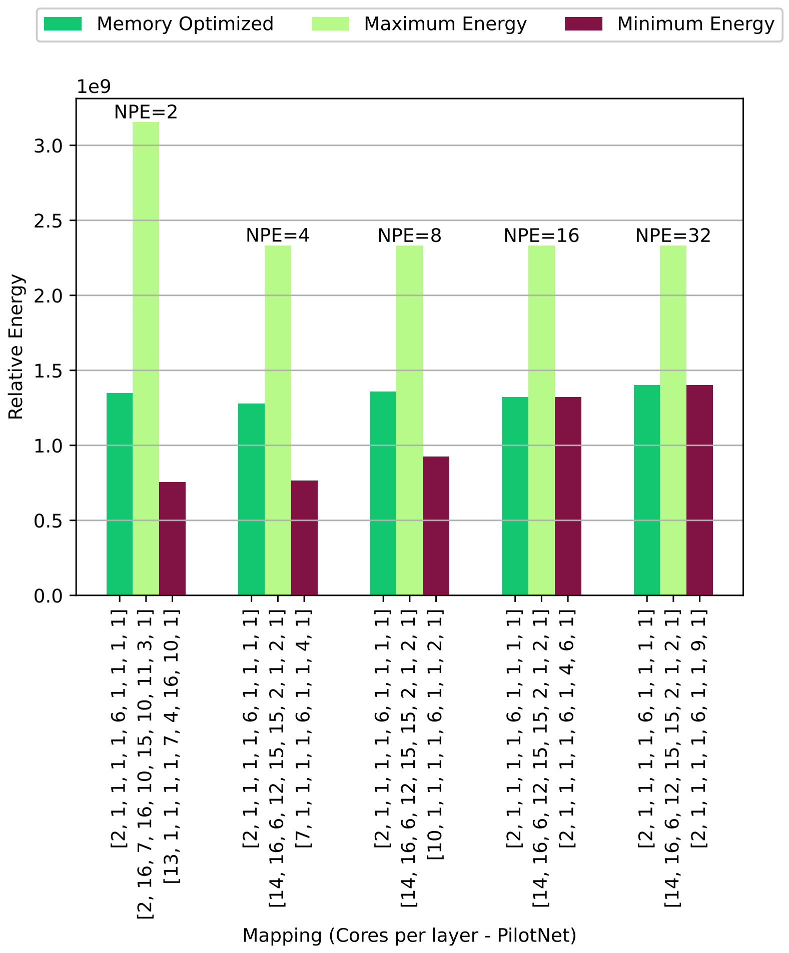


**Figure 5.1:** Relative total energy vs mapping (Cores per Layer) for PilotNet simulated for 0-10 images

### 5.1.2. Experiment 2: Relative energy optimization for different numbers of NPEs (video stream 0-20 images) fps=0

Figure 5.2 shows that the simulation results are derived with 0-20 images and a frame rate of 0 (event-based processing). The NPEs as architectural parameters were varied along with the mapping (core per layer) in PilotNet, and a comparison of the optimized memory, the maximum energy, and the minimum energy configuration was mademade. The bars show the relative energy of the different mappings in a given experiment for different numbers of NPEs.

For the minimum energy case, the figure shows that reducing the number of NPEs used per core results in a lower total energy consumption. In contrast, the energy consumption for memory-optimized mappings stays rather the same. The maximum energy consumption also stays the same for 4 to 16 NPEs per core and only increases in the case of NPE=2.

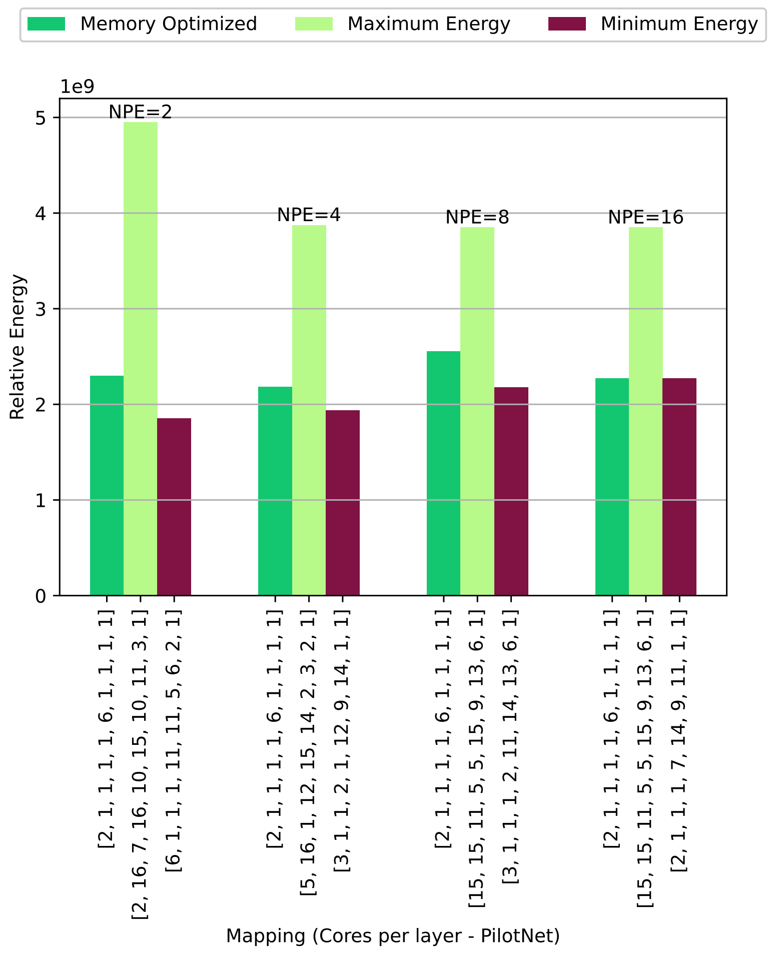


**Figure 5.2:** Relative Energy vs Mapping comparison for PilotNet simulated for 0-20 images

### 5.1.3. Experiment 3: Relative energy optimization for a less sparse segment varying the NPEs (video stream 260-280 images) fps=0

Figure 5.3 shows the relative energy consumption for the simulations of the mappings of 260-280 images. In the given experiment, the NPEs as architectural parameters were varied along with the mapping (core per layer) in PilotNet and a comparison of the optimized memory configuration, maximum energy, and minimum energy was performed. The bars show the relative energy of the different mappings in a given experiment for different numbers of NPEs per core.

For both the minimum-energy case and the memory-optimized case, the figure shows that reducing the number of NPEs used per core results in a lower total energy consumption. In contrast, the energy consumption for memory-optimized mappings increases with decreasing number of NPEs per core.

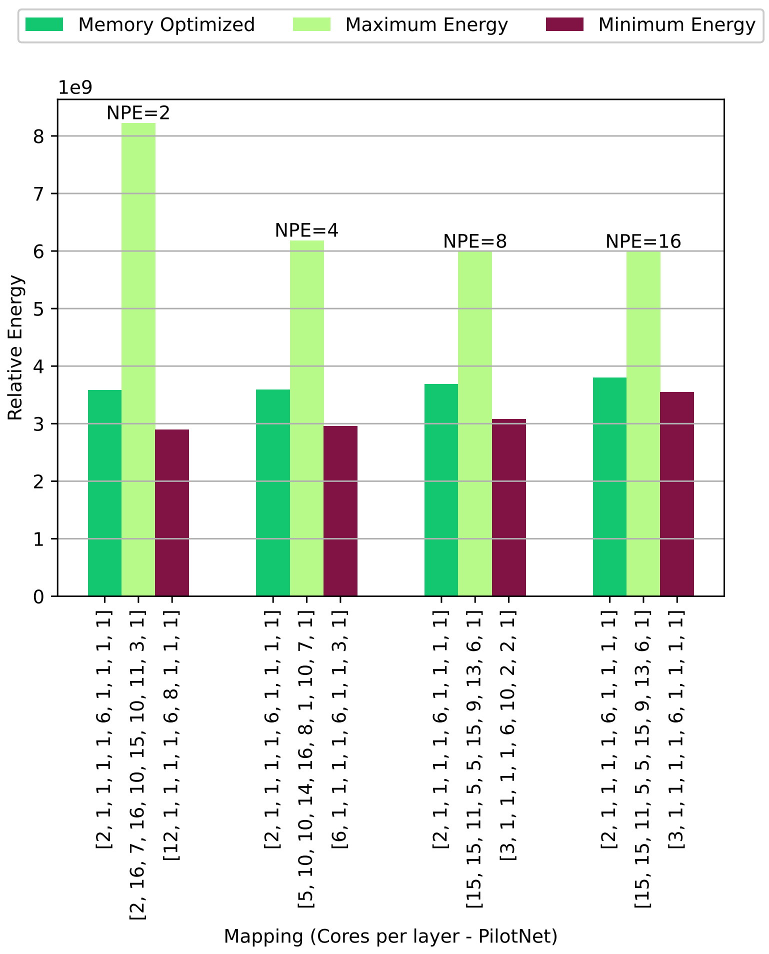


**Figure 5.3:** Relative energy vs mapping comparison for PilotNet for 260-280 images

### 5.1.4. Experiment 4: Energy optimization for a by varying the NPE's (video stream 0-20 images) fps = 60

Figure 5.4 shows the simulation results for 0-20 images with fps=60. In the given experiment, the NPEs as architectural parameters were varied along with the mapping (core per layer) in PilotNet and a comparison of the memory optimized, maximum energy, and minimum energy was made. The bars show the relative energy of the different mappings in a given experiment for different NPEs.

For both the minimum energy case and the memory-optimized case, the figure shows that there is an optimal choice of NPEs per core with NPE=16 that results in the lowest energy consumption. This is expected since our system streams data between NPEs in a given core. This means that the more NPEs we have, the more data is processed within the core without the need to copy data back and forth from the main memory. As a result, more NPEs make the system run more efficiently. In contrast, the energy consumption for maximum-energy mappings remains almost the same when the number of NPEs per core is changed, since the maximum energy is not affected by the optimizations used.

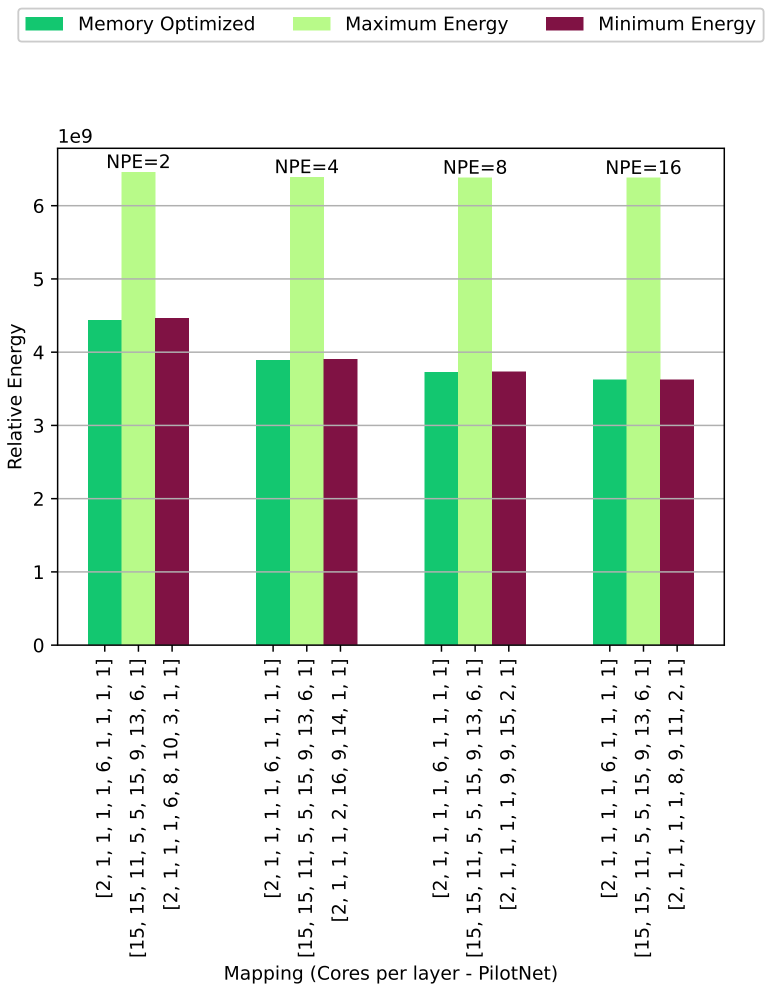


**Figure 5.4:** Relative energy vs mapping comparison for PilotNet for 0-20 images with fps=60

To further validate the observation from the above experiments that the lower number of NPEs perform better in general for an event based system as the system becomes asynchronous in the experiments, take the NPE as architectural parameters were taken as variables into the optimization problem.

### 5.1.5. Experiment 5: Comparison of relative energy for a segment less sparse (0-20) and more sparse (260-280) with NPE as optimization variable at fps = 0

Figure 5.5 shows a comparison of the results of a high-sparsity segment (in the two bars on the left for 0-20 images) and a low-sparsity segment (in the two bars on the right for 260-280 images). In the given experiment, the NPEs were taken as variables along with the (core per layer) mapping in PilotNet, and a comparison of the minimum energy and memory-optimized was made. The bars show the relative energy of the different mappings for fewer (left) and more (right) NPEs for different numbers of NPEs per core.

From the experiment, it can be inferred that an energy efficient mapping scheme with the NPEs for both the low sparsity segment and the high sparsity segment would result in a mapping with a lower number of NPEs. We also observe that the 1st core per layer is affected while the rest of the cores per layer remain the same. The high sparsity segment consumes less energy as compared to the low sparsity segment.

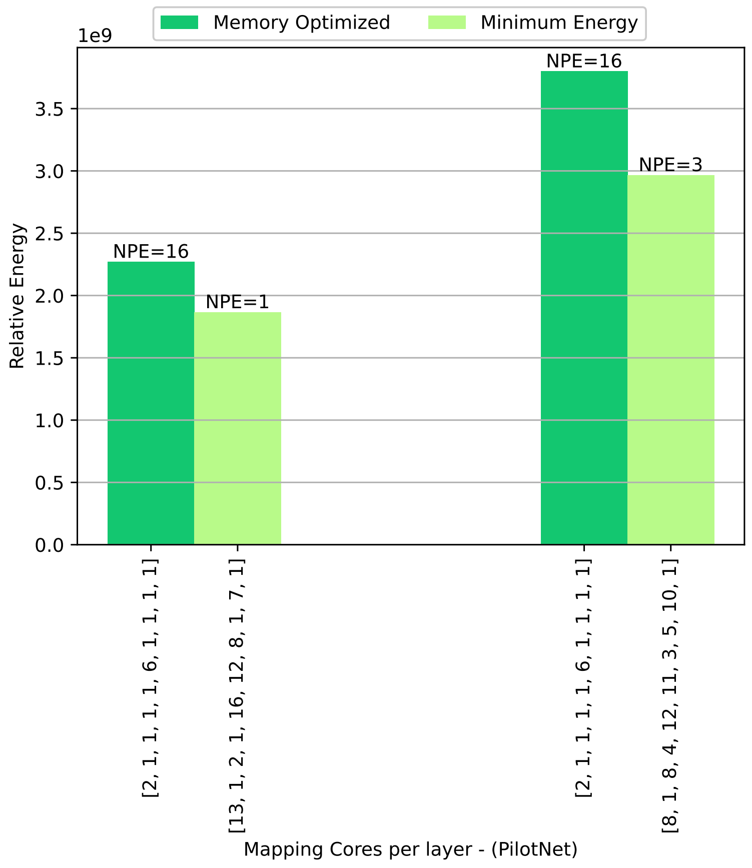


**Figure 5.5:** Relative energy vs mapping comparison for PilotNet for 0-20 images and 260-280 with NPE as the optimization variable for images with fps=0 (event based processing)

### 5.1.6. Experiment 6: Comparison of relative energy for a segment less sparse (0-30) and more sparse (260-290) with NPE as optimization variable at fps = 0

Figure 5.6 shows a comparison of the results of a high-sparsity segment (0-30 images) (2 bars on the left side) and a low-sparsity segment (260-290 images) (two bars on the right side).

In the experiment, the NPEs as architectural parameters were taken as variables along with the mapping (core per layer) in PilotNet, and a comparison of the memory-optimized and minimum-energy was done. The bars show the relative energy of the different mappings in a given experiment in a low- and high-sparsity segment. In general, one can observe that the lower number of NPEs for a given mapping scheme performs relatively better.

From the experiment, it can be inferred that an energy-efficient mapping scheme with the NPEs for both the low-sparsity segment and the high-sparsity segment would result in a mapping with a lower number of NPEs. We also observe that the 1st core per layer is affected while the rest of the cores per layer remain the same. The high-sparsity segment consumes less energy as compared to the low-sparsity segment.

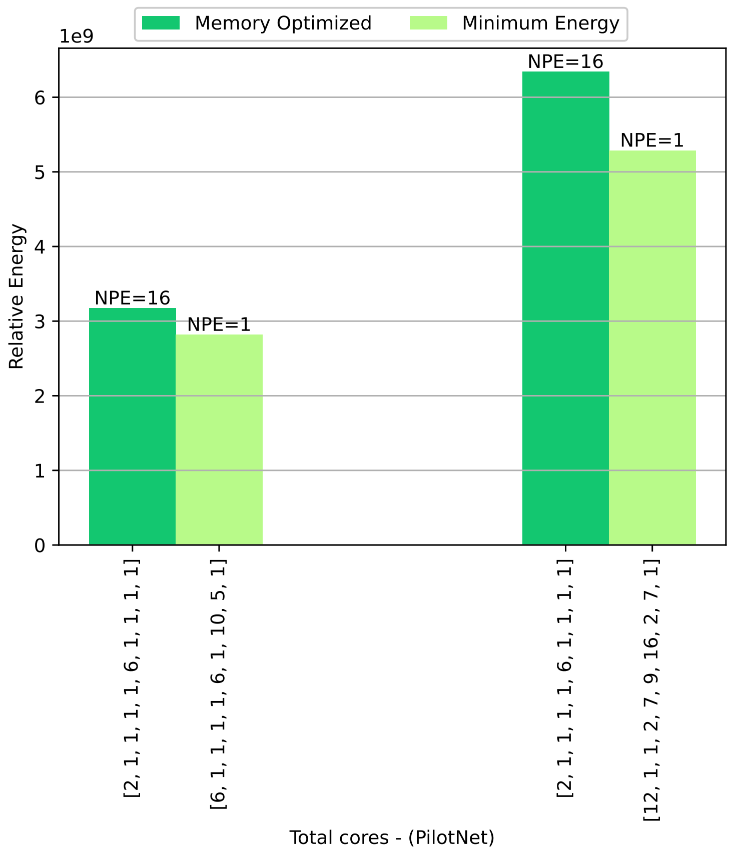


**Figure 5.6:** Relative energy vs mapping comparison for PilotNet for 0-30 images and 260-290 with NPE as the optimization variable for images with fps=0 (event-based processing)

### 5.1.7. Experiment 7: Relative energy-latency co-optimization with a variable number of NPE's (video stream 0-10 images) fps = 0 (event-based processing)

In this experiment, both latency and energy were optimized and the results were compared as shown in Figures 5.7 and 5.8. We also observe that in the case of energy-latency co-optimization, the number of solutions for mapping also varies, and a comparison of the memory-optimized, maximum energy, and minimum energy was done. The columns show the relative energy of the different mappings in a given experiment for different NPEs. In general, one can observe that the lower number of NPEs for a given mapping scheme performs relatively better for both latency and energy.

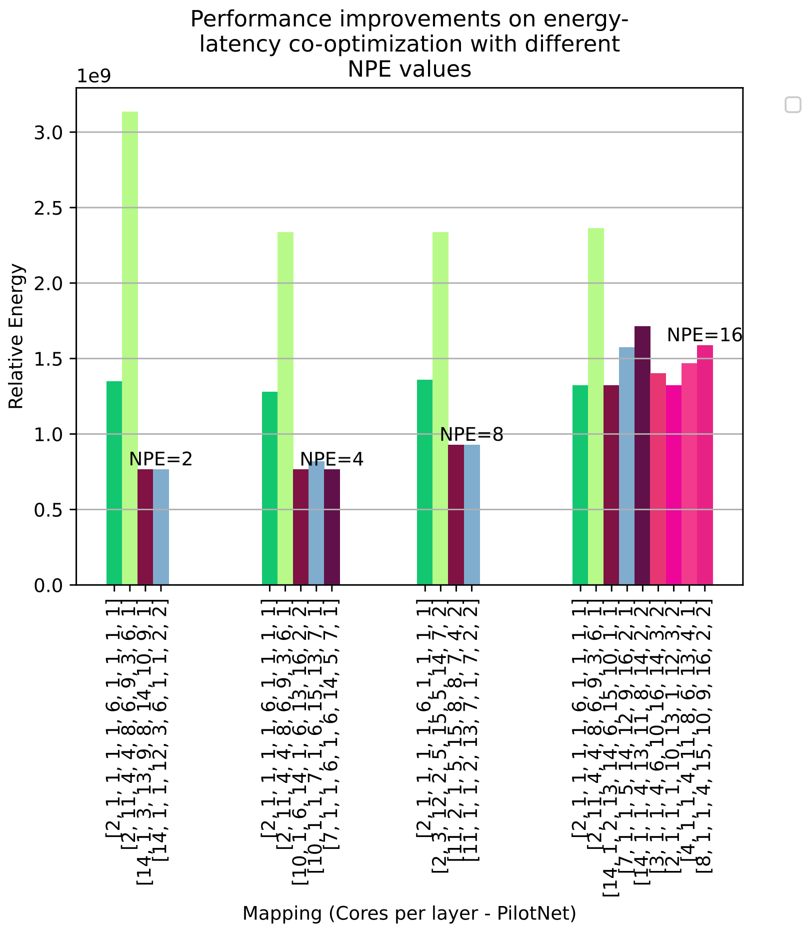


**Figure 5.7:** Baseline (dark green), maximum (light green) and energy-latency optimal (rest) mappings comparison for fps=0 video stream=(0-10 images) PilotNet

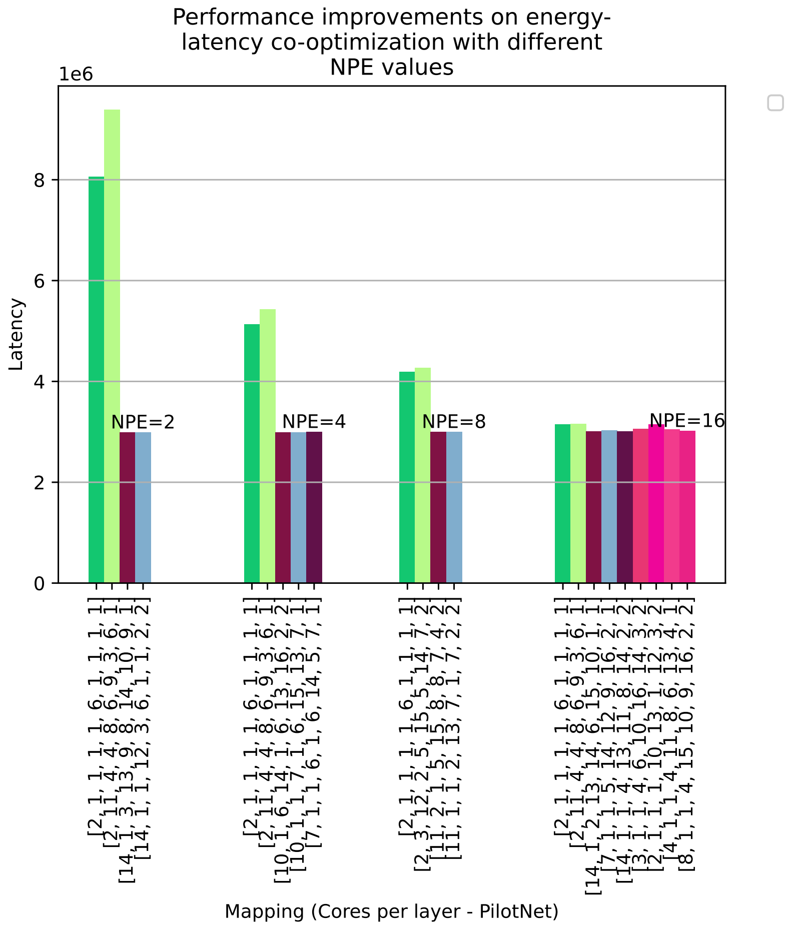


**Figure 5.8:** Baseline (dark green), maximum (light green) and energy-latency optimal (rest) mappings comparison for fps=0 video stream=(0-10 images) PilotNet

## 5.2. Validation and reasoning

### 5.2.1. Validation: SENSIM output comparison for (260-290) video stream and energy mapping comparison

From the above experiments conducted with PilotNet inference, it could be concluded that the NPEs and mapping depend on the rate. From Figure 5.9, one can observe the change in the output of SENSIM as the rate is varied. The output waveform is for the most optimal mapping, along with the NPEs, being recorded at the particular rate.

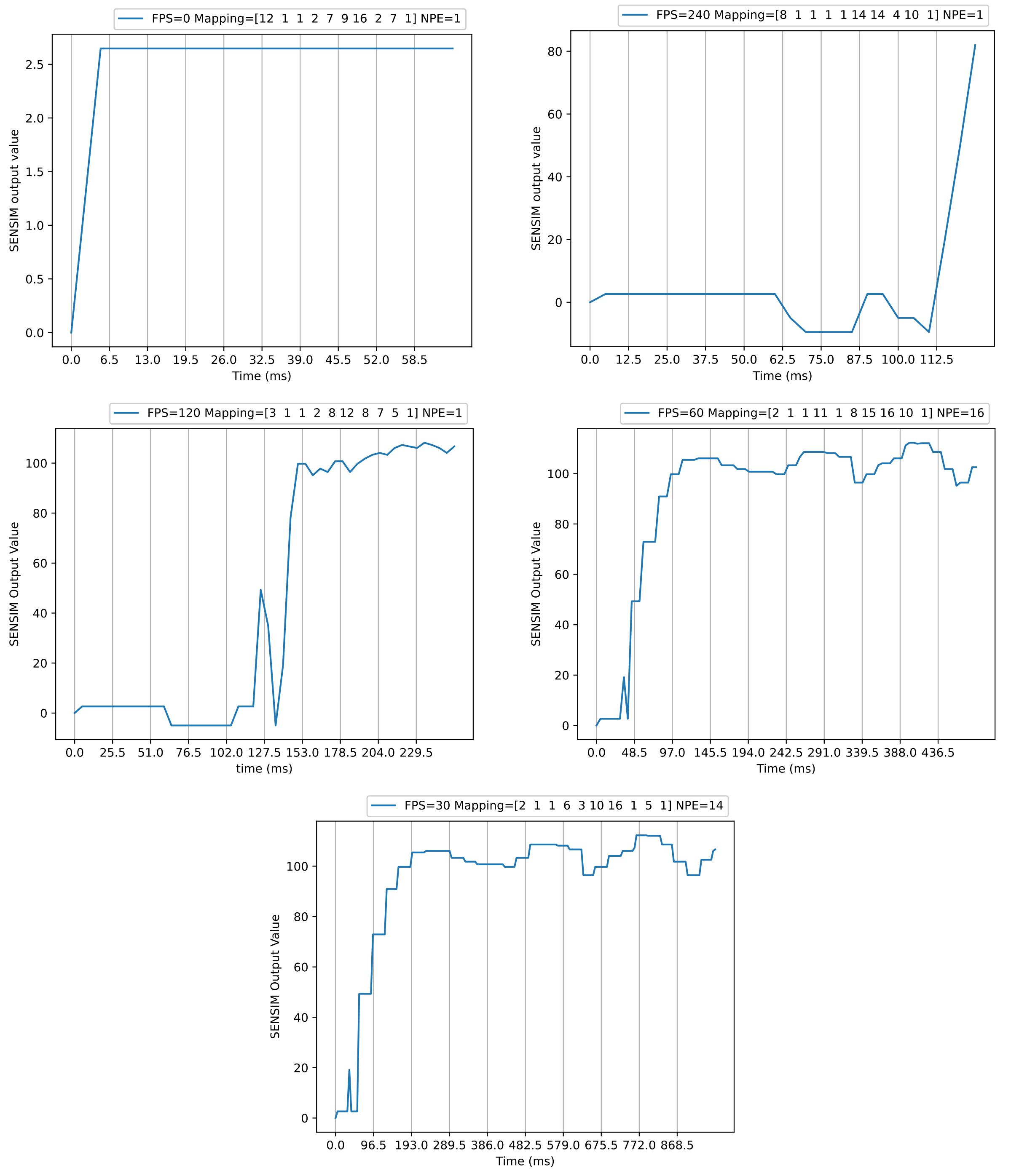


**Figure 5.9:** SENSIM output comparison (260-290) frames for different mapping schemes and the frames per rate

Figure 5.10 gives the energy reduction for different mappings with different NPE values at different rates. By adding the NPEs, into the optimization loop, one can also do an architectural design space exploration with SENMap.

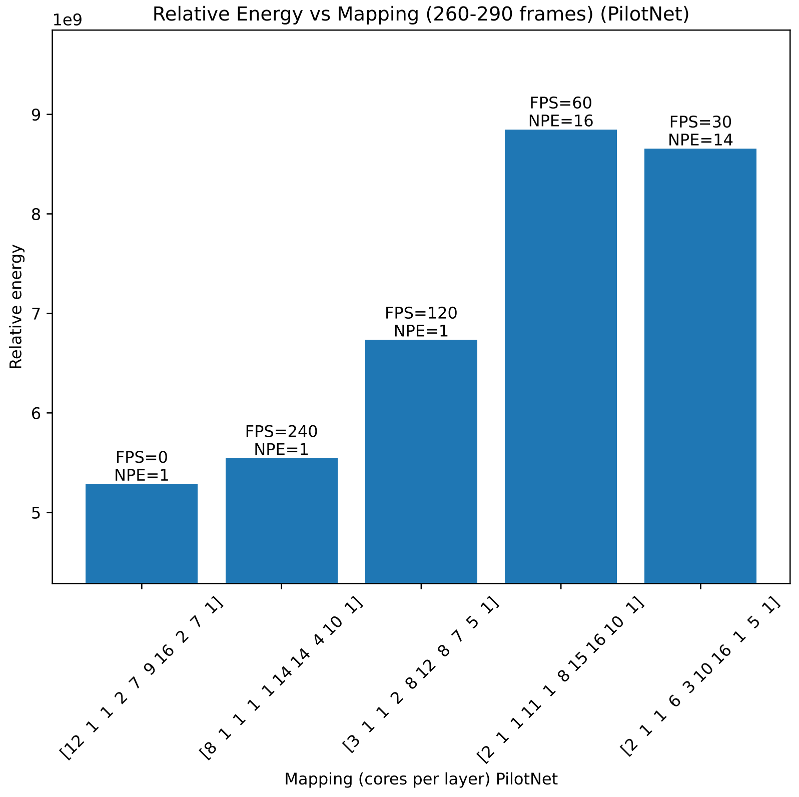


**Figure 5.10:** Relative energy vs mapping for (260-290) frames (PilotNet) at different rates

Figure 5.11 shows the comparison between the mapping and the PilotNet output signal for the first 500 images between fps=30,fps=60, and fps=120 and the optimal mapping with the NPE configuration. One can see from the figure that the signals at fps=30,fps=60, and fps=120 are pretty much alike. One can observe a clear difference in the shift in mapping on layer 1 and the total number of neural processing elements. For fps=120 the total system becomes rather an asynchronous system in comparison to fps=30, fps=60, and fps=120 correlation methods can be used.

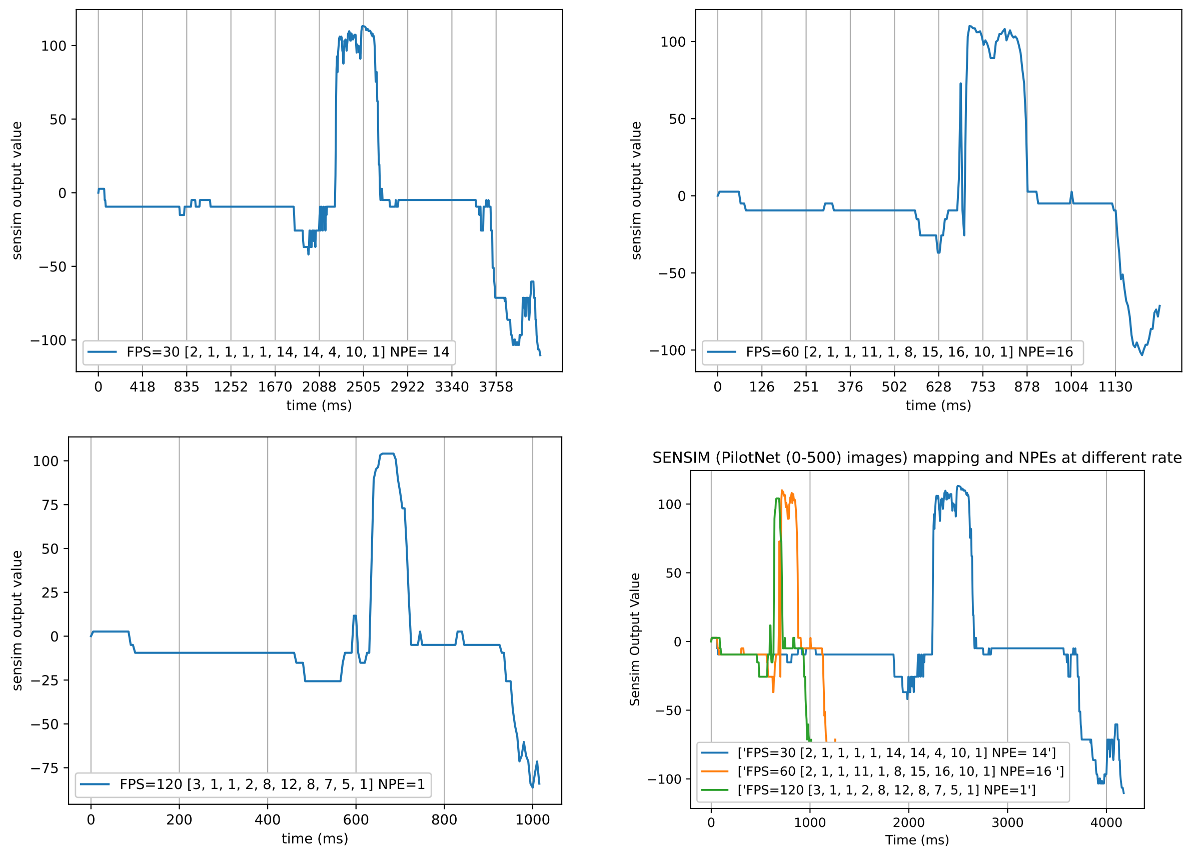


**Figure 5.11:** Comparing the output signal of PilotNet for fps=30,60,120 for a dataset of (0-500) images

### 5.2.2. Reasoning

In the case of SNNs, due to the delta activation layer, the amount of information from one layer to another is reduced to a significant amount depending on the neural network and the sparsity between the layers. When data are transferred at a significantly fast rate (fps = 0 or infinitely fast), the network/communication is the main bottleneck. In the case of a design with SIMD array, as suggested by SENeCA Section 2.2.1 and the synchronization of the current mode of operation time steps according to Table 2.12 the requirement of the total number of cores increases with the reduction of NPE. Since memory is a vague concept in SENSIM, an asynchronous mode of data transfer would be the most ideal for SNN applications with the optimal amount of NPEs that can be obtained from the SENMap.

In the case of a rate-based DNN architecture like trained with MNIST, if a particular rate (or time between the frames) is crossed, the signals start to interleave and in the case of a 10-digit class problem, there might be some interleaving of signals, which possibly could lead to incorrect output detection. The output accuracy can be measured via visual inspection only in the case of a DNN application. If there is more than one signal, the correlation techniques might change.

A single architecture design that serves the requirements of both rate-based DNN and SNN might not be the ideal choice. A DNN with a 4-layered and Pilot-Net-SNN are 2 applications that have been used in the experiment that needs to be done to understand the optimal mapping. An architecture that serves both DNN / SNN like spinnaker2 [27] would be ideal for energy latency and area-efficient edge AI applications. SENeCA, a RISC-V based Next generation Edge AI chip can easily be modified to serve both SNN/DNN applications and SENMap also serves as a tool for both SNN/DNN applications giving an optimal mapping.

# 6 Conclusion and future work

## 6.1. Conclusion

In this section, we discuss the main conclusions of the thesis. Here, we address all the research questions defined in the introduction of the thesis and discuss how we answered each one of them. In the following, we first reiterate each research question, followed by its answer. The research questions are preceded by a letter indicating whether the questions relates to software (**S**), algorithm (**A**) or hardware (**H**).

**S** How to make the mapper flexible for various large-scale applications, make it compatible with the existing software framework and make it scalable for future projects?
Answer: SENMap was designed in a way such that if the design of one core changes or the mathematical model of the core changes, the SENMap will not need any further modification. To make the mapper flexible, the mapper was integrated with a framework with several heuristics and meta heuristics algorithm with a flexibility to add algorithmic parameters, architectural parameters and much more.

**A** How to design a mapping algorithm and what preexisting algorithms could come into use?
Answer: Over the recent years, there are has been a significant development in heuristics, meta-heuristics and evolutionary algorithm. This thesis work combines the simulator with pre-existing frameworks and opens rooms for many optimization strategies.

**HA** How to partition or combine neurons in an SNN to map onto the hardware for better efficiency?
Answer: There were several single and multi objective optimization strategies explored and the SENMap framework opens room for multiple optimization strategies and algorithms such as genetic algorithms, particle swarm optimization algorithms, non-dominated sorting genetic algorithm etc.

**A** What optimization algorithms to use for mapping, and what would be the criteria to choose from the available set of algorithms?
Answer: Modular implementation of genetic algorithms and their multi objective variants were chosen for experimentation purpose with a fixed set of parameters.

**A** How to form the optimization problem and how to formulate the objective function?
Answer: The optimization problem was formed in such as way that the time to solution for the mapping is minimized. To minimized time to solution, the data-set (images in a video frame) inference was reduced, the problem was formed in such a way that the design space is minimized.

**H** What objectives will need to be focused on while formulating the objective function ?
Answer: The primary objectives for the optimization problem would be energy, latency, area depending on the chip that needs to be fabricated.

**HA** What could be the constraints and bounds for the problem one has to optimize on?

Answer: The constraints and bounds for the problem majorly depend on the physical constraints of the chip like the chip area.

**SA** How to speed up the process of finding the optimal mapping?
Answer: To speed up the optimization process, parallel computation and parallel meta heuristics could be leveraged depending upon the compute resources available.

**SH** How to add more parameters to the existing mapping problem and find ways to design a better architecture (hardware/software co-optimization)?
Answer: SENMap was designed in such a way that more architectural parameters like the queue size, the memory per core could be changed for every experiment conducted which can make hardware software co-design possible.

**A** What factors depend on the mapping?
Answer: On experimentation, it was evident that along with the mapping, the NPEs and the rate at which the system is processing data will most affect the mapping for any given neural network.

While all of the research questions were answered in the thesis, there remains scope for more improvement as discussed below in the future work Section 6.2.

## 6.2. Future work

**Combine layers in a neural network**
The SENSIM execution model does not support the combination of different layers. Adding the feature to SENSIM would give more access to possible mapping schemes. While combining of layers and clustering neurons of adjacent layer was attempted in the thesis work the end signal being distorted beyond 60 fps. An attempt to keep the end signal in the loop with an error factor of the correlation to be minimized would make the clustering of layer possible.

**Optimization for 3D stacking of cores**
The curse of dimensionality [77] refers to various phenomena that arise when analyzing and organizing data in high-dimensional space, as coined by Richard E. Bellman. By stacking up cores or introducing 2.5D or 3D memory [20], the time to solution for mapping the neurons on to the chip increases exponentially. The focus of the future work would be on designing optimization strategies for getting a better time to solution for mapping large scale spiking neural networks on these architectures.

**Run complete closed loop synthesis**
Currently in the experimentation, neural processing elements were added to the optimization loop. There could be other parameters like the communication parameters which could be added to the optimization strategies, along with placement routing/hops that could also be optimized.

**Multistep optimization**
There are heuristics such as [42] and [45] that primarily build graph-based heuristics that could possibly reduce time to solution by performing a step-wise optimization.

**Area modeling and optimization**
In the current SENSIM version, the area is not modeled from the synthesized measurements from [2] and if exact measurements are made, a closed-loop synthesis of the chip can be made possible with multi-objective optimization algorithms [71], [73] etc. Instead of a total number of cores as an objective, the die area of the chip can be extracted.

**Multiple application execution and optimization**
Currently, mappers map only one SNN/DNN application such as PilotNet [34] [33] but the mapper can be easily scaled to large-scale cascaded SNN and DNN spiking neural networks. An example that can be cascaded could be YOLO [78] SNN/DNN with the RESNET [79] SNN/DNN application.

**Run Time Mapping optimization**
[44] proposes a method in which the mapping of neurons in runtime optimizes on a chip such that the end mapping after certain epochs gives the most energy efficient mapping.

# References

[1] A. Yousefzadeh *et al.*, “SENeCA: Scalable Energy-efficient Neuromorphic Computer Architecture,” Jun. 2022. DOI: `10.1109/AICAS54282.2022.9870025`.

[2] A. Yousefzadeh *et al.*, “SENeCA: Scalable Energy-efficient Neuromorphic Computer Architecture,” en, p. 4,

[3] W. Maass, “Networks of spiking neurons: The third generation of neural network models,” en, *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997, ISSN: 0893-6080. DOI: `10.1016/S0893-6080(97)00011-7`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0893608097000117` (visited on 12/23/2022).

[4] M. Davies *et al.*, “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning,” *IEEE Micro*, vol. PP, pp. 1–1, Jan. 2018. DOI: `10.1109/MM.2018.112130359`.

[5] S. Furber, “Large-scale neuromorphic computing systems,” en, *Journal of Neural Engineering*, vol. 13, no. 5, p. 051 001, Oct. 2016, ISSN: 1741-2560, 1741-2552. DOI: `10.1088/1741-2560/13/5/051001`. [Online]. Available: `https://iopscience.iop.org/article/10.1088/1741-2560/13/5/051001` (visited on 07/28/2022).

[6] C. M. Vineyard, R. Dellana, J. B. Aimone, F. Rothganger, and W. M. Severa, “Low-Power Deep Learning Inference using the SpiNNaker Neuromorphic Platform,” en, in *Proceedings of the 7th Annual Neuro-inspired Computational Elements Workshop on - NICE ’19*, Albany, NY, USA: ACM Press, 2019, pp. 1–7, ISBN: 978-1-4503-6123-1. DOI: `10.1145/3320288.3320300`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=3320288.3320300` (visited on 07/29/2022).

[7] C. Teeter *et al.*, “Generalized leaky integrate-and-fire models classify multiple neuron types,” en, *Nature Communications*, vol. 9, no. 1, p. 709, Feb. 2018, Number: 1 Publisher: Nature Publishing Group, ISSN: 2041-1723. DOI: `10.1038/s41467-017-02717-4`. [Online]. Available: `https://www.nature.com/articles/s41467-017-02717-4` (visited on 11/18/2022).

[8] R. Brette and W. Gerstner, “Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity,” *Journal of Neurophysiology*, vol. 94, no. 5, pp. 3637–3642, Nov. 2005, Publisher: American Physiological Society, ISSN: 0022-3077. DOI: `10.1152/jn.00686.2005`. [Online]. Available: `https://journals.physiology.org/doi/full/10.1152/jn.00686.2005` (visited on 11/18/2022).

[9] R. Jolivet, A. Rauch, H.-r. Lüscher, and W. Gerstner, “Integrate-and-Fire models with adaptation are good enough,” in *Advances in Neural Information Processing Systems*, vol. 18, MIT Press, 2005. [Online]. Available: `https://proceedings.neurips.cc/paper/2005/hash/42a6845a557bef704ad8ac9cb4461d43-Abstract.html` (visited on 11/13/2022).

[10] P. O’Connor and M. Welling, “SIGMA-DELTA QUANTIZED NETWORKS,” en, p. 15, 2017.

[11] P. O’Connor and M. Welling, *Deep Spiking Networks*, arXiv:1602.08323 [cs], Nov. 2016. DOI: `10.48550/arXiv.1602.08323`. [Online]. Available: `http://arxiv.org/abs/1602.08323` (visited on 09/11/2022).

[12] K. Koch *et al.*, “How Much the Eye Tells the Brain,” *Current biology : CB*, vol. 16, no. 14, pp. 1428–1434, Jul. 2006, ISSN: 0960-9822. DOI: `10.1016/j.cub.2006.05.056`. [Online]. Available: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1564115/` (visited on 09/11/2022).

[13] O. Vermesan and M. D. Nava, “Intelligent Edge-Embedded Technologies for Digitising Industry,” en, in 2022, pp. 1–340. DOI: `10.13052/rp-9788770226103`. [Online]. Available: `https://www.riverpublishers.com/research_details.php?book_id=1023` (visited on 12/02/2022).

[14] D. Kalamkar *et al.*, *A Study of BFLOAT16 for Deep Learning Training*, arXiv:1905.12322 [cs, stat], Jun. 2019. DOI: `10.48550/arXiv.1905.12322`. [Online]. Available: `http://arxiv.org/abs/1905.12322` (visited on 10/30/2022).

[15] B. Jacob *et al.*, *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*, arXiv:1712.05877 [cs, stat], Dec. 2017. DOI: `10.48550/arXiv.1712.05877`. [Online]. Available: `http://arxiv.org/abs/1712.05877` (visited on 10/30/2022).

[16] U. Köster *et al.*, *Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks*, arXiv:1711.02213 [cs, stat], Dec. 2017. DOI: `10.48550/arXiv.1711.02213`. [Online]. Available: `http: //arxiv.org/abs/1711.02213` (visited on 10/30/2022).

[17] A. Yousefzadeh *et al.*, *Multiplexing AER Asynchronous Channels over LVDS Links with Flow-Control and Clock-Correction for Scalable Neuromorphic Systems*. May 2017. DOI: `10.1109/ISCAS.2017.8050802`.

[18] A. Waterman, K. Asanovic, and C. Division, en, p. 145,

[19] *Ibex: An embedded 32 bit RISC-V CPU core — Ibex Documentation 0.1.dev50+g9a65bc1.d20221129 documentation*. [Online]. Available: `https://ibex-core.readthedocs.io/en/latest/` (visited on 12/02/2022).

[20] F. Sheikh, R. Nagisetty, T. Karnik, and D. Kehlet, "2.5D and 3D Heterogeneous Integration: Emerging applications," *IEEE Solid-State Circuits Magazine*, vol. 13, no. 4, pp. 77–87, 2021, Conference Name: IEEE Solid-State Circuits Magazine, ISSN: 1943-0590. DOI: `10.1109/MSSC.2021.3111386`.

[21] L. Bamberg, J. M. Joseph, A. García-Ortiz, and T. Pionteck, *3D Interconnect Architectures for Heterogeneous Technologies: Modeling and Optimization*, en. Cham: Springer International Publishing, 2022, ISBN: 978-3-030-98228-7 978-3-030-98229-4. DOI: `10.1007/978-3-030-98229-4`. [Online]. Available: `https://link.springer.com/10.1007/978-3-030-98229-4` (visited on 10/30/2022).

[22] E. Painkras *et al.*, "SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation," *Solid-State Circuits, IEEE Journal of*, vol. 48, pp. 1943–1953, Aug. 2013. DOI: `10.1109/JSSC.2013.2259038`.

[23] C. Mayr, S. Hoeppner, and S. Furber, *SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning*, arXiv:1911.02385 [cs], Nov. 2019. DOI: `10.48550/arXiv.1911.02385`. [Online]. Available: `http://arxiv.org/abs/1911.02385` (visited on 12/12/2022).

[24] F. Akopyan *et al.*, "TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip," en, vol. 34, no. 10, p. 21, 2015.

[25] J. Stuijt, M. Sifalakis, A. Yousefzadeh, and F. Corradi, "MBrain: An Event-Driven and Fully Synthesizable Architecture for Spiking Neural Networks," *Frontiers in Neuroscience*, vol. 15, 2021, ISSN: 1662-453X.[Online]. Available: `https://www.frontiersin.org/articles/10.3389/fnins.2021.664208` (visited on 10/30/2022).

[26] M. Davies, "Intel Labs' new Loihi 2 research chip outperforms its predecessor by up to 10x and comes with an open-source, community-driven neuromorphic computing framework," en, p. 7,

[27] S. Höppner *et al.*, *The SpiNNaker 2 Processing Element Architecture for Hybrid Digital Neuromorphic Computing*, arXiv:2103.08392 [cs], Aug. 2022. DOI: `10.48550/arXiv.2103.08392`. [Online]. Available: `http://arxiv.org/abs/2103.08392` (visited on 12/11/2022).

[28] PyQT, "PyQt Reference Guide," 2012. [Online]. Available: `http://www.riverbankcomputing.com/static/Docs/PyQt4/html/index.html`.

[29] F. Lundh, "An introduction to tkinter," *URL: www. pythonware. com/library/tkinter/introduction/index. htm*, 1999.

[30] H. Jun *et al.*, "HBM (High Bandwidth Memory) DRAM Technology and Architecture," in *2017 IEEE International Memory Workshop (IMW)*, May 2017, pp. 1–4. DOI: `10.1109/IMW.2017.7939084`.

[31] E. M. Izhikevich, "Resonate-and-fire neurons," en, *Neural Networks*, vol. 14, no. 6, pp. 883–894, Jul. 2001, ISSN: 0893-6080. DOI: `10.1016/S0893-6080(01)00078-8`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0893608001000788` (visited on 11/13/2022).

[32] A. Burkitt, "A Review of the Integrate-and-fire Neuron Model: I. Homogeneous Synaptic Input," *Biological cybernetics*, vol. 95, pp. 1–19, Aug. 2006. DOI: `10.1007/s00422-006-0068-6`.

[33] M. Bojarski *et al.*, *Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car*, en, Number: arXiv:1704.07911 arXiv:1704.07911 [cs], Apr. 2017. [Online]. Available: `http://arxiv.org/abs/1704.07911` (visited on 07/28/2022).

[34] M. Bojarski *et al.*, *End to End Learning for Self-Driving Cars*, en, Number: arXiv:1604.07316 arXiv:1604.07316 [cs], Apr. 2016. [Online]. Available: `http://arxiv.org/abs/1604.07316` (visited on 07/28/2022).

[35] *MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges*. [Online]. Available: `http://yann.lecun.com/exdb/mnist/` (visited on 12/08/2022).

[36] C.-K. Lin, A. Wild, G. N. Chinya, T.-H. Lin, M. Davies, and H. Wang, "Mapping spiking neural networks onto a manycore neuromorphic architecture," en, in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia PA USA: ACM, Jun. 2018, pp. 78–89, ISBN: 978-1-4503-5698-5. DOI: `10.1145/3192366.3192371`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3192366.3192371` (visited on 11/17/2022).

[37] M. Beyeler, K. D. Carlson, T.-S. Chou, N. Dutt, and J. L. Krichmar, "CARLsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks," in *2015 International Joint Conference on Neural Networks (IJCNN)*, ISSN: 2161-4407, Jul. 2015, pp. 1–8. DOI: `10.1109/IJCNN.2015.7280424`.

[38] A. Balaji *et al.*, "Mapping Spiking Neural Networks to Neuromorphic Hardware," en, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 76–86, Jan. 2020, ISSN: 1063-8210, 1557-9999. DOI: `10.1109/TVLSI.2019.2951493`. [Online]. Available: `https://ieeexplore.ieee.org/document/8913677/` (visited on 07/28/2022).

[39] A. Balaji *et al.*, *PyCARL: A PyNN Interface for Hardware-Software Co-Simulation of Spiking Neural Network*, arXiv:2003.09696 [cs], May 2020. [Online]. Available: `http://arxiv.org/abs/2003.09696` (visited on 09/18/2022).

[40] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, "A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)," eng, *IEEE transactions on biomedical circuits and systems*, vol. 12, no. 1, pp. 106–122, Feb. 2018, ISSN: 1940-9990. DOI: `10.1109/TBCAS.2017.2759700`.

[41] S. Li *et al.*, "SNEAP: A Fast and Efficient Toolchain for Mapping Large-Scale Spiking Neural Network onto NoC-based Neuromorphic Platform," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '20, New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 9–14, ISBN: 978-1-4503-7944-1. DOI: `10.1145/3386263.3406900`. [Online]. Available: `https://doi.org/10.1145/3386263.3406900` (visited on 11/26/2022).

[42] G. Karypis and V. Kumar, "Multilevelk-way Partitioning Scheme for Irregular Graphs," en, *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, Jan. 1998, ISSN: 0743-7315. DOI: `10.1006/jpdc.1997.1404`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0743731597914040` (visited on 11/26/2022).

[43] F. Kelber *et al.*, "Mapping Deep Neural Networks on SpiNNaker2," in *Proceedings of the Neuro-inspired Computational Elements Workshop*, ser. NICE '20, New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 1–3, ISBN: 9781450377188. DOI: `10.1145/3381755.3381778`. [Online]. Available: `https://doi.org/10.1145/3381755.3381778` (visited on 12/10/2022).

[44] A. Balaji, T. Marty, A. Das, and F. Catthoor, *Run-time Mapping of Spiking Neural Networks to Neuromorphic Hardware*, Number: arXiv:2006.06777 arXiv:2006.06777 [cs], Jun. 2020. DOI: `10.48550/arXiv.2006.06777`. [Online]. Available: `http://arxiv.org/abs/2006.06777` (visited on 07/28/2022).

[45] M. Predari and A. Esnard, "A k-Way Greedy Graph Partitioning with Initial Fixed Vertices for Parallel Applications," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, ISSN: 2377-5750, Feb. 2016, pp. 280–287. DOI: `10.1109/PDP.2016.109`.

[46] L. Hernando, A. Mendiburu, and J. A. Lozano, "Hill-Climbing Algorithm: Let's Go for a Walk Before Finding the Optimum," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, Jul. 2018, pp. 1–7. DOI: `10.1109/CEC.2018.8477836`.

[47] *Simulated annealing*, en, Page Version ID: 1125897531, Dec. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Simulated_annealing&oldid=1125897531` (visited on 12/19/2022).

[48] *Evolutionary algorithm*, en, Page Version ID: 1127919436, Dec. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Evolutionary_algorithm&oldid=1127919436` (visited on 12/19/2022).

[49] J. F. Gonçalves and M. G. C. Resende, “Biased random-key genetic algorithms for combinatorial optimization,” en, *Journal of Heuristics*, vol. 17, no. 5, pp. 487–525, Oct. 2011, ISSN: 1572-9397. DOI: 10.1007/s10732-010-9143-1. [Online]. Available: https://doi.org/10.1007/s10732-010-9143-1 (visited on 11/18/2022).

[50] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002, Conference Name: IEEE Transactions on Evolutionary Computation, ISSN: 1941-0026. DOI: 10.1109/4235.996017.

[51] *Swarm intelligence*, en, Page Version ID: 1140889260, Feb. 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Swarm_intelligence&oldid=1140889260 (visited on 02/24/2023).

[52] M. Dorigo, M. Birattari, and T. Stutzle, “Ant colony optimization,” *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, Nov. 2006, ISSN: 1556-6048. DOI: 10.1109/MCI.2006.329691.

[53] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of ICNN’95 - International Conference on Neural Networks*, vol. 4, Nov. 1995, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.

[54] F. Glover, M. Laguna, and R. Marti, *Tabu Search*. Jul. 2008, vol. 16, Journal Abbreviation: Tabu Search Publication Title: Tabu Search. DOI: 10.1007/978-1-4615-6089-0.

[55] B. W. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, Feb. 1970, ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[56] J. Blank and K. Deb, “Pymoo: Multi-Objective Optimization in Python,” *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2990567.

[57] *How to cite OR-Tools*, en. [Online]. Available: https://developers.google.com/optimization/support/cite (visited on 07/28/2022).

[58] P. Virtanen *et al.*, “SciPy 1.0: Fundamental algorithms for scientific computing in Python,” en, *Nature Methods*, vol. 17, no. 3, pp. 261–272, Mar. 2020, ISSN: 1548-7091, 1548-7105. DOI: 10.1038/s41592-019-0686-2. [Online]. Available: http://www.nature.com/articles/s41592-019-0686-2 (visited on 07/28/2022).

[59] A. Benítez-Hidalgo, A. J. Nebro, J. García-Nieto, I. Oregi, and J. Del Ser, “jMetalPy: A Python framework for multi-objective optimization with metaheuristics,” en, *Swarm and Evolutionary Computation*, vol. 51, p. 100 598, Dec. 2019, ISSN: 2210-6502. DOI: 10.1016/j.swevo.2019.100598. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210650219301397 (visited on 07/28/2022).

[60] F. Biscani and D. Izzo, “A parallel global multiobjective framework for optimization: Pagmo,” en, *Journal of Open Source Software*, vol. 5, no. 53, p. 2338, Sep. 2020, ISSN: 2475-9066. DOI: 10.21105/joss.02338. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.02338 (visited on 07/28/2022).

[61] J. Mejía, *Metaheuristics*, original-date: 2017-10-29T05:05:37Z, Jan. 2023. [Online]. Available: https://github.com/jmejia8/Metaheuristics.jl (visited on 01/31/2023).

[62] C. Gagné and M. Parizeau, “Open BEAGLE: A new C++ Evolutionary Computation framework,” in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO’02, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Jul. 2002, p. 888, ISBN: 978-1-55860-878-8. (visited on 11/19/2022).

[63] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, “Opt4J: A modular framework for meta-heuristic optimization,” en, in *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO ’11*, Dublin, Ireland: ACM Press, 2011, p. 1723, ISBN: 978-1-4503-0557-0. DOI: 10.1145/2001576.2001808. [Online]. Available: http://portal.acm.org/citation.cfm?doid=2001576.2001808 (visited on 11/19/2022).

[64] *Differential Evolution*, en. [Online]. Available: https://link.springer.com/book/10.1007/3-540-31306-0 (visited on 07/28/2022).

[65] J. A. Nelder and R. Mead, “A Simplex Method for Function Minimization,” *The Computer Journal*, vol. 7, no. 4, pp. 308–313, Jan. 1965, ISSN: 0010-4620. DOI: `10.1093/comjnl/7.4.308`. [Online]. Available: `https://doi.org/10.1093/comjnl/7.4.308` (visited on 07/28/2022).

[66] R. Hooke and T. A. Jeeves, ““ Direct Search” Solution of Numerical and Statistical Problems,” *Journal of the ACM*, vol. 8, no. 2, pp. 212–229, Apr. 1961, ISSN: 0004-5411. DOI: `10.1145/321062.321069`. [Online]. Available: `https://doi.org/10.1145/321062.321069` (visited on 11/18/2022).

[67] N. Hansen, “A global surrogate assisted CMA-ES,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO ’19, New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 664–672, ISBN: 978-1-4503-6111-8. DOI: `10.1145/3321707.3321842`. [Online]. Available: `https://doi.org/10.1145/3321707.3321842` (visited on 10/30/2022).

[68] T. Runarsson and X. Yao, “Stochastic ranking for constrained evolutionary optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 4, no. 3, pp. 284–294, Sep. 2000, Conference Name: IEEE Transactions on Evolutionary Computation, ISSN: 1941-0026. DOI: `10.1109/4235.873238`.

[69] T. Runarsson and X. Yao, “Search biases in constrained evolutionary optimization,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 35, no. 2, pp. 233–243, May 2005, Conference Name: IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), ISSN: 1558-2442. DOI: `10.1109/TSMCC.2004.841906`.

[70] K. Deb and J. Sundar, “Reference Point Based Multi-Objective Optimization Using Evolutionary Algorithms,” en, p. 11,

[71] Y. Vesikar, K. Deb, and J. Blank, “Reference Point Based NSGA-III for Preferred Solutions,” in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, Nov. 2018, pp. 1587–1594. DOI: `10.1109/SSCI.2018.8628819`.

[72] J. Blank, K. Deb, Y. Dhebar, S. Bandaru, and H. Seada, “Generating Well-Spaced Points on a Unit Simplex for Evolutionary Many-Objective Optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 1, pp. 48–60, Feb. 2021, Conference Name: IEEE Transactions on Evolutionary Computation, ISSN: 1941-0026. DOI: `10.1109/TEVC.2020.2992387`.

[73] Q. Zhang, S. Member, and H. Li, *MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition*.

[74] A. Panichella, “An adaptive evolutionary algorithm based on non-euclidean geometry for many-objective optimization,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO ’19, New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 595–603, ISBN: 978-1-4503-6111-8. DOI: `10.1145/3321707.3321839`. [Online]. Available: `https://doi.org/10.1145/3321707.3321839` (visited on 07/28/2022).

[75] K. Li, R. Chen, G. Fu, and X. Yao, “Two-Archive Evolutionary Algorithm for Constrained Multiobjective Optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 2, pp. 303–315, Apr. 2019, Conference Name: IEEE Transactions on Evolutionary Computation, ISSN: 1941-0026. DOI: `10.1109/TEVC.2018.2855411`.

[76] Y. Ho and D. Pepyne, “Simple Explanation of the No-Free-Lunch Theorem and Its Implications,” en, *Journal of Optimization Theory and Applications*, vol. 115, no. 3, pp. 549–570, Dec. 2002, ISSN: 1573-2878. DOI: `10.1023/A:1021251113462`. [Online]. Available: `https://doi.org/10.1023/A:1021251113462` (visited on 12/10/2022).

[77] *Curse of dimensionality*, en, Page Version ID: 1125139893, Dec. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Curse_of_dimensionality&oldid=1125139893` (visited on 12/18/2022).

[78] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, arXiv:1506.02640 [cs], May 2016. [Online]. Available: `http://arxiv.org/abs/1506.02640` (visited on 11/17/2022).

[79] K. He, X. Zhang, S. Ren, and J. Sun, *Deep Residual Learning for Image Recognition*, arXiv:1512.03385 [cs], Dec. 2015. [Online]. Available: `http://arxiv.org/abs/1512.03385` (visited on 11/17/2022).