EXPLORING OPPORTUNITIES AND CHALLENGES IN ENABLING NEURO-EVOLUTIONARY ALGORITHMS IN HARDWARE

A Dissertation
Presented to
The Academic Faculty

By

Parth Mannan

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2018

EXPLORING OPPORTUNITIES AND CHALLENGES IN ENABLING NEURO-EVOLUTIONARY ALGORITHMS IN HARDWARE

Approved by:

Dr. Tushar Krishna, Advisor
School of Electrical and Compute Engineering
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Saibal Mukhopadhyay
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: December 7, 2018

Imagination is more important than knowledge

Albert Einstein

Dedicated to my Parents and Grandmother

ACKNOWLEDGEMENTS

I consider myself extremely blessed to have had the opportunity to work with one of the brightest minds in this new era of Computer Architecture, Dr. Tushar Krishna. This work would have been impossible without his continuous guidance and mentorship at every step in my research and beyond. I hope I have been able to absorb some of his quick-thinking and his skill at presenting research. His constant support in academic, professional and personal aspects of my life have made this Masters possible.

This page would be incomplete without thanking Dr. Moin Quereshi whose ECE 6100 ignited my apetite for Computer Architecture. I also would like to sincerely thank Dr. Hyesoon Kim and Dr. Saibal Mukhopadhyay for agreeing to be on my thesis committee and their constant support and guidance. I would also like to sincerely thank Anand Samajdar. His constant enthusiasm and his belief in our work has always served as my biggest motivation.

Through the course of this Masters, I have been fortunate enough to have forged exceptional bonds with a few exceptional people. Each person I have interacted with has left an un-erasable mark on me. Thanks to Anita Ramasamy for her unfailing support and her unshakable belief in me. Thanks to Sanjana Sampath and Nikhil Gupta for their amazing companionship. Thanks to Keshav Ramani for his everlasting excitement about everything. Thanks to Adithya NC for sharing my enthusiam for Computer Architecture every single day. I would also like to thank Shruti Shinde, Girish Narayanan and Keval Kamdar for making my time here a wonderful one.

My sincerest gratitude goes to my parents and my grandmother who have supported me unconditionally in every endeavor I have chosen to embark upon and the success of this one would not be a possibility without their love, support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Recent advancements in the machine learning algorithms, especially the development of Deep Neural Networks (DNNs) have transformed the landscape of Artificial Intelligence (AI). With every passing day, deep learning based methods are applied to solve new problems with exceptional results. However true impact of AI could only be fully realized if it interacts with the real world and solves everyday problems.

The everyday problem however, is new everyday and subject to increasingly changing requirements. The Deep Learning (DL) landscape today is incapable of solving these dynamic problems as the performance of DL today is heavily tied to the topology which is often task specific and hand-tuned by experts. Not only is rigidity of the solution the problem but also the high memory and compute requirements of DNNs to perform training on terabytes of data acts a huge barrier in bringing true intelligence to the edge which is the true portal to the 'real world'.

NeuroEvolution (NE) are a class of algorithms that can circumvent this problem by 'learning on the fly'. These algorithms continuously interact with the environment and update their models based on how fruitful their last interaction proved. This way the solution is not tied to a topology and these algorithms do not need to perform memory and compute intensive backpropagation operations (BP) making them ideal for solving dynamic problems in a robust manner on the edge.

However, the barrier to deployment of NE today is the lack of its widespread adoption and understanding of its compute behavior. This thesis attempts to lift that barrier by characterizing the compute and communication behavior a NE algorithm NEAT (NeuroEvolution of Augmenting Topologies) and is an attempt to propel further research in this direction. This Thesis also attempts to bring intelligence to the edge using a distributed system solution.

This thesis demonstrates **CLAN**, **C**ollaborative **L**earning using **A**synchronous **N**euro-

evolution. It proposes techniques for enabling adaptive intelligence on the edge using NE algorithms collaboratively on Raspberry Pis and demonstrate that CLAN can match performance of higher end computing devices with better energy efficiency at scale. Further, this thesis also propose algorithmic modifications to improve the scalability.

The study performed in this work aims to drive key insights to both computer architects and distributed system engineers to enable effort in deploying NE on the modern day compute platform.

CHAPTER 1
INTRODUCTION

The past decade has seen tremendous efforts in the space of Machine Learning (ML) and Artificial Intelligence (AI) by both the academic and industry research communities. This growth can be attributed to the capability of Neural Networks to solve a wide variety of complex problems. Advancements in deep learning (DL) is the defining technological milestone of the present times. Tasks like, computer vision, speech comprehension etc. which were deemed impossible by computers at the turn of the decade, are now being routinely performed at superhuman accuracy. No wonder it has captured the imagination of many. While some see these advancements as the dawn of the new industrial revolution, some interpret it as a foreboding of an AI uprising. Superlatives aside, new techniques have industry altering consequences, and deep learning has already proved its mettle at it.

Supervised Learning, the engine powering this revolution, however has its limitations. DL solutions are only as good as its topology and the data-set it has been trained on. Prerequisites are therefore, gargantuan amounts of data, meticulous labelling and careful construction of the topology by experts. After the ingredients are in place, the model then needs to be run on a high performance computing system for weeks at a time for training. These dependencies make Deep Learning applicable only to a limited to set of problems. It is therefore quite natural that AI happened only on the cloud. The edge devices, for most cases, serve only as an interface collecting queries and displaying the results processed in the cloud. Although recent advancements in specialized accelerators have enabled inference on edge, training still remains confined to cloud.

The true potential of machine learning can only be realized when intelligence moves to edge. However, performing intelligence on the edge has a unique set of challenges. First the environment of deployment can have a huge degree of variation, second, over

time the requirements can drift. It can be immediately recognized that present offload model to the cloud is far from ideal. The first problem dictates that the training data-set needs to be diverse enough to handle the variations of all different types of deployment environments, which is not realistic. Second challenge requires the model to be updated and trained on a regular basis to ensure adaptability. As more and more edge devices use artificial intelligence, the scale of the problem grows out of hand. For instance the bandwidth requirement in and out of the datacenter will grow exponentially. Also edge devices are often used in a setting where they have access to privileged data, which might pose serious privacy risks when backed up on the cloud.

The challenges mentioned above indicate a clear need for a solution that can learn and adapt to the dynamics of new environments and changing problems on the edge. Figure 1.1 shows an example of such a system wherein a trained model/expert is deployed onto the edge and the agents adapt to the new environment/task autonomously. Each agent uses the deployed expert to perform the task at hand and continues to evaluate its fitness against a rubric as a measure of how well the expert is performing at the task. In the event of a change of task or environment, if the fitness of the expert deteriorates below a certain threshold, the agents invoke the learning process and continue to learn a new expert until the desired fitness is achieved. This new expert is used until another change occurs and learning needs to be performed again.

Reinforcement learning approaches have been successful in tackling the first challenge of developing such an adaptive learning system. In last few year Deep Reinforcement Learning algorithms like DQN, DDQN, Duel DQN, A3C [1, 2, 3, 4] have shown that Deep RL can learn from the environment, without a dataset and surpass humans in complex tasks like playing a games involving complex strategies like DOTA [5], or develop intuitive understanding like Go [6]. RL works by interacting with the environment ie. performing an action in order to make progress towards the objective. Internally a policy function determines the suitable action given the state of the environment. The environment responds by

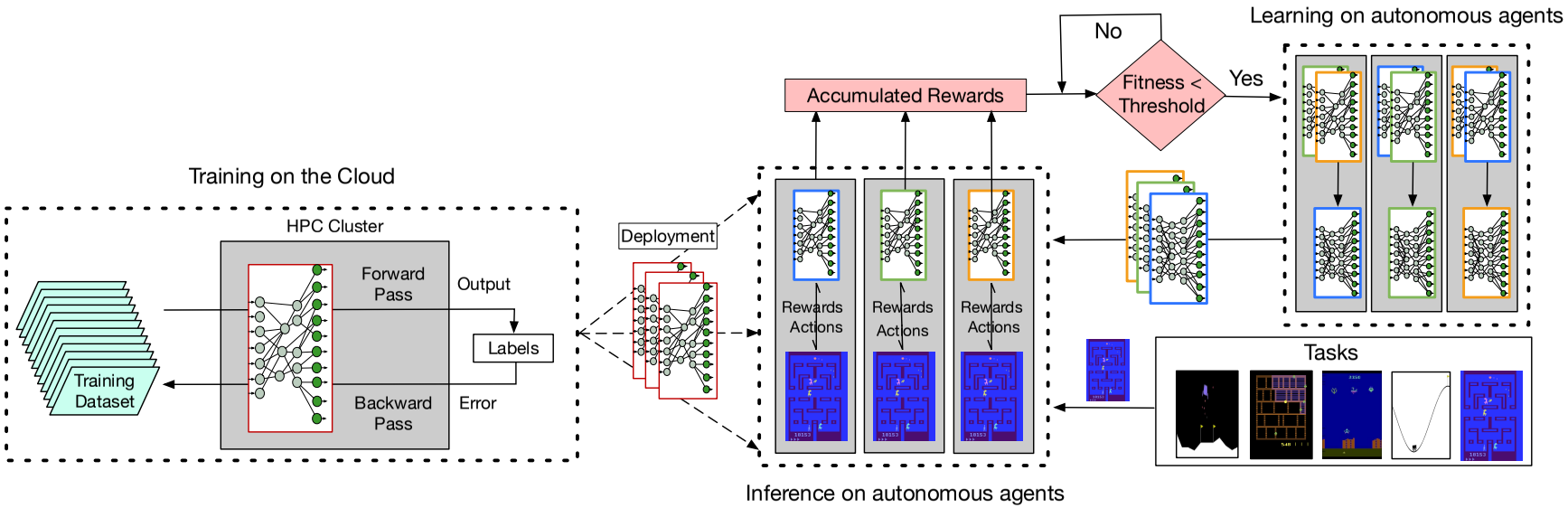

Figure 1.1: Overview of a continuous learning setup

generating a reward value for the action taken. The policy function periodically adapts to maximize this reward. In Deep RL this policy function is approximated by a DNN, and the reward obtained is used as a cost estimate. Therefore each update of the policy function results in training the DNN, which makes Deep RL is extremely costly in term of memory and compute.. An alternative to Deep RL is a set of algorithms called Evolutionary Algorithms (EA). Ea is described in detail in Section 2.1.3. Similar to RL, these algorithms also interact with the environment and adapt to maximize the reward. However unlike Deep RL, each update step does not require back-propagation training but uses Genetic Algorithms instead. A subclass of evolutionary algorithms is Neuro-Evolutionary(NE) algorithm. In a NE setting we start with a *population* of simple neural-networks, which interact with the environment. Each of these networks are then allowed to interact and thus obtain a reward value. The obtained reward are then translated to a fitness value which is then used by the genetic algorithm to evolve a new generation of population which maximizes the reward.

Until very recently Evolutionary Algorithms were seen as inferior to Deep RL and their ability to converge within a reasonable time has been questioned. However recent work from Uber AI labs, DeepMind, OpenAI, Google Brain has shown that EAs scale well. The above reasons including absence of any backpropagation, robustness and ability to adapt to changing evnironments and taks make the study of Neuroevolutionary algorithms and their computation characteristics on hardware an exciting task worth exploring.

This work aims at studying and characterizing the computation of one such algorithm, NEAT. We start by exploring the algorithm and analyzing its performance on commodity hardware such as HPC and embedded CPUs and GPUs. We discuss potential bottleneck to these implementations and the motivate a hardware solution. Next this work discusses the scalability of evolutionary algorithms on a distributed system by performing a scalability study of the algorithm, agnostic to the implementation platform and later also explore the performance of the distributed algorithm on a test bed of multiple Raspberry Pi units. Overall, this work makes the following contributions,

- It characterizes NEAT, a NE algorithm providing key insights to computer architects on how to design efficient systems for such algorithms.
- It analyzes performance of NE on commodity CPU and GPU platforms, identifying bottlenecks and key attributes of the algorithm making single node commodity platforms an inefficient choice.
- As per my knowledge, this work is the first of its kind to explore the distribution of NE on the edge delivering insights to distributed system architects on challenges and opportunities with NE.
- It explore various ways of implementing NE in a distributed setting, within the constraints of edge devices.
- It studies the scalability of each major compute component in NE algorithms and characterize the cost of distributing this compute over multiple nodes.
- It proposes algorithmic modifications that allow NE scale.
- It evaluates various distributed system configurations on a real test-bed consisting of Raspberry Pi units.
- Proposed modifications bring down the share of communication to 22% against a 50% share when naively scaled as is.
- It opens up and discusses future research work and predicts performances in this area based on insights developed by real system evaluations.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Supervised Learning

Conventional machine learning algorithms learn from data. In a machine learning flow, a model is constructed with a fixed set of parameters and initialized with arbitrary set of values. The model is then fed with inputs from a labelled dataset to generate inference outputs. These generated outputs are then compared with the set of labelled correct output values and the difference are recorded. These difference or errors are then formulated into a loss value, which is used to update the model weight using optimization algorithms like gradient descent. These set of steps are called training.

For neural networks the most widely used training algorithm is called Backpropagation(BP). In BP, the parameters of the neural networks are updated by repeated gradient calculation and error calculation starting from the output layers towards the input ones, hence the name *backpropagation*. The quality of training greatly depends on the quality and the size of the dataset. Also, the minimum amount of data required to train a model scales with the number of parameters in the model itself. Although appreciating this fact requires a deeper insight of statistics, a first order intuition is that more variation a model sees, more complex features it could learn.

Deep Learning. Deep learning is a supervised learning setting where the model is a neural network with large number of layers of *deep neural network*. It is natural that with each layer the number of parameters in the model increases. In state-of-the-art neural networks this number is usually in the range if millions. Naturally with models of this size the number of compute operations in inference itself is large and is even larger in training.

Furthermore training DNNs is challenging not only because of the increased compute cost but also because of the demand for large labelled dataset which is why deep learning based solutions have a high entry barrier for new tasks as discussed in chapter 1.

2.1.2 Reinforcement Learning

In layman's terms Reinforcement Learning can be depicted as learning on the fly. In a reinforcement learning setting there is an agent which tries to learn the optimal policy to complete a task by performing repeated actions in a given environment. Each interaction with the environment generates a reward value which encapsulates the effectiveness of the given action. With each such reward obtained the agent updates its policy such that future rewards are maximized.

For the uninitiated policy refers to the mapping of observed state and the action taken. In contemporary RL algorithms like DQN, A3C etc. [1, 4] the policy function is approximated by a Deep Neural Network. This internal neural network is thus trained periodically using backpropagation, with the recorded state-pairs and their corresponding reward values as a measure of loss.

RL on the Edge

Edge devices are particularly constrained on memory and it is thus important to analyze the memory requirement of a learning algorithm to determine its viability. Training in Deep RL, like in DNNs require storing weight parameters and activations as an example forward propagates through the network as these must be retained to calculate the gradients during the backward propagation. As an example, the model described in DQN [1] stores about 1.7 million parameters and computes about 22 thousand activations for each run. Using 32-bit floating point storage, this amounts to a storage requirement close to 7 MB, which is already higher than the typical on-chip available memory storage. As described in the paper, using a batch size of 32 quickly increases the burden of storage to more than 220 MB.

To efficiently utilize the compute power of GPUs, the convolution operations are lowered into matrix multiplications duplicating data adding another factor to storage requirement quickly taking the requirement well over a GigaByte.

Clearly, the memory constraint would make efficiently deploying intelligence on the edge using Deep RL a very challenging task.

2.1.3 Neuro-Evolutionary Algorithms

Neuro-evolutionary algorithms work in the same setting of RL algorithms, where the solution to a given problem is learnt by continuous interaction with the environment. However unlike RL algorithm there is no fixed model which is trained to learn the policy function, but the policy function is evolved using genetic algorithm. NE algorithms start with a population of simple neural networks. The topology and weights of these networks are then tweaked and built upon using the cross-over and mutation operations over multiple generations. In each generation, every member of the population (the neural networks) are given a chance to solve the given problem. A fitness value is then assigned to the members based on how well they perform in the given task. After the fitnesses are obtained for each member of the population, a few fittest members are chosen. These chosen members are then passed to a genetic algorithm which forms new members for the next generation and the cycle repeats over. NE algorithms and their characteristics are discussed in more detail in chapter 3

2.2 Related Work

2.2.1 Distributing DNN computation

Apart from related works on distributed training of DNN [7, 8, 9, 10, 11, 12, 13], there has been significant research in distributing the inference of DNNs across devices. Significant work done [14, 15] has been on high performance computing clusters, typically using many CPU-GPU heterogenous nodes whereas we aim to distribute computation on the edge.

Since edge devices typically face very different compute and energy constraints, the trade-off space is naturally dissimilar. Additionally, edge devices in our work do not rely on any high-speed interconnect which is often present in these systems. Some works such as [16, 17, 18] focus on distributing computation between an edge device(s) and the cloud and are thus dependent on the availability of a cloud service. Along with accelerating inference on edge devices [19, 20] there has been some work done in distributing DNNs on the edge devices [21, 22] but such techniques are dependent on DNN primitives such as convolution layers and cannot be extended directly to the computation graph of NeuroEvolutionary algorithms.

2.2.2 Distribution of EAs

Distributing EAs is an exciting challenge and has been explored by researchers in ORNL, Uber AI [23, 24] but on HPC systems spanning up to hundreds of CPUs. As mentioned before, HPC systems do not operate under the same constraints as an edge device. However, we believe that some insights from our work such as using Asynchronous Speciation can be leveraged even in HPC environments and hence has a broader scope.

2.2.3 Custom HW platforms

Accelerators for DNN inference have been an increasing trend in the research community [25, 26, 27, 28, 29, 30, 31, 32, 33, 29, 34] and ASICs such as GeneSys [35] have demonstrated that there is tremendous parallelism available in EAs and can be leveraged to speed-up computation efficiently. This work leverages that insight to distribute computation across many devices and on the other hand, some insights from our work are agnostic to the implementation platform and could even leverage custom HW platforms as a node in the system speeding up computation significantly and allowing researchers to target even more complex problems with EAs.

CHAPTER 3

NEUROEVOLUTION OF AUGMENTING TOPOLOGIES

NeuroEvolution of Augmenting Topologies is an algorithm developed by Stanley et al. [36] that falls under a class of NE algorithms known as TWEANNs (Topology and Weight Evolving Artificial Neural Networks). I use the NEAT algorithm to drive the motivations and insights in this work, however I expect the same insights to be valuable to other TWEANNs as well. Figure 3.1 depicts the NEAT Algorithm at a coarse granularity and below I define the terminology that will be used throughout to describe various parts of the algorithm.

3.1 Terminology

Gene: This is the basic building block in NEAT, which can be of two types; NN node (i.e. Neuron), or a connection (i.e. Synapse). Each gene has an associated Gene ID and appropriate attributes. For example, connection genes have an associated weight value, an input and output node gene.

Genome: The unique collection of genes that describe one NN topology is known as a genome.

Population: The group of various NN topolgies in a particular generation is known as the population.

Species: A group of genomes with similar NN topolgies are grouped under one species. The process of speciation has been described later in this section in more detail.

Generation: One complete step in the cycle of performing inference, selection and reproduction (as defined below) is known as a generation.

Inference: The step involving evaluating all the genomes in the population for a task at hand.

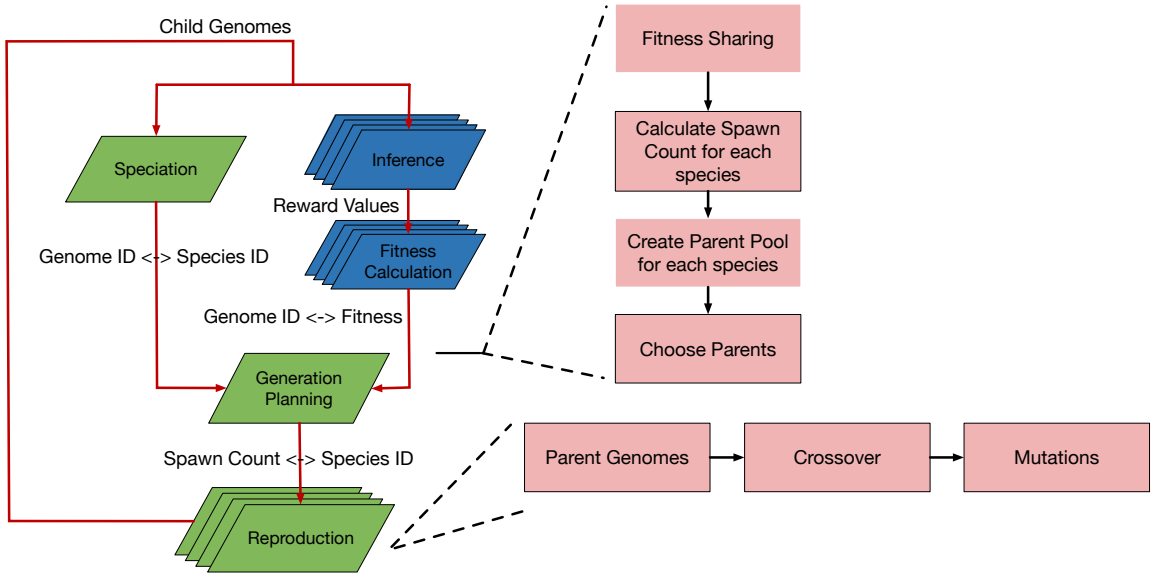

Figure 3.1: NEAT

Selection: The step involving sorting the various members of the population according to fitness, collecting the parent pool for each species, determining the number of children for each species (spawn count) and selecting parent genomes for each child is known as selection.

Reproduction: This step performs the crossover and mutation operations to form children for the next generation.

Crossover: This operation picks attributes from parent genes based on the relative fitness of parents. It is important to note that crossover operations in NEAT can combine any two topologies without any topological analysis using *Historical Markings*.

Mutation: This operation is responsible to tweaking the inherited genes. It is through mutations that genomes of varying sizes and dissimilar structures can be created, leading to the search of an effective topology. Mutations in NEAT are controlled by predetermined probabilities and can be of a few different types described below:

- Add Connection - A new connection gene is added between two previously unconnected nodes.

- Delete Connection - A previously existing connection can be deleted using this op-

eration.

- Add Node - An existing connection is split and disabled and a new node is inserted. Two new connection genes are inserted to replace the disabled connection.

- Delete Node - An existing node can be removed and its associated connections are disabled using this mutation.

Speciation: The addition of new structures might not always immediately result in a better individual/increased fitness as it may need time to optimize. Naively selecting the fittest individuals as parents for the next generation can lead to premature extinction of useful topological features. NEAT speciates the population to protect such individuals and allow them time to optimize their structure before elimination as genomes only compete within their own species.

Fitness Sharing: To ensure any one species cannot take over the entire population, NEAT performs fitness sharing where each genome must share the fitness of their species. Depending on whether this adjusted fitness is higher or lower than the population average, each species grows or shrinks. This new count of individuals for the species is what is termed as *Spawn count* in this work.

3.2 NEAT Breakdown and Analysis

Figure 3.1 represents the flow of the NEAT algorithm broken down into compute and communication. Considering the figure as a graph, the vertices represent compute whereas edges represent data movement. Blue represents Inference, green represent Evolution and Communication is represented by red. It is necessary to understand the compute and communication involved in an algorithm before one can reason about the performance of the same. To characterize NEAT, we analyze the compute costs of the three major compute blocks *Inference, Reproduction and Speciation*. We also analyze the communication that needs to take place between these blocks, represented by the edges in Figure 3.1. Since,

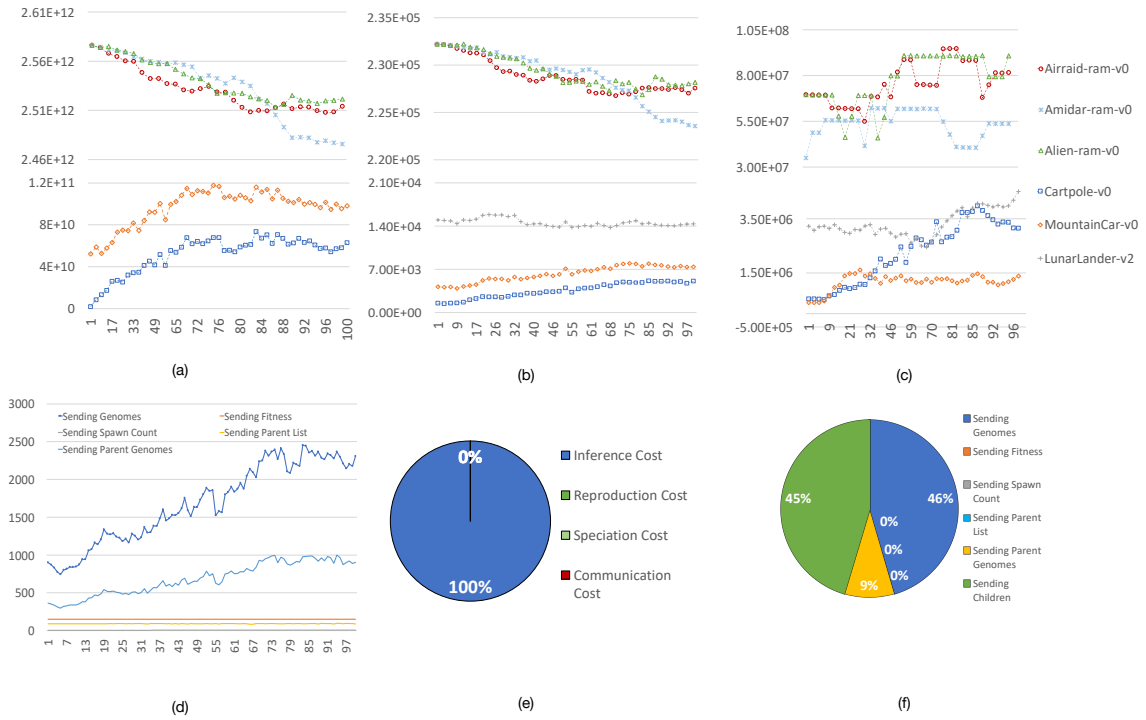

Figure 3.2: Analysis of Compute and Communication Costs in NEAT

compute and communication costs grow proportionally to the number of genes, we use the number of genes processed/communicated by different compute and communication blocks as a measure of cost, effectively making number of genes the metric for evaluation. Figure 3.2(a,b,c,f) shows the trend of various costs in terms of genes across generations for different workloads.

Figure 3.2(e) shows the contribution of each of the four mentioned costs. It is immediately clear that inference dominates by a huge margin and this sort of behavior is not unexpected as will be explained in Section 3.4.2. It is also useful to note from Figure 3.2(a,d) that any communication involving genomes are the most expensive edges in the flow of NEAT seen earlier, contributing more than 99% of communication costs.

3.3 Target Environments

The target environments to the perform the analysis presented in this work have been taken from OpenAI gym [37]. The environments have been carefully selected to represent bigger and more challenging workloads like the Atari games, which have been an important bench-

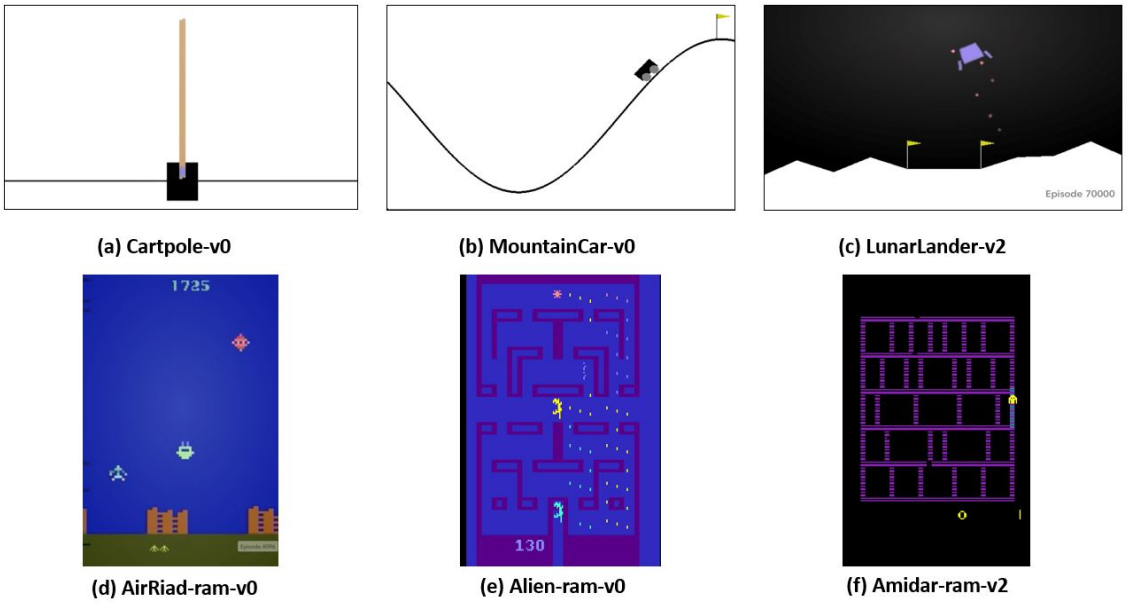

Figure 3.3: OpenAI gym Environments

mark in the RL community in history being used to evaluate evolutionary strategies [38], Deep RL algorithms [1] to Continual Learning [39] and smaller and easier workloads such as Cartpole-v0. I used an open source implementation of NEAT [40] to run the algorithm on the environments described below. Changes were made to the code-base to suit the needs of the experiments and designs described later in this work.

Next I describe in detail the environments used for the experiments detailing the task, the observation and reward space, the fitness functions used etc. The tasks can also be seen visually in Figure 3.3.

3.3.1 Environments

Cartpole-v0

The task is to balance an un-actuated inverted pendulum attached to the top of a cart. The pendulum starts upright and the goal is to prevent it from falling over by applying a force of +1 or -1 to the cart. The observation for the agent is 4 floating point numbers and output is a binary value defining the +1/-1 action. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. The episode

was also limited to 200 time steps in experiments conducted in this work (universal to all environments). The task is considered solved when the pole is balanced for an average of 195 time steps. To evaluate the fitness of an individual attempting to solve the task, the fitness function used is an accumulated reward over all time steps till end of episode.

MountainCar-v0

There is a car on a one-dimensional track stuck between two mountains. The task is to climb the mountain on the right. The car's engine is incapable of scaling the mountain in one pass and the only way to succeed is to build momentum by oscillating between the two slopes. The fitness function used to evaluate fitness uses the velocity gradient between the current and previous time step as a scaling factor to the reward before accumulation.

LunarLander-v2

The goal of this task is to land a module on to the landing pad on the lunar surface by firing the appropriate thruster. The landing pad is always at coordinates (0,0). The episode finishes in the event the lander crashes or comes to rest receiving -100 or +100 points respectively. Ground contact of each leg is +10 points as well. Landing on the pad and coming to rest is 100 to 140 points. The fuel is infinite allowing the module to learn to fly before landing however, there is a -0.3 reward for using the main engine. There are four discrete actions available dictated by one integer number less than 4 and the observation state has 8 floating numbers representing x,y coordinates of the lander, velocity in the x,y direction, angle of the lander, angular velocity and whether or not the two legs have touched the ground respectively. 200 points is considered as solved and the fitness function used here is similar to that of Cartpole-v0.

Atari games

The task here is to learn how to play the Atari games using button presses which is indicated by one integer output. The goal is to maximize the score on the game which is indicated by the reward at each time step. The observation state for these workloads is 128 bytes of the game RAM indicating the current state. I chose three games Airraid-ram-v0, Alien-ram-v0 and Amidar-ram-v0 for evaluating Atari performance. Atari games have been been an important benchmark in the RL community as indicated above and are thus an important workload for analyzing the performance of NEAT. These workloads are evaluated with the fitness function considering the total accumulated reward, same as before.

3.4 Computational Behavior of NEAT

In this section, I present a case study on NEAT which is intended to help drive relevant insights and characterize the computational behavior of NE algorithms.

3.4.1 Accuracy and Robustness

NEAT starts with an initial population of very simple topologies; genomes with only an input layer with nodes equal to that of the observation space and an output layer with nodes equal to the action space of the environment. The network starts with a fully-connected topology and the weights are all initialized to be zero in the beginning but other initialization techniques can be used as well.

The same codebase was used for every environment, changing only the fitness function and the structure of the initial topology to suit the observation and action space between different runs. All the environments achieved target fitness without changing any hyper-parameters involved (such as mutation probabilities) - demonstrating the robustness of the NEAT algorithm.

3.4.2 Compute Behavior

As shown in Figure 3.1, EAs essentially comprises of two broad categories of compute; namely *Evolution* which is running evolutionary learning algorithm to create new genomes every generation and *Inference* which involves running the observation through these genomes and deciding an action. NEAT also comprises of an additional category called *Speciation* which is responsible for grouping topologically similar genomes.
To characterize the compute behavior of NEAT, we look at these three categories in further detail.

Evolution

Evolution can further be split into two parts which are *selection* and *reproduction*. Selection is the process of performing fitness sharing within each species, calculating the spawn count (number of children each species is allowed to produce) and selecting the fittest parent genomes for each child genome. Reproduction performs two classes of computations, i.e. Crossovers and Mutations.

1. Reproduction

   Both crossover and mutations are operations that increase or decrease with the number of genes as can be understood from Section 3.1 and Figure 3.2 represents the amount of operations in terms of genes. A key insight is that each of these operations are independent from each other and can be performed in parallel showcasing a case of raw parallelism. This has been termed as **Gene Level Parallelism (GLP)** in this work.

2. Selection

   Selection is the step that performs fitness sharing and calculates the average fitness of each species and based on this average in comparison to the population average, decides the spawn count for each species as to whether it grows or shrinks. The

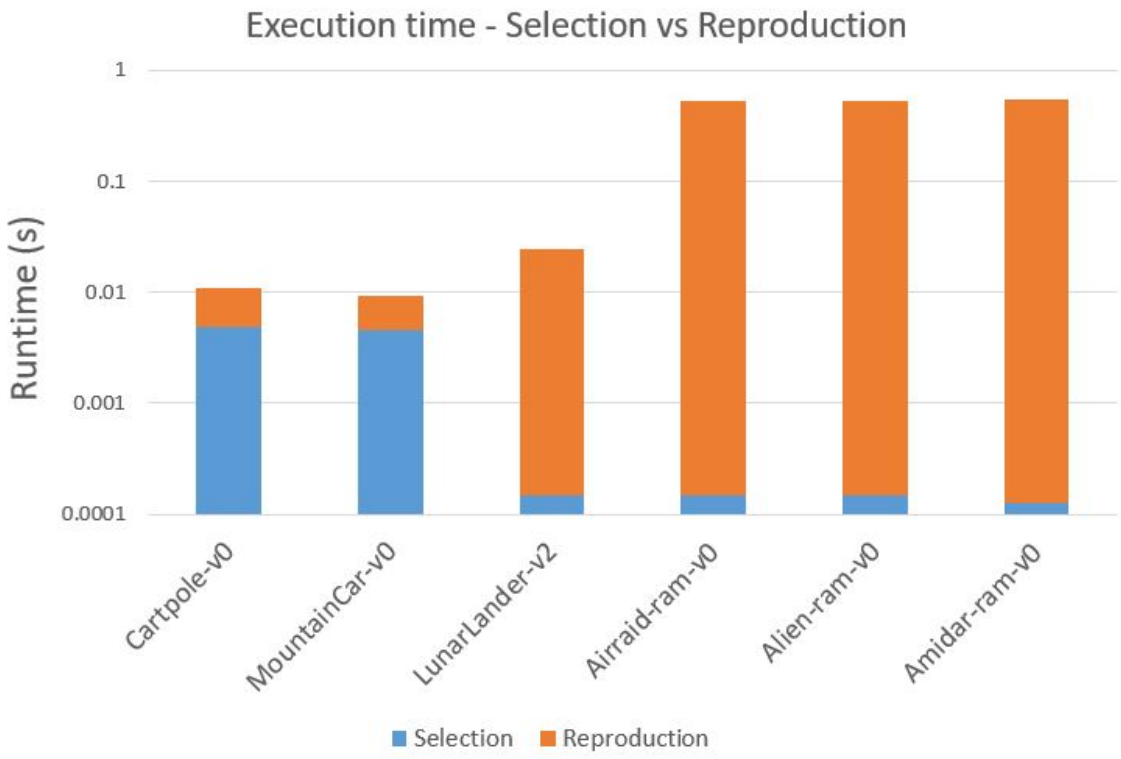

Figure 3.4: Execution Time - Selection vs Reproduction

compute cost of this operation is constant as it does not depend on the structure of any genome, just the population size and their fitnesses.

A cost comparison of the two in terms of raw execution time has also been presented in Figure 3.4. As discussed, the selection cost is independent of the structure and is thus insiginificant for larger workloads where the large structure of the genomes makes reproduction a much costlier operation. It is key to note that this cost ratio of reproduction to selection is also agnostic to the implementation platform. Another key observation is that the reproduction operation to one child genome is completely independent of another, thus showcasing significant availability parallelism. This available parallelism due to the notion of a population with independent genomes has been termed as **Population Level Parallelism (PLP)**. However, selection does not show any such parallelism opportunity as it operates on collective information from all members of the population.

Inference

Inference involves evaluating all the NNs in the population. However, it should be noted that inference in NEAT is different than traditional Multi-layer Perceptron (MLP) NNs. The reason for this being the probabilistic nature of NEAT, wherein nodes and connections are added or deleted based on random probabilities (recall crossovers and mutations in Section 3.1), thereby making genomes generated by NEAT highly irregular and sparse. Another key observation is that, inference of any genome is completely independent of others leveraging PLP similar to Reproduction.

As mentioned in Section 3.3, each environment is run through completion which can occur either through failure or successful completion **OR** the environment has completed 200 time steps (experiments in this work). The input at every time step is the observation of the state of the environment, which is the result of the action taken in the previous time step. Each time step also returns a reward value. Each time step can be seen as a tuple *s,a,r,s'* where s is the current state, a is the determined action after one inference pass, r is the resulting reward and s' is the resulting state (current state for the next time step).

It can be seen now that the same genome goes through multiple inferences attributed to multiple time steps every generation. Due to this nature of inference, it tends to dominate computation as shown in Figure 3.5. It shows execution time of inference normalized to the execution time of evolution + speciation (explained subsequently) for various environments. It can be seen that inference takes 2-3 orders of magnitude more time and indicates a clear need to speed it up. The same result can also be inferred by looking at the compute costs of performing Inference in Figure 3.2(a,e) occupying 99% of computation costs.

Speciation

Speciation is the step of grouping new members of the population each generation into buckets. NEAT performs speciation to protect nascent features that may need time to mature. It is easy to see that *survival of the fittest* policy of EAs can easily wipe out nascent

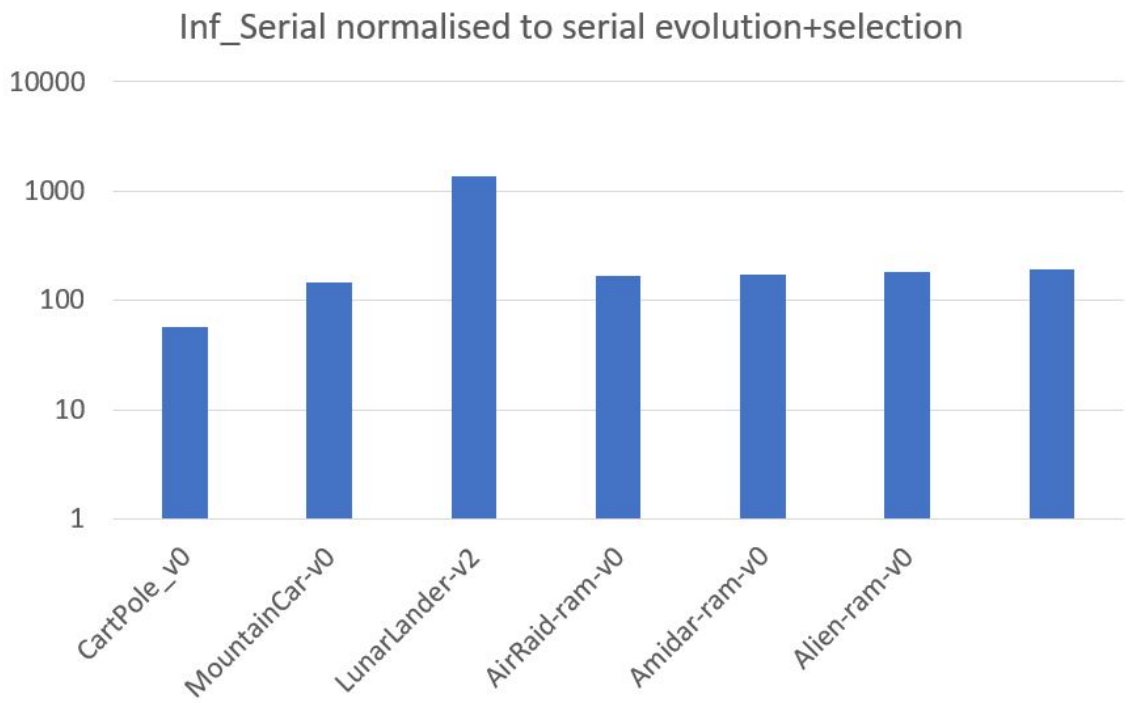

Figure 3.5: Execution time of Inference Normalized to Evolution

features that may or may not have matured in later generations to achieve better fitness. NEAT uses *Genome Distance* to group genomes based on topological similarity. Each disjoint and excess gene contributes to the distance and the higher this value, the less topologically similar the two genomes are considered. Therefore this distance can be measured simply as a linear combination of the number of excess and disjoint genes, as well as the average weight differences of matching genes [36].

Each genome is compared with a representative of each species and genome pairs having a distance less than a tunable parameter, the *Compatibility Threshold* are placed in the same species. If no representative is similar enough, the genome forms its own species and marks itself as the representative. This way NEAT can use historical markings to find topological similarity.

The compute characteristics of comparing different genomes with the representatives and forming species has been demonstrated in Figure 3.2(c) in terms of genes. Number of comparisons made every generation was recorded for various workloads and was used along with the average number of genes each generation to compute these characteristics.

Also, to prevent any one species from taking over the entire population, each genome must share the fitness of its entire species, which has been referred to as *Explicit Fitness Sharing*.

In other words, speciation in NEAT allows for sufficient exploration. The *Exploration-Exploitation dilemma* has been a constant discussion in the RL community and it should be noted that this exploration is not done by suppressing exploitation. After the step of fitness sharing, the fitter species are allowed to grow whereas the weaker ones shrink while ensuring safety from annihilation. This allows exploitation along with sufficient exploration.

Thus, speciation is key to the performance of the algorithm. However, Speciation does not enjoy the same kind of parallelism as Reproduction and Inference and is thus a serial step requiring the topology information of all genomes to function. This limitations becomes crucial as becomes evident in the upcoming discussions.

3.5 Evaluations on Commodity Hardware

After characterizing the compute and communication costs of NEAT, the next step in exploring opportunities and challenges to NE deployment is to evaluate performance of the algorithm on commodity hardware. In the next few sections, I implement and analyze NEAT on various heterogenous platforms and attempt to identify bottlenecks to performance and its reasons.

3.5.1 Methodology

Metrics such as energy, runtime and memory footprint were studied and analyzed on embedded and desktop class CPU and GPU platforms. Using the NEAT-python code base [40] and modifying it as needed to exploit available parallelism, different implementations mentioned in Table 3.1 were studied.

PLP - Population Level Parallelism BSP - Bulk Synchronous Parallelism

Table 3.1: Target System Configurations

Legend	Inference	Evolution	Platform
CPU_a	Serial	Serial	6th gen i7
CPU_b	PLP	Serial	6th gen i7
GPU_a	BSP	PLP	Nvidia GTX 1080
GPU_b	BSP + PLP	PLP	Nvidia GTX 1080
CPU_c	Serial	Serial	ARM Cortex A57
CPU_d	PLP	Serial	ARM Cortex A57
GPU_c	BSP	PLP	Pascal
GPU_d	BSP + PLP	PLP	Pascal

3.5.2 CPU Evaluations

I measure the completion time and power measurements on two classes of CPU platforms, desktop and embedded. The desktop CPU is a 6th generation Intel i7, while the embedded CPU is the ARM Cortex A57 housed on Jetson TX2 board. On desktop, power measurements are performed using Intels power gadget tool while on the Jetson board I use the on-board instrumentation amplifier INA3221. I capture the average runtime for evolution and inference using the codebase, and use it to calculate energy consumption.

Two flavors of NEAT were used to evaluate CPU performance on each of the two platforms, embedded and desktop. CPU_a and its embedded counterpart CPU_c are serial implementations of NEAT. In CPU_b (CPU_d for embedded), the multiprocessing library is used to leverage ***PLP*** by parallelizing inference utilizing 4 cores.

Figure 3.6 shows the runtime and energy for inference for the various OpenAI gym environments described above on different platforms. It can be seen that CPU_b and CPU_d are 3.5x times faster when implemented using 4 cores than their serial counterparts CPU_a and CPU_c respectively. Such a speed-up despite having a serial evolution step is expected as we have seen earlier (refer to Figure 3.5) that inference tends to be orders of magnitude slower than evolution and the massive parallelism available.

Similar analysis for evolution can be seen in Figure 3.7 for the two different platforms.

.

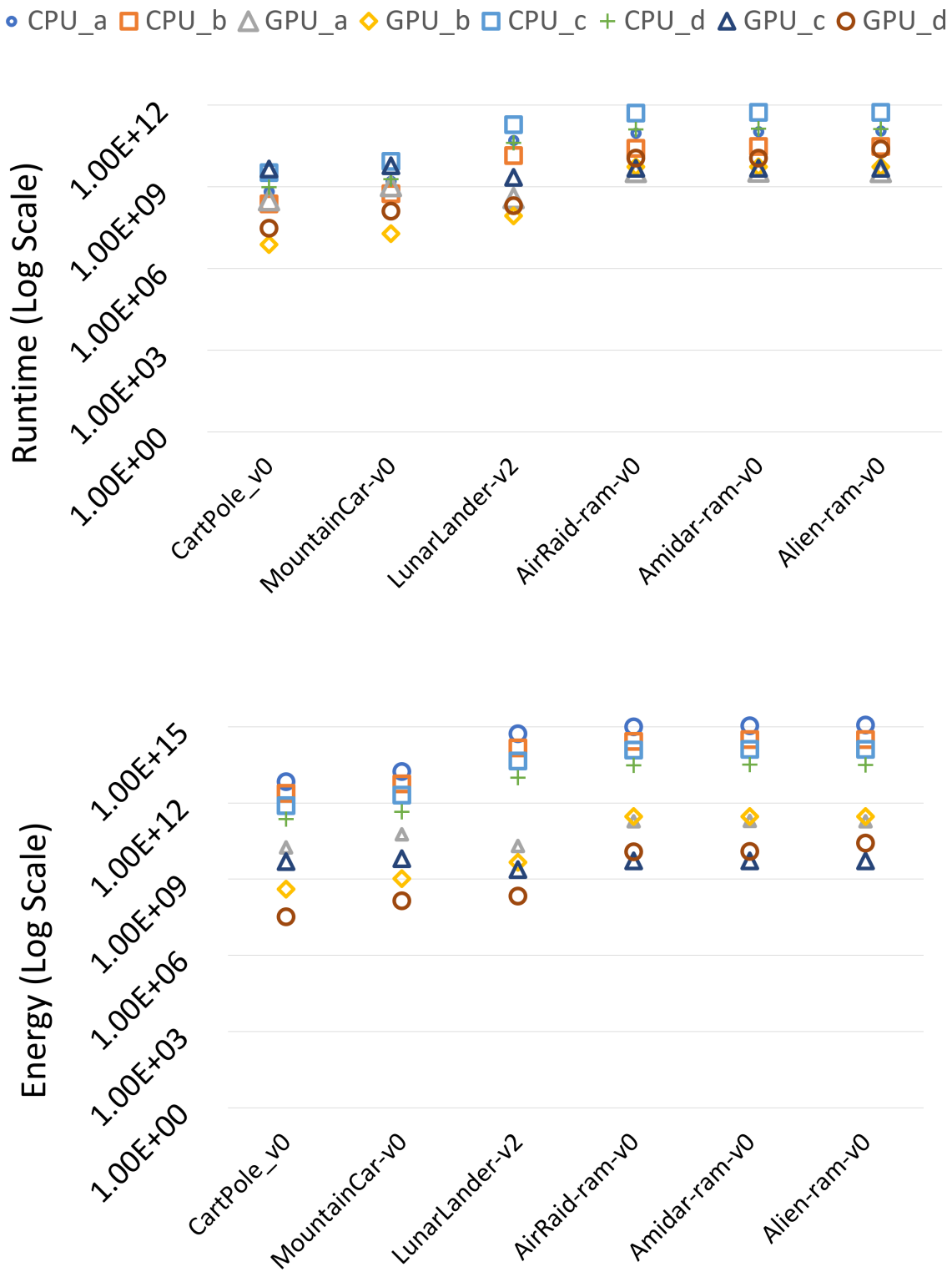

Figure 3.6: Inference Runtime and Energy on various platforms

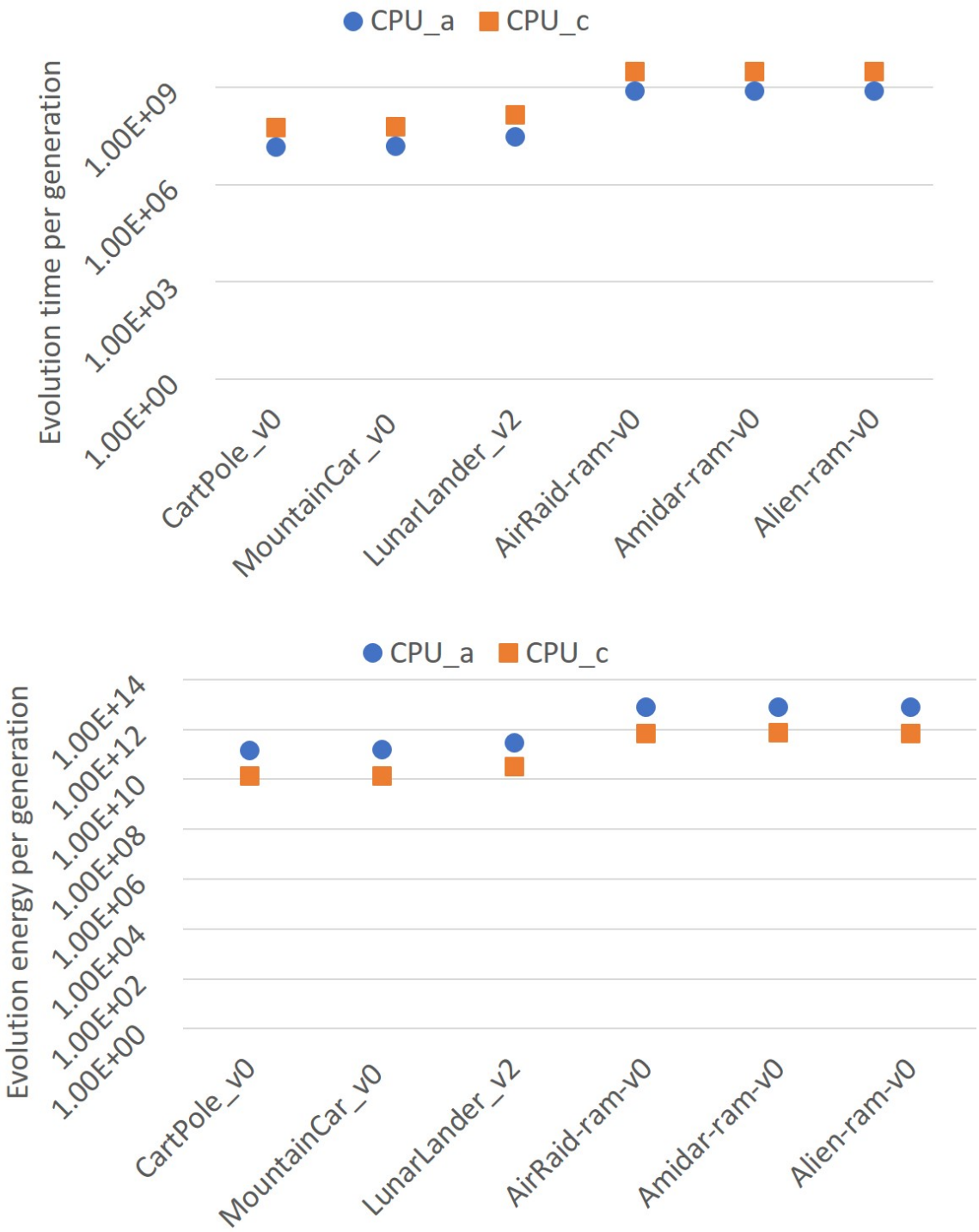

Figure 3.7: Evolution Runtime and Energy on various platforms

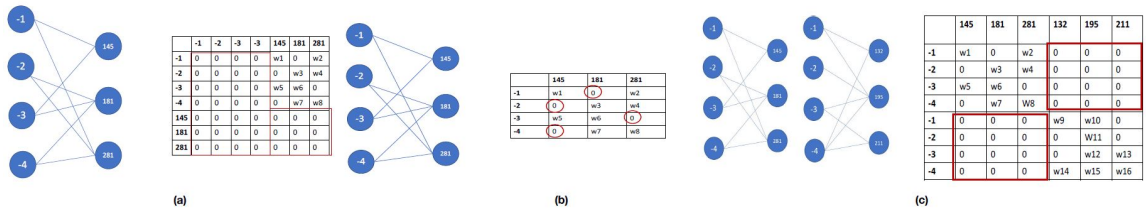

Figure 3.8: Different representations for Inference on GPUs

3.5.3 GPU Evaluations

The massive parallelism available in GPUs coupled with the massive parallelism opportunities of NEAT are naturally compatible. We implemented inference using PyCUDA and evaluated it on the Tegra GPU on the NVIDIA Jetson X2 board for the embedded class and a GeForce GTX1080 for the desktop class. Two flavors of inference were implemented as well which have been explained in further detail below.

Figure 3.8(a) shows a naive representation of a layer in a genome; the popular adjacency matrix. It can clearly be observed that this representation is highly sparse. Along with the obvious disadvantages of sparsity, it suffers from redundant multiplications. We define redundant multiplications as those multiplications those which would have either (1) not been possible despite a fully connected NN layer, such as row -1 to column -1 as loops are not possible or (2) reverse connections such as row 145 to column -2 as the graph is directed strictly from left to right (Not true for recurrent neural networks) or (3) connections within a layer such as row 145 to column 181. Sparsity due to the layer not being fully connected is referred to as multiplication with 0s hereafter. It is clear that the adjacency matrix representation leads to both redundant multiplications and multiplication with 0s. Therefore we chose two custom representations for our GPU analysis.

Two flavors of inference were implemented to utilize the Bulk-synchronous parallelism offered by GPU programming model. The first implementation evaluates multiple vertices of a genome in a layer in parallel by mapping the computation on all incoming connections to the vertices to different threads using a vector-matrix multiplication as shown in Figure 3.8(b). It can be seen that the size of these matrices leave the GPU largely underuti-

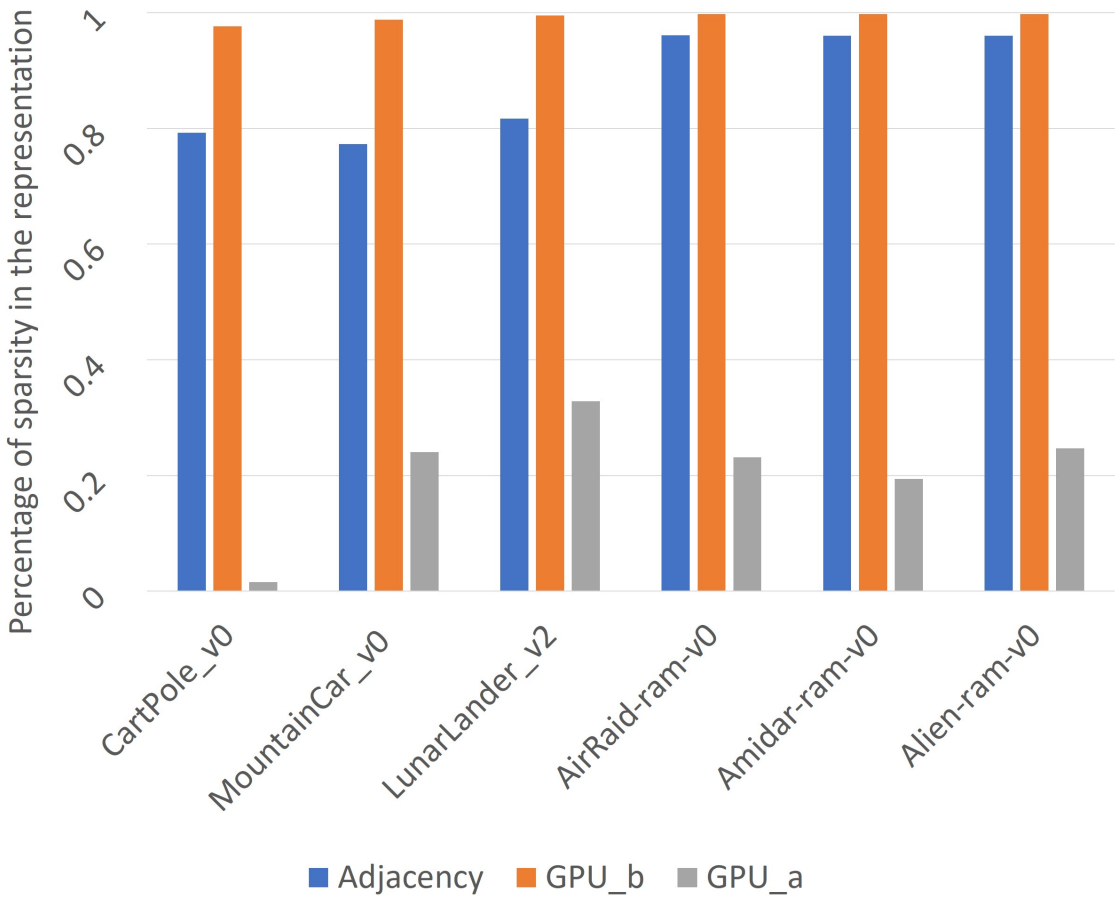

Figure 3.9: Average Percentage sparsity induced in the different representations

lized, especially the smaller workloads as it can support many more threads. This characteristic along with other issues such as extensive API calls resulting from having multiple genomes in a population inspires the second implementation which evaluates multiple vertices across all genomes in a layer in parallel as shown in Figure 3.8(c). However, this does introduce significant sparsity in the matrix. Figure 3.9 shows the sparsity of the two approaches along with the adjacency matrix representation. Note that the first implementation is not free of sparsity because of the irregular structure of the neural network. Also note that the adjacency matrix considered is for one genome at a time. Having multiple genomes in one matrix will have extremely high sparsity.

Figure 3.11 shows the breakdown of runtime on the GPU depicting contribution of memory transfers and the kernel execution time. We can see in Figure 3.10 that 70% of

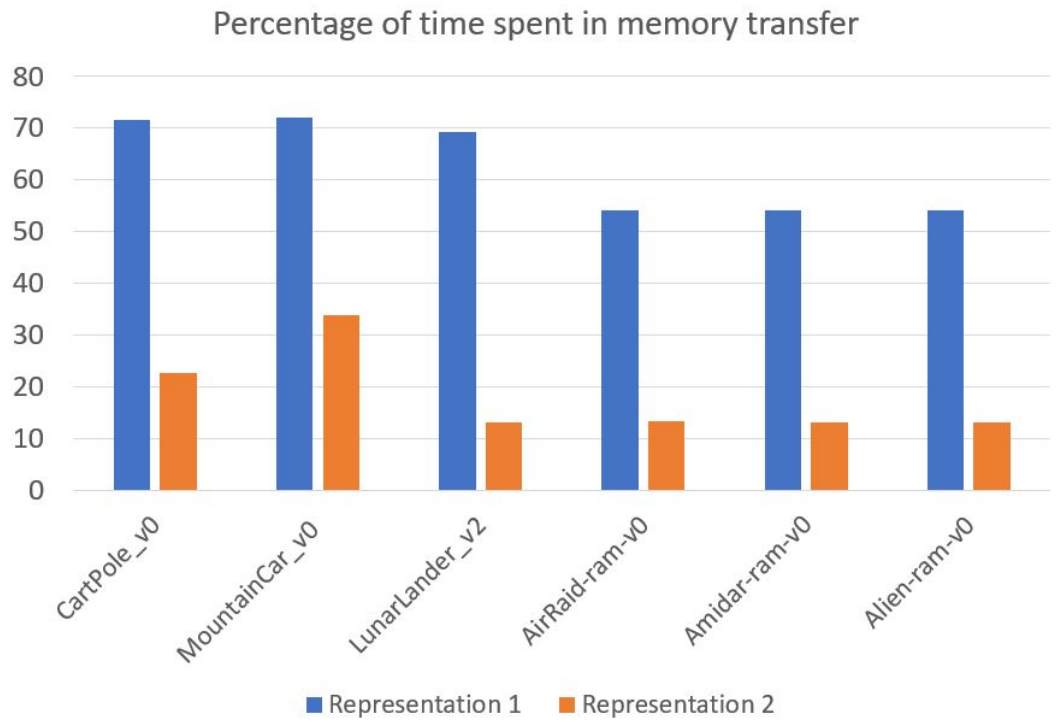

Figure 3.10: Percentage of time spent in data movement for various representations

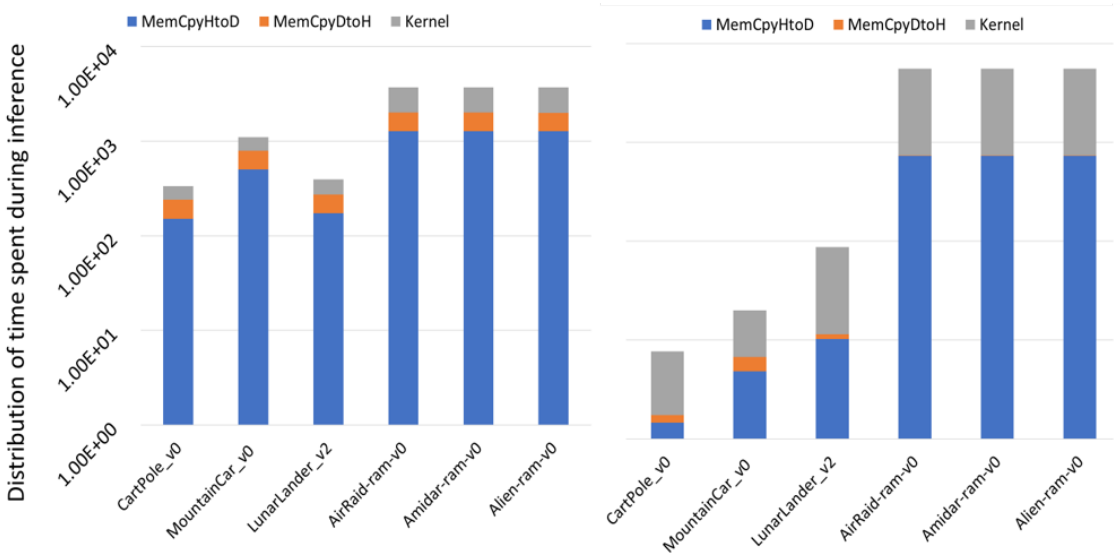

Figure 3.11: Breakdown of execution time on GPUs under the two representations (a) Representation 1 and (b) Representation 2

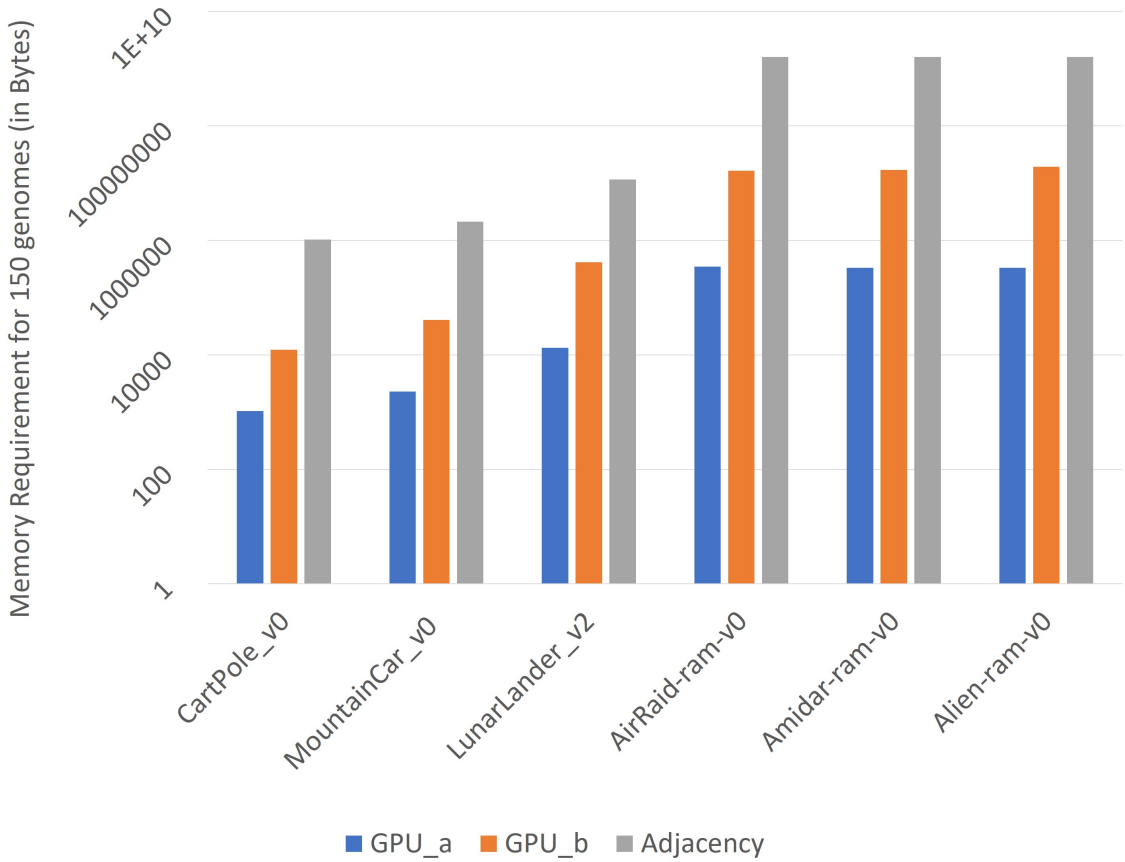

Figure 3.12: Average memory requirement by different representations

time is spent on memory transfers in the first implementation whereas it is only a maximum of 30% in the second implementation owing the reduced memory transfer calls as calls for each genome are combined into one. It is important to note that the breakdown here does not consider contributions from other sources such as pointer transfer among others.

Figure 3.12 shows the average memory requirement by of these matrices across the workloads showing that the memory requirement can be fulfilled even by a modern on-chip memory making EAs favorable to Deep RL networks which suffer from intensive memory requirements due to Back-propagation.

We turn attention to Figure 3.13 which shows all 4 flavors of inference discussed so far vs the slowest evolution. Though we notice the GPU implementation of inference approaches the speed of evolution computation, we still notice that for all workloads barring

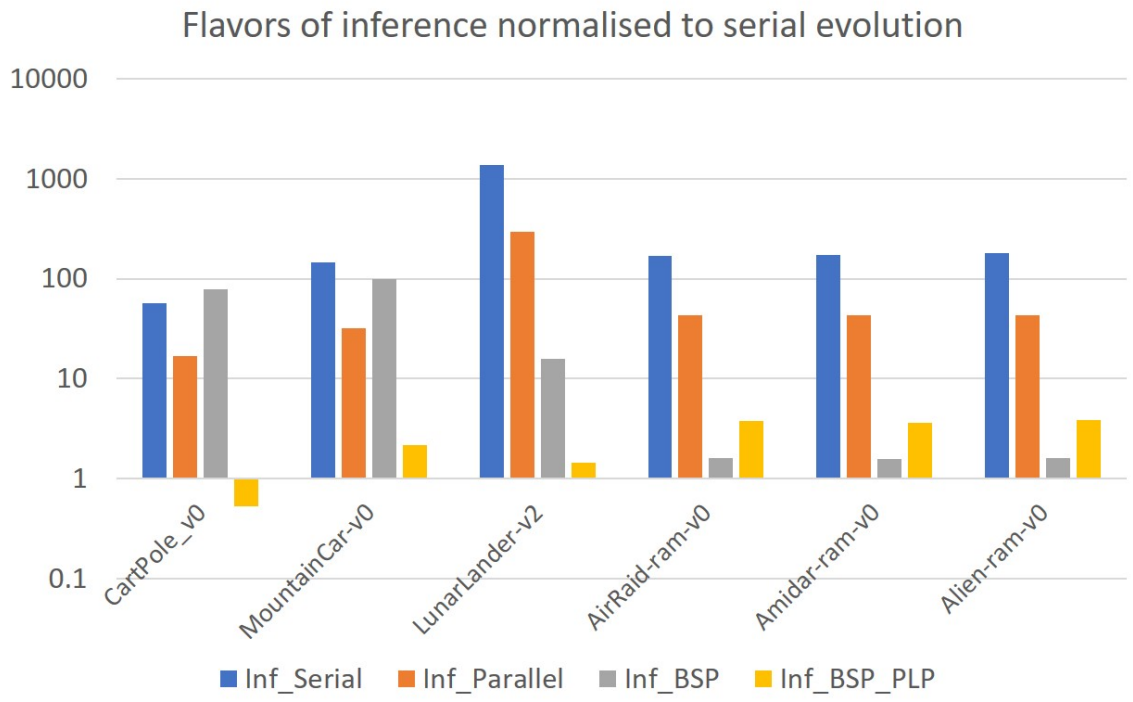

Figure 3.13: Runtime of different flavors of Inference implemented compared against serial evolution

cartpole_v0, evolution still remains faster. Also as seen earlier in this section, GPU implementations suffer from a multitude of other issues that urge us to explore other options.

.

3.6 A Case for Hardware Acceleration

As discussed, we can see the numerous problems with using commodity hardware for an algorithm like NEAT. CPUs are slow and perform extremely poorly on the EDP front. GPUs on the other hand are faster and offer significant acceleration but are still an inefficient solution to the NE problem. This is due to the many small irregular and sparse networks that go through inference in NEAT, both of which are not friendly to efficient GPU processing using the BSP model. GPUs also spend significant time in initiating and performing memory transfers. It is key to note that the GPU implementations can possibly be made more efficient by writing heavily customized kernels but that is beyond the scope of this work and this work focuses on using available open source libraries for GPU computations.

We have also seen various characteristics of the algorithm earlier in this section. Given the large amounts of parallelism available in the form of GLP and PLP with the added advantage of having significantly lower memory foot print (less than 1 MB for the workloads in consideration) than traditional Deep RL algorithms such as DQN, it is evident that custom chip with a sufficiently sized on-chip memory can help remove/reduce off-chip accesses, save both energy and bandwidth while significantly accelerating the performance. Also, the compute operations in EAs (crossovers and mutations) are simple and hardware friendly. Furthermore, the absence of gradient calculation and significant data movement overheads facilitate scalability [35]. The small and irregular structures of the networks generated by NEAT also call for a new choice of hardware platform.

As motivated in chapter 1, the goal of this work is to enable the evolution of complex behaviors in mobile autonomous agents which are limited both by area and energy constraints. If we can reduce the energy consumption of the compute operations by implementation in hardware, pack a lot of compute engines in a small form factor, and store as much as possible of the genomes on-chip and reduce cost of data movement, this goal can be achieved.

GENESYS [35] is a SoC aimed at realising this goal in hardware. To summarise some key results from the work; it can be seen GENESYS outperforms the best GPU implementation by over 100x in inference. ADAM (Inference engine of GENESYS) contributes to 100x more energy efficiency, while EVE (Learning/Evolution engine of GENESYS) turns out to be 4 to 5 orders of magnitude more efficient than GPU_c, the most energy efficient among commodity platforms.

CHAPTER 4
COLLABORATIVE LEARNING USING ASYNCHRONOUS NEUROEVOLUTION

Massive and easy availability of custom AI accelerators that fit the constraints of the edge might not be a distant reality but such a situation is certainly not the scenario today. Bringing true intelligence that is adaptive and robust to the edge on commodity available hardware can rapidly change the the way we experience AI today. Our aim through this work is to enable solutions and ideas that further enable solutions which can use available off-the-shelf hardware for the intelligent edge.

4.1 Target Setup

We have already seen why EAs should be the algorithm of choice when trying to deploy intelligence on the edge. The massive parallelism available in NE and typically low compute capabilities of a single edge device call for a distributed solution to the problem. The notion that there exists a population of problem solvers in EA that each tries to learn the solution also lends itself to a distributed setting. By virtue of distribution, different problem solvers on the edge will experience different scenarios and learning collaboratively can accelerate the learning process. Such deployment of collaborative learning on the edge can result in a mass proliferation of autonomous robotic swarms capable of adapting to a new problem setting in a robust manner. We present *CLAN - Collaborative Learning using Asynchronous Neuroevolution* and aim to take the first step in bringing true intelligence to the edge.

4.2 Hard Scaling

As we have seen in Figure 3.1, there are three major compute components - *Inference, Reproduction and Speciation*. The authors of [35] characterize NEAT and discuss the raw parallelism opportunity in NE algorithms and define two new terms

- Population Level Parallelism (PLP) - The parallelism offered by multiple agents in the population, each of which are independent.
- Gene Level Parallelism (GLP) - The parallelism offered by the independent crossover and mutation operations.

Of the three main compute components listed above, Inference and Reproduction can leverage PLP and could be performed in parallel in a distributed setting, potentially improving performance. To characterize and identify the areas that stand to gain the most by distribution, we analyze the compute costs of the three major compute blocks. We also analyze the communication that needs to take place between these blocks, represented by the edges in Figure 3.1.

Figure 3.2(e) shows the contribution of each of the four mentioned costs. It is immediately clear that inference dominates by a huge margin and this sort of behavior is not unexpected due to each inference step being performed over multiple time steps as noted in Section 3.4.2. Therefore, this becomes our first candidate for distribution. It is also useful to note from Figure 3.2(a,d) that any communication involving genomes are the most expensive edges in the flow of NEAT seen earlier, contributing more than 99% of communication costs.

Remember that in a distributed setting, the actual cost of communication in terms of time and energy is much higher. It is evidently clear that the partitioning of the algorithm across the nodes will define the costs to be paid and subsequently the scalability of the design.

In the next couple of sections we try to asses the potential gains and try to identify the cost

involved while distributing various blocks in the algorithm. While naming different configurations possible of the distributed system, we follow the convention CLAN_IRS where I(Inference) and R(Reproduction) can either be distributed(D) or central(C). Speciation(S) can be performed synchronously(S) or Asynchronously(A).

4.2.1 CLAN_DCS - Distributing Inference

In every generation each member of the population interacts with the environment to attain its fitness score. This leads to multiple inferences owing to multiple time steps as mentioned in Section 3.4.2 raising the compute costs of performing inference. However there is no dependence between the inferences across genomes and thus could be performed in parallel essentially leveraging PLP. This motivates our first design choice CLAN_DCS, where inferences for multiple genomes are performed concurrently in a distributed fashion. Therefore, the inference step is distributed while reproduction and speciation are performed centrally and synchronously respectively.

To achieve this, an additional step is now introduced which involves sending out genomes formed by reproduction to multiple computing agents for inference and subsequently gathering back the fitness values for every genome once inference steps are completed. The configuration and time-line of compute/communication followed in such a setup has been shown in Figure 4.1.

It is easy to observe that at scale, it won't take long for Amdahl's law to catch up. An implementation with distributed inference, though a good start can only be as fast as the serial steps. Moreover, there does not exist a necessary condition of repeated inference over multiple time steps reducing the compute needs of inference and making them equivalent to the needs of performing the evolution step. This motivates us to look at the next block of compute that can leverage PLP: Reproduction.

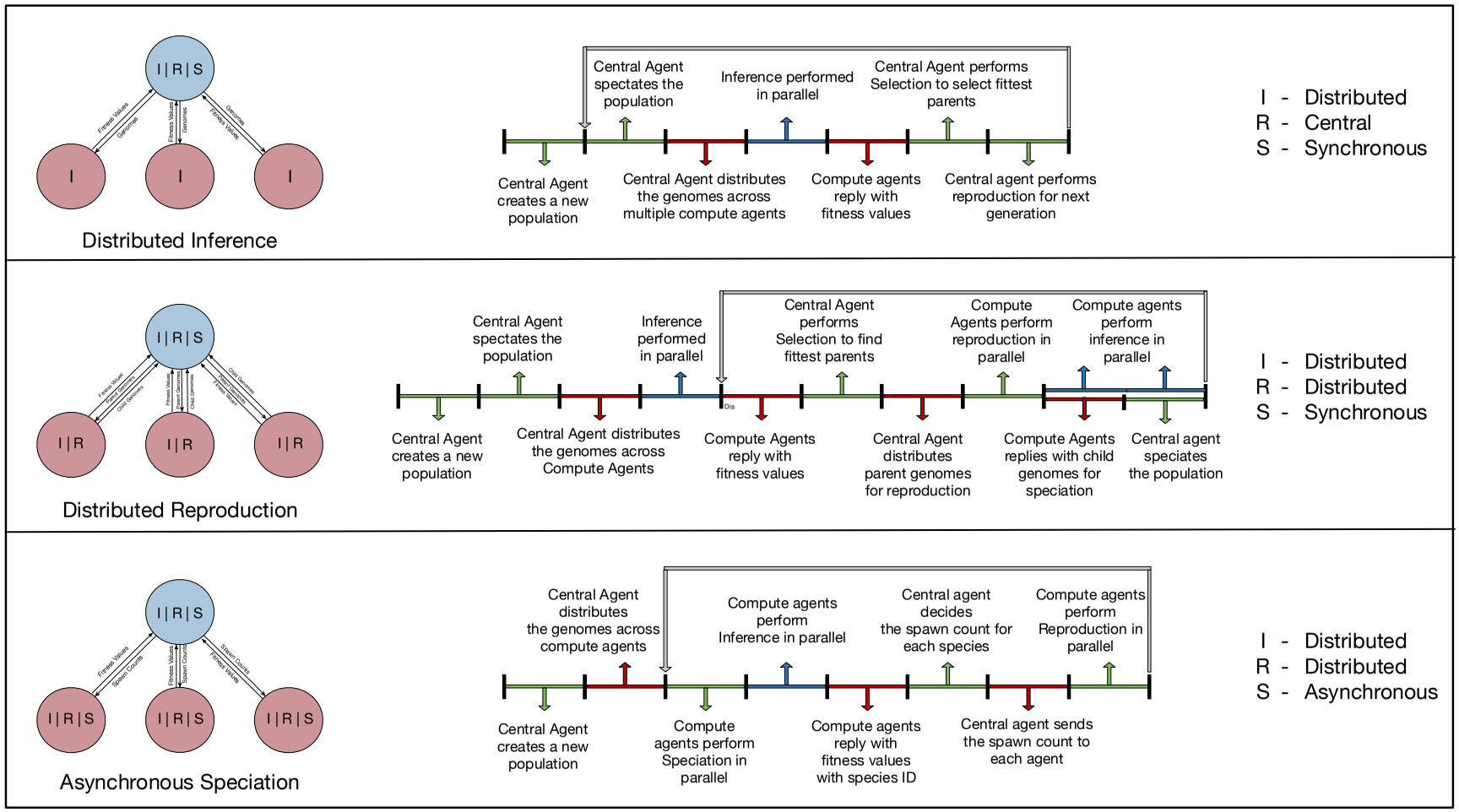

Figure 4.1: Proposed Distributed System Configurations

4.2.2 CLAN_DDS - Distributing Reproduction

We now move on to the next candidate that can exploit parallelism - Reproduction in CLAN_DDS where along with inference, reproduction is also distributed. We distribute the reproduction step by forming children across agents in parallel. The configuration time-line of this implementation can be seen in Figure 4.1. We can see the additional communication over CLAN_DCS which is due to speciation. The step of speciation needs to see the genome structure to appropriately speciate and therefore needs the communication of all genomes from agents to a central node. Additionally, the central agent also needs to communicate parent genomes to agents for reproduction as it is not a necessary condition that the fittest genomes chosen as parents are available on any given agent. Hence, a choice attempting to naively scale reproduction involves higher communication costs due to the repeated back and forth of genomes between the agents and center. Whether this cost inhibits scaling of Evolution is a question we examine in Section 5.1.

4.3 Soft Scaling

However, it is immediately clear that despite forming and evaluating child genomes on a single node, the communication costs do not seem to reduce but rather counter-intuitively increase. The share of each of the different possible transactions (Refer Figure 3.2(d)) indicates that the transfer of genomes pays the heaviest cost. As we can see from Figure 4.2, majority of the cost involved in CLAN_DCS and CLAN_DDS is due to the communication of genomes including sending children genomes for speciation and parent genomes for reproduction. Reduction/removal of this cost could considerably improve the performance of the algorithm by reducing communication overhead.

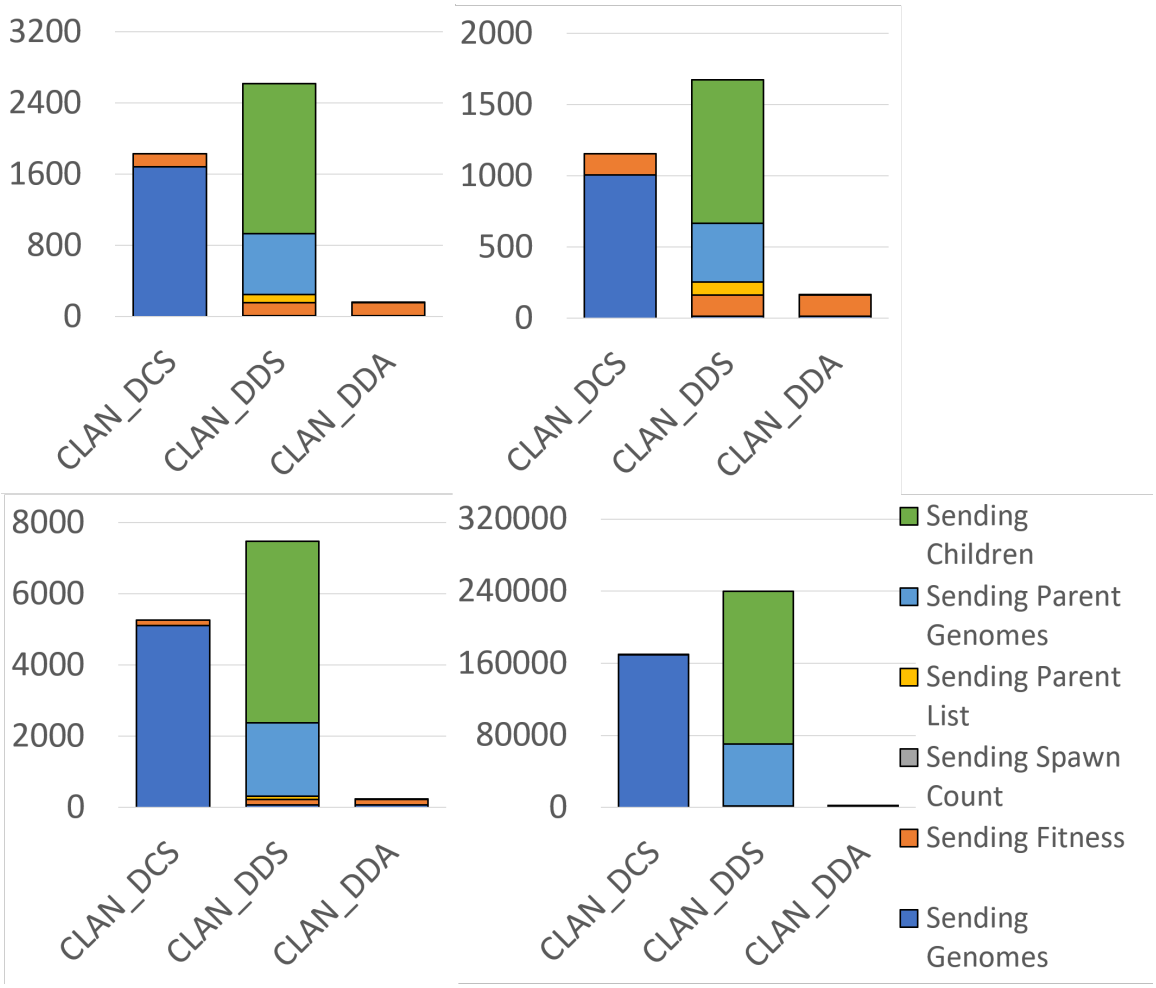

Figure 4.2: Breakdown of Communication Cost for various configurations

4.3.1 CLAN_DDA - Asynchronous Speciation

As noted before, localized speciation is the only major serial component that cannot exploit any parallelism. Furthermore, it is the reason behind the elevated costs of distributing reproduction. To overcome this limitation, we propose *Asynchronous Speciation (AS) aka Asynchronous NeuroEvolution* in CLAN_DDA where inference and reproduction are distributed and speciation is performed asynchronously. AS refers to speciation performed on small *clans* of members of the population independently instead of entire population itself. Using clans, we allow multiple agents to perform independent speciation. The configuration setup and time-line for this design choice can be seen in Figure 4.1. As can be seen, there is no communication of genomes in this design after the necessary initialization. Al-

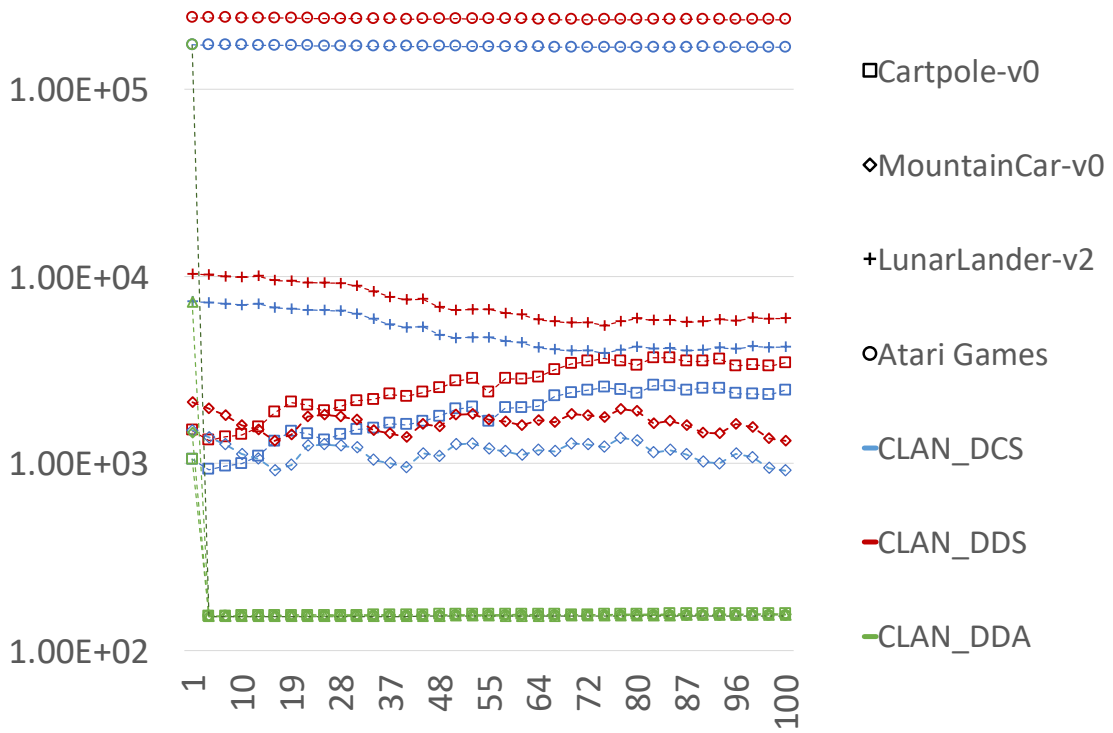

Figure 4.3: Communication Cost for different design choices

ternatively, we can also visualize this in Figure 4.2, the communication cost is the least for CLAN_DDA across the workloads.

Figure 4.3 shows the behavior of the total communication cost involved in the three proposed designs over generations. We see that the communication cost for CLAN_DDS is higher throughout for each workload as expected. It is significantly worse for larger workloads due to the larger genome structures. However, performing AS in CLAN_DDA has to pay the cost of communicating genomes only in the first generation and then continues to pay negligible cost than other two design choices.

CHAPTER 5
EVALUATIONS AND DISCUSSION

5.1 Evaluations

5.1.1 Methodology

In order to evaluate the design choices proposed above, we constructed a test bed of 15 Raspberry Pi (RPi) units connected in a distributed setting. A local WIFI network with the measured bandwidth of 62.24 Mbps and a measured client-to-client latency of 8.83 ms for 64 B is used. This allows us to evaluate the designs within the constraints of the edge and observe the scalability in a real system.

As described earlier, we choose OpenAI gym [37] workloads for our evaluations. To evaluate the scalability of our distributed system, we start with 2 RPi units and increase the number of units gradually. We refer to each of these Raspberry Pi unit as an agent. In the following text, we discuss our evaluations using different configurations.

5.1.2 Experiments

CLAN_DCS

We plot the runtime for performing inference on various workloads with varying number of RPi units in Figure 5.1. We see that for smaller workloads such as Cartpole-v0 and Mountaincar-v0, scaling stops earlier at 5 and 10 agents respectively. Other heavier workloads continue to scale till we exhaust our test bed of 15 RPi units. It is particularly interesting to note that for larger workloads, due to the massive amount of inference compute available, the speed-up attained is almost linear and equal to the scaling factor initially and slows down only mildly in the much later stages of scaling.

To evaluate the reason why smaller workloads like Cartpole-v0 stop scaling, we mea-

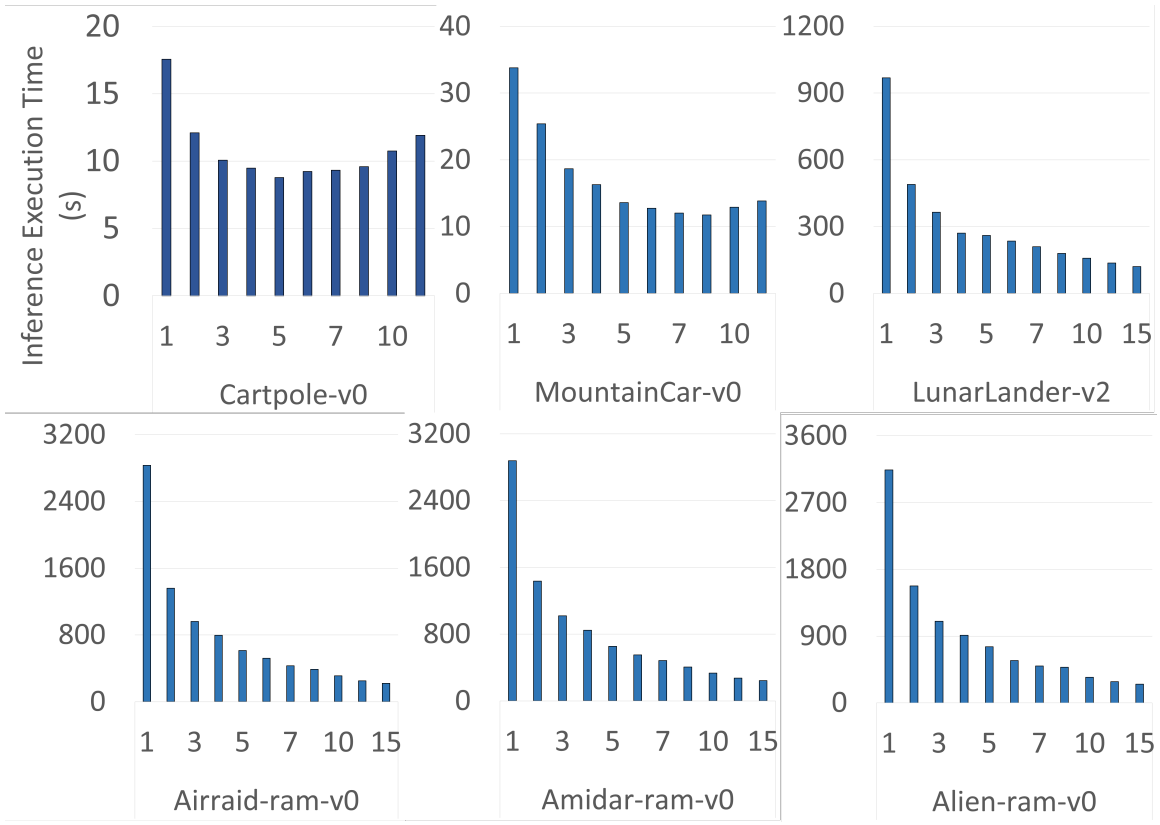

Figure 5.1: Execution Runtime at scale for Distributed Inference

sure the breakdown of inference time into raw inference compute and time spent in communication in Figure 5.2 at 5 and 6 agents, i.e. before and after scaling ends. As expected, one can see that communication time starts dominating when scaling stops. The raw inference compute time continues to scale but is defeated by the increasing communication overhead. The cost of communicating genomes overtakes the benefit of parallelism due to the low compute in such workloads as the genome size remains fairly small, even after many generations of evolution.

CLAN_DDS

We now implement the CLAN_DDS configuration and measure runtime of performing reproduction in parallel. Inference continues to scale as before, however evolution is unable to scale even at 2 RPi units and progressively gets worse if attempted to scale further as shown in Figure 5.3. The reason for this is the increased communication as expected in

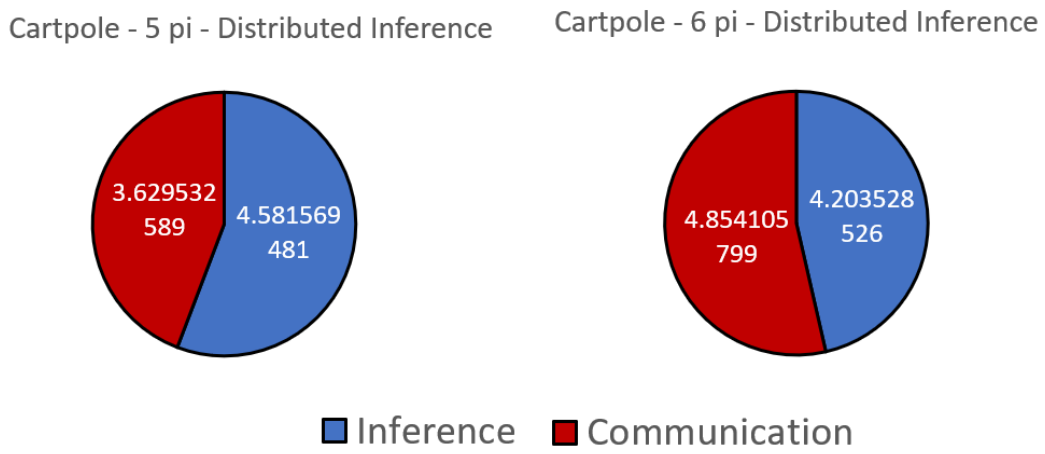

Figure 5.2: Share of Inference Compute and Communication in Inference Time

Figure 4.2. We see the breakdown of execution time taken by evolution into raw evolution compute (including reproduction, selection and speciation) and communication in Figure 5.3. The time taken to perform evolution increases 2x even at the scale of 8 units at which point the evaluations are stopped. The raw evolution compute scales similarly to inference and is capable of leveraging PLP but it can be immediately seen that the communication dominates the runtime and starts to dominate further as scaling is attempted.

As discussed before, the choice of partitioning the algorithm dictates what edges pay the higher communication cost of transmission between agents. Therefore, we move to the final configuration CLAN_DDA that performs Asynchronous Speciation to significantly reduce the cost of communication and evaluate its performance

CLAN_DDA

We measure the runtime of performing inference and reproduction in parallel while performing asynchronous speciation The runtime plot showing the breakdown between communication and raw evolution compute in the evolution phase can be seen in Figure 5.4. Looking at the larger workloads where evolution contributes meaningful amounts, we can see how the trend of reducing raw evolution compute time continues, similar to CLAN_DDS, however, the communication cost is not prohibitive anymore and allows evolution to scale

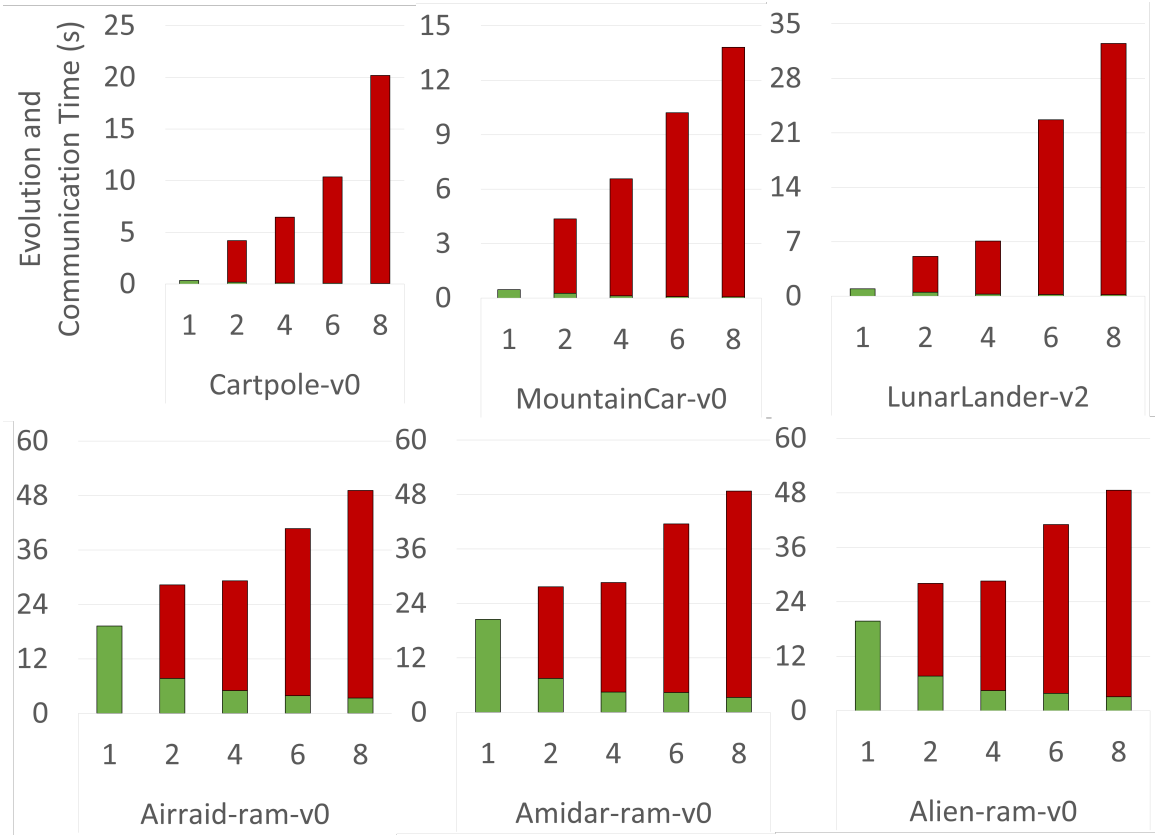

Figure 5.3: Execution Runtime at scale for Evolution and Communication using Distributed Reproduction

as well as Inference, similar to CLAN_DCS.

One can observe that the inflection point in the scaling of Evolution arrives far before that of Inference, but it is however, important to remember that Inference compute is significantly higher due to repeated inference for reward collection. Hence, observing the total time taken by learning process combining Inference and Evolution is key. Also, It is important to note the scale at which Evolution is able to operate under this configuration, less than 5x of that of Distributed Reproduction. We notice significantly less communication overhead when compared to previous designs allowing higher scalability, especially for potentially larger real-life workloads.

This configuration allows Evolution to scale unlike CLAN_DDS however only until the overhead becomes larger than serial localized Evolution itself, a point where CLAN_DCS could potentially prove to be a better choice. However, this is something which we have not seen till exhaustion of our test bed as execution time for Distributed Asynchronous

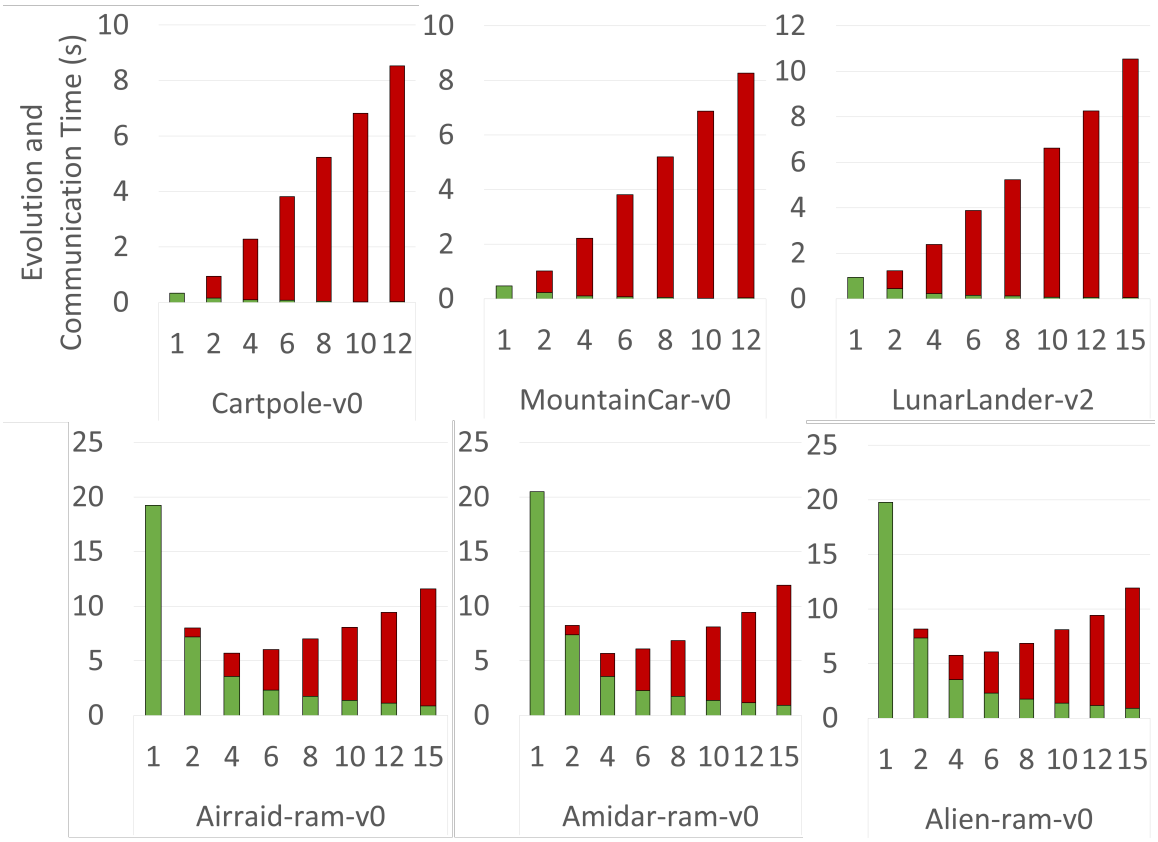

Figure 5.4: Execution Runtime at scale for Evolution and Communication using Asynchronous Speciation

Evolution continues to be lesser than that of localized Evolution. We further this line of thought in Section 5.2.2

5.1.3 Evaluating Scalability

In our evaluations so far, we have operated all workloads with each generation and consequently inference lasting multiple time steps. However, this assumption does not always hold true outside of typical RL game workloads such as using NE in autonomous robotics. So far, because of this nature, inference compute has dominated heavily and not allowed evaluations a chance to understand the difference in performance of the complete learning process under different configurations. This limitation can be overcome by evaluating each genome only once in a given generation, thus limiting the dominance of inference. In such a scenario, we can truly test the mettle of various configurations.

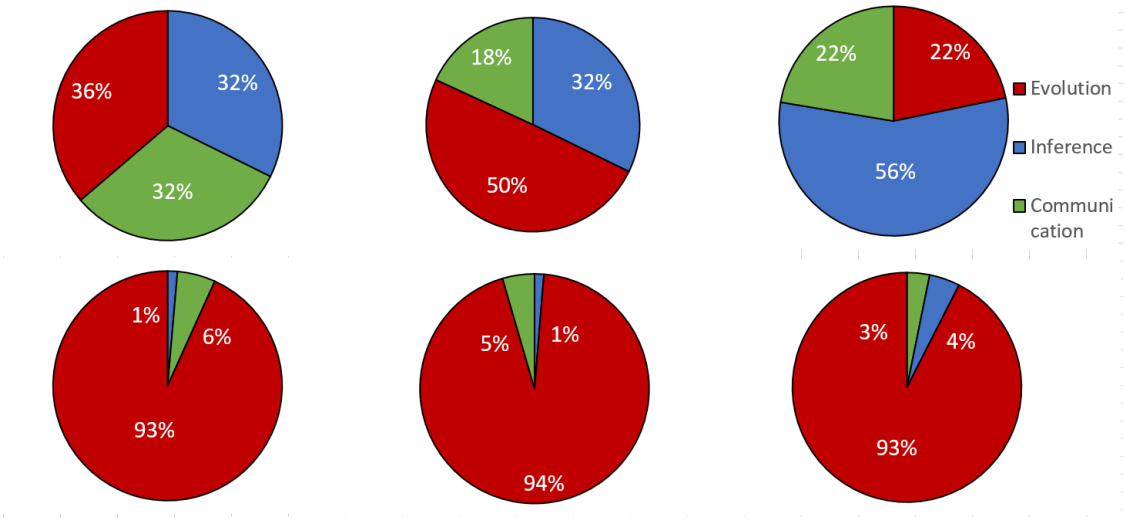

Figure 5.5: Breakdown of Compute Share in Different Designs

We use two workloads; one from each class of large and small workloads namely Airraid-ram-v0 and Cartpole-v0 respectively. The share of different compute for each of the two workloads has been shown in Figure 5.5 for comparison. The effect of the increased amount of communication in CLAN_DDS becomes more evident here. Along with the increased amount of communication involved, the constant cost of invoking the communication channels also kills such a design when such a scenario is presented where the amount of communication is no longer sufficient to amortize the constant setup costs. Looking at Figure 5.5, it can be seen that communication consumes about 50% and 94% of the share for larger and smaller workload respectively while in the CLAN_DDS configuration. This share reduces to 36% and less significantly to 93% respectively in CLAN_DCS. The best result is seen while using CLAN_DDA, where the share of communication is only 22% and 93% respectively indicating both energy and runtime savings. This result is significantly important to note as the cost of communication can get extremely high between edge devices using slower and more distant mediums of communication or go the other way by using better and faster mediums, an effect which we discuss in Section 5.2.2.

As we have seen in Section 5.1.2, Asynchronous Speciation allows reproduction to scale along with inference. Scalable reproduction and inference along with reduced cost of communication make CLAN_DDA the preferred configuration.

5.2 Discussion

5.2.1 Performance per dollar

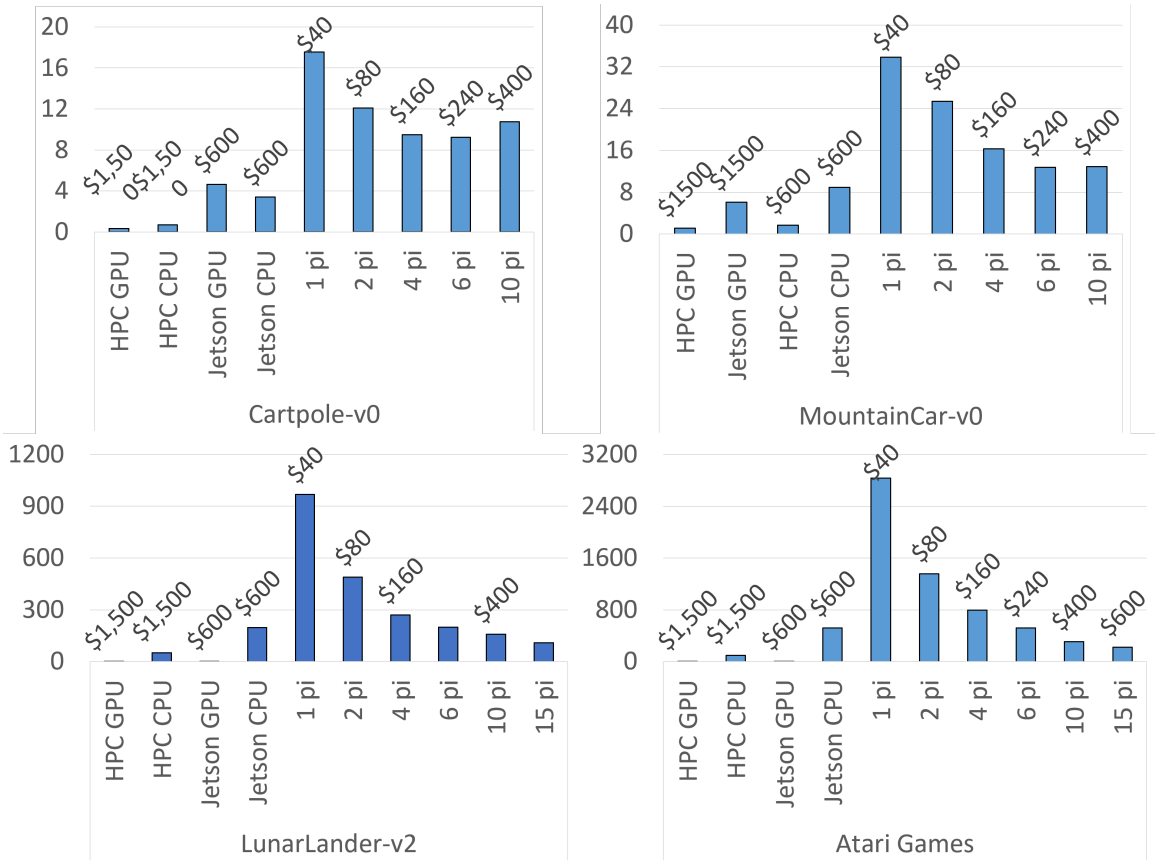

Figure 5.6: Comparing Platforms - Performance per Dollar

Deploying massive intelligence on the edge needs an additional performance metric to be evaluated, i.e. the price for performance. To evaluate this metric, we also compare the results of Inference of the distributed system with Raspberry Pis to two localized implementations, one in each of the desktop and embedded class: (a) High-Performance Machine (HPC) and (b) Nvidia Jetson Tx2 described in Table 5.1. The price of HPC machine and Jetson Tx2 is comparable to 40x and 15x to the cost of a RPi respectively. An interesting question now is whether such a distributed system at scale can achieve similar performances to these much more expensive platforms while paying the communication latency and if yes, at what scale does it achieve equivalent performance.

We examine this question in Figure 5.6 for various workloads. We see that for extremely small workloads such as *Cartpole-v0*, the distributed system cannot achieve performance equivalent to that of localized implementations on higher end platforms. This is due to the communication overhead which cannot be amortized by low amount of compute. However, for larger workloads, we see interesting results. At a scale of 6 compute units, the system can achieve performance similar to a Nvidia Jetson Tx2 board, a *Price-Performance Product (PPP)* improvement of 2.5x. At a scale of 15 units, we can compare the system with the HPC system, a achieving a PPP benefit of 1.2x. The GPU performances of both the higher end platforms could not be rivaled within the limits of our single core experiments with our test bed. Although, we haven't noticed scaling stop yet for the larger workloads.

Table 5.1: Platform Specifications

Platform	Processor	Price
HPC CPU	6th gen i7	$1500
HPC GPU	Nvidia GTX 1080	$1500
Jetson Tx2 CPU	ARM Cortex A57	$600
Jetson Tx2 GPU	Pascal	$600
Raspberry Pi CPU	ARM Cortex A53	$40

5.2.2 Extrapolating Scalability

We begin this section trying to answer a line of *What if* questions in order to understand the nature of scalability under different configurations and under various assumptions and assess how changes to our evaluation settings could potentially impact scalability. First important question that needs answering is:

What if we weren't limited by resources?

It is important in a scalability study that we assess the limits of the system. For larger workloads such as those of Atari games, we do not notice a point where scalability stops for Inference. Whereas for Evolution, despite noting an inflection point while using CLAN_DDA, we do not yet see a point where a serial choice would be better under the limitation of our test-bed.

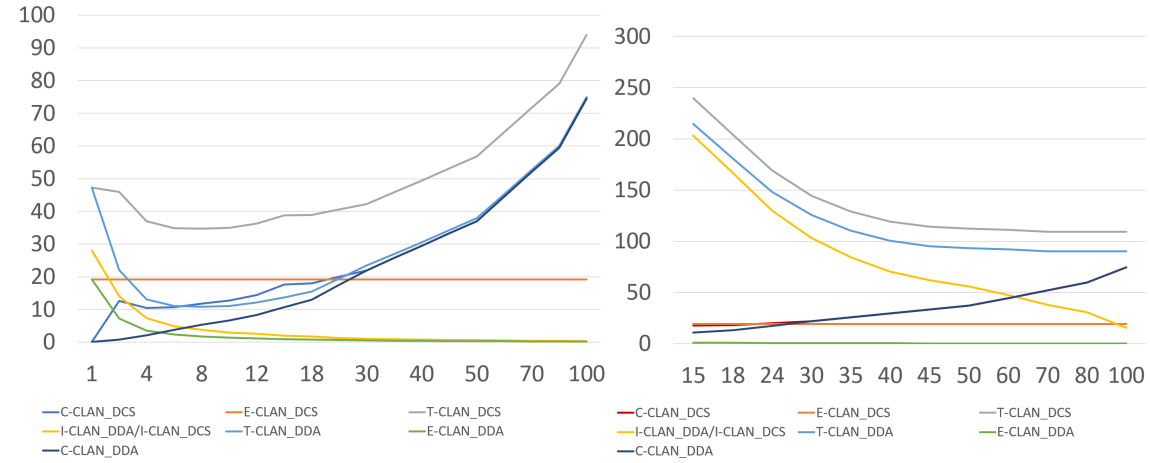

Figure 5.7: Extrapolating to assess Scalability

As Inference continues to scale benefiting the overall runtime, we wish to assess at what point does scaling stop for different configurations and as we continue to scale, what configuration proves to be a better choice. In order to do so, we need to go beyond the limit of our test-bed of 15 Raspberry Pis. It is also important to evaluate both multi-step and single-step inference as both operate under two different interesting situations. We use inference, communication overheads in each configuration and evolution trends seen to extrapolate these curves to predict these points for CLAN_DCS and CLAN_DDA. For more accurate extrapolation of compute trends, we performed experiments using different population sizes, effectively emulating higher scale as each compute agent performs computation for a fewer genomes, as would happen if there were more compute agents with the same population size. We do not study CLAN_DDS as it is clear that it performs worse in any setting. The extrapolated curves for both multi-step and single-step inference have been shown in Figure 5.7. We show raw inference, evolution compute along with communication overheads and also plot the total execution time represented by I, E, C and T in the figure.

The key result here is that for both multi and single-step inference, CLAN_DDA always performs better than CLAN_DCS in total time taken. It is also interesting to note that as raw compute continues to scale due to the available PLP, the total time taken is defined by communication overhead. For single-step inference, CLAN_DCS continues to scale till 10 units and CLAN_DDA pushes scalability to 12 units owing to its low communication

overhead. For multi-step inference, the performance stagnates for both configurations with CLAN_DDA performing better by 1.1x throughout the scale.

What if the communication technology used was better? Over the past few years, the idea of a connected edge has grown more and more. From IoT for smart devices to autonomous driving (V2V communication), the discussion of communication between devices has been prevalent in both academia and industry. Innovations made in the field such as LIDAR etc. could certainly be leveraged in the future in a setting like ours. To analyze the effect such a change would have on the scalability in our distributed configuration, we halve the communication cost as an approximation and plot the total execution time curves for CLAN_DCS and CLAN_DDA for both multi-step and single-step inference in Figure 5.8

We notice the scalability of CLAN_DCS improves from 10 nodes to 12 in such a scenario whereas it does not show any change in CLAN_DDA where communication was a less significant challenge.

What if we used Custom HW instead of Raspberry Pi? Needless to say, there has been a tremendous surge of research work in the field of custom DNN inference hardware in recent years. Many DNN accelerators have been proposed [25, 26, 27, 28, 29, 30, 31, 32, 33]. Going by the trend, it is not far fetched to imagine the availability of these accelerators as commodity hardware. We assume a 32x32 systolic array implementation and evaluate performance using a cycle-accurate simulator SCALE-sim [41] and plot again the total runtimes for multi-step inference for both configurations under this assumption in Figure 5.8. Signicantly faster compute performance means communication becomes a much more serious issue and that is exactly what we can witness from the runtime plots. CLAN_DCS cannot scale under such a situation due to the heavy communication overhead and serial evolution. CLAN_DDA however, still shows scalability scaling upto 6 nodes showing a performance improvement of over 2.5x when compared to CLAN_DCS.

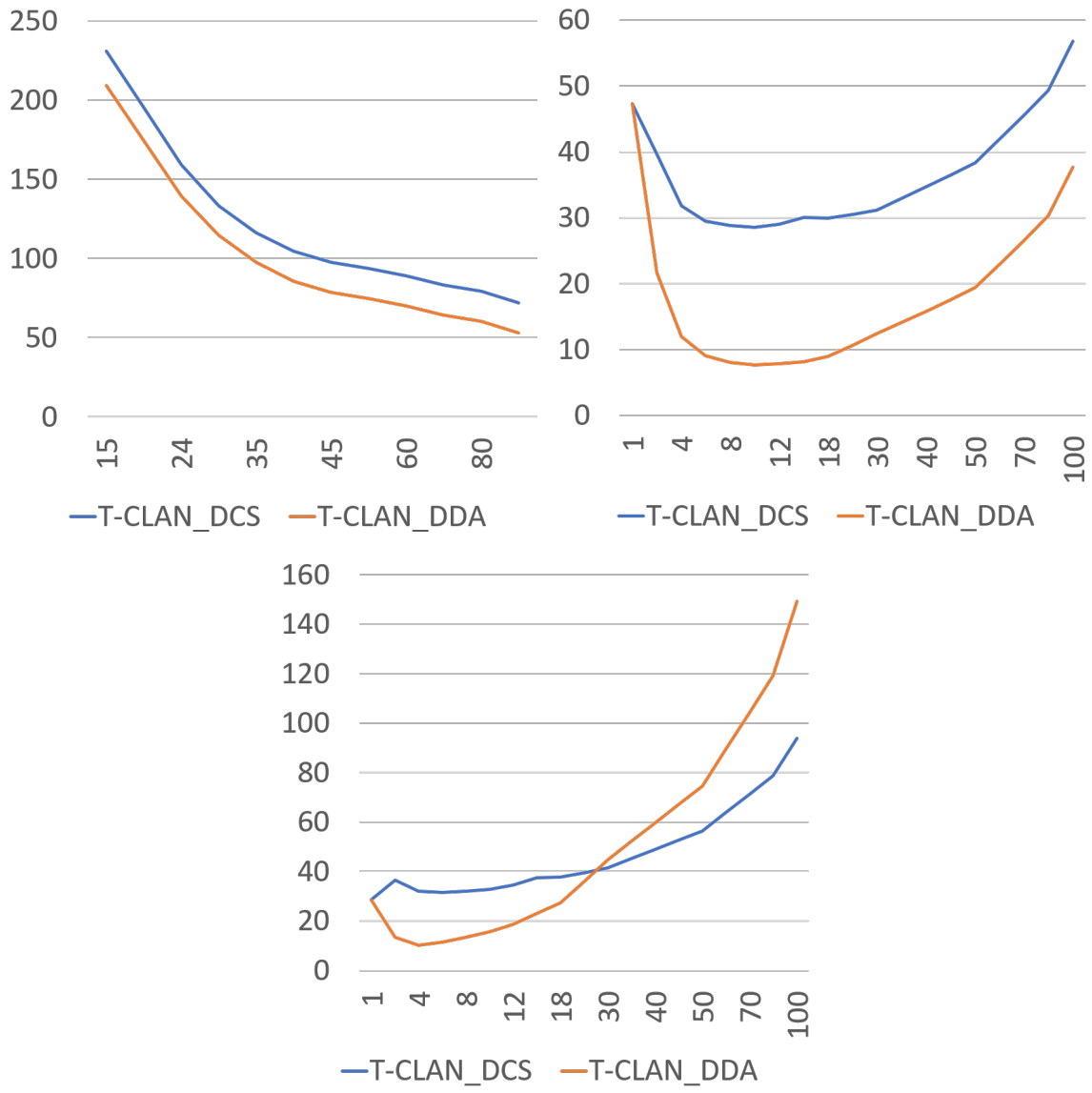

Figure 5.8: Extrapolating to assess Scalability with alternate computing and communication paradigms

CHAPTER 6
CONCLUSION AND FUTURE WORK

Significant focus and effort has been made in characterizing, analyzing and understand the compute behavior of Deep Learning workloads and designing custom accelerators for the same. However, the DL landscape today cannot bring true general intelligence to the edge due to heavy training requirements and rigid topologies. Evolutionary algorithms, particularly the field of NeuroEvolution brings the promise of solving these problems as they do not require backpropagation based training and do not suffer from rigidity but instead can evolve continuously.

Even with increasing interest in NE, we have not seen NE solutions being widely accepted due to the lack of understanding of their compute behavior and suitable platforms. This research work tries to bridge the gap by characterizing and understanding opportunities and challenges in enabling NE on various heterogeneous platforms. This work also argues for bringing intelligence to the edge as it can significantly alter our experiences with AI as a human race.

To enable such a solution, this work studies in detail the compute and communication behavior of NEs and its performance within the constraints of the edge. This work also motivates the need for a distributed collaborative learning solution the edge problem and studies two different scaling configurations. Finally, it also proposes an algorithmic modification *Asynchronous Speciation* and a distributed solution *CLAN - Collaborative Learning using Asynchronous NeuroEvolution* that can help scale the algorithm further and improve performance by significantly cutting the synchronization cost paid by the system.

6.1 Future Work

6.1.1 Allowing Global Speciation

Allowing complete Asynchronous Speciation (AS) means that at the very least there will be as many species as the number of computing units in the distributed system. This can eventually get out of hand as the number of units continue to scale and the number of species continue to increase. Intuitively increased number of species mean lesser genomes in each species and subsequently lesser competition for survival as each member only competes within its own species. It could also mean slower exploration as more of similar genomes survive. Both of these factors could potentially hamper the learning and diminish the convergence rate for a given task. We aim to study this effect in more detail in the future.

Given that such a situation exists, we need to come up with new techniques that allows synchronous speciation without a significant bump in the communication costs. One straightforward way of achieving such a result is to allow synchronous speciation periodically each n generations. This period can be tweaked based on the accuracy loss incurred due to asynchronicity.

An alternative to *Periodic Synchronous Speciation (PSS)* can be imagined as *Synchronous Speciation using Representative Members(SSR)*. SSR(PSR) performs speciation every(n) generations using a representative member of the species formed by AS. Two representatives belonging to the same species automatically clubs their entire species into one. This form of synchronous speciation reduces cost, however it is important to note that it does not reduce the amount of synchronization points needed to scale evolution and thus still suffers on that front.

These are interesting ideas that need to be explored in the future.

6.1.2 Asynchronous Generation Planning

In AS, the only synchronization step that leaves units idle is the calculation and communication of spawn count which allows reproduction to begin once received. The above means that if we are able to somehow generate children without receiving an exact spawn count, we can eliminate significant portion of the idle time on the agents. To achieve this, we propose *Asynchronous Generation Planning (AGP)*. This translates to removing the requirement of maintaining an exact population size. The need for generation planning arises from the need to allow fitter species to form more children to help move the exploration in the direction of the optimal solution to the task. If each species could individually gauge its fitness w.r.t to other species, it can estimate the number of children it has to form. A mechanism whereby this is achievable as an approximation can solve this problem. In such a scheme, each species decides its spawn count by looking at its current percentage of the population size, i.e. size of species/size of population, its own fitness history and its history of spawn counts defined by the central agent. The idea is to see if the fitness of the species has been growing in the past and correspondingly if the center has historically assigned more and more members to the species or vice-versa. If the former is the case, then there is an estimated increase in spawn count and can be estimated using the current species size and a model of the history. If latter, the decrease is estimated. It is immediately obvious that such a model will be based on assumption and can make mistakes. To mitigate these mistakes, one can add completely asynchronous updates to the model where correct spawn counts are sent to the agents by the center. This communication is completely overlapped with computation as agents never wait for this information. Once received, agents look at the spawn count and compare against assumed spawn count. If assumption was lower, more children are added to the species in the next evolution step and if the assumption is higher, few children are killed off to maintain the balance of the population size.

6.1.3 Reliability and Fault Tolerance

In the mentioned design choices, it is easy to see the key role the central agent plays to the learning process. Throughout the generation and across them, multiple one-to-many and many-to-one communications happen where one is always the same agent. Having such an agent means that this agent can get overworked. In our experiments, we noticed similar issues with the slowing down of the central agent as we continued to run multitudes of experiments and eventually even causing failures in rare occurrences. For a reliable solution, we can switch the agent performing the task of the central agent every few generations to ensure there is no load imbalance on average across the distributed computing units. Fault-tolerance is simple to achieve in case of failure of any peripheral agents as such members of the population on that node can be considered as dead and more members can be assigned to other units in the next generation. Achieving fault-tolerance for the central node is a non-trivial matter and something that needs further exploration in the future along with achieving reliability.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.," in *AAAI*, Phoenix, AZ, vol. 2, 2016, p. 5.

[3] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[5] *Open ai five*, `https://openai.com/five/`, 2017.

[6] *Alphago*, `https://deepmind.com/research/alphago`, 2017.

[7] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[8] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[9] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, Association for Computational Linguistics, 2010, pp. 456–464.

[10] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011, pp. 693–701.

[11] H. Su and H. Chen, "Experiments on parallel training of deep neural network using model averaging," *arXiv preprint arXiv:1507.01239*, 2015.

[12] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: Near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2592–2600.

[13] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[14] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[15] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, IEEE, 2018, pp. 620–629.

[16] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, IEEE, 2017, pp. 328–339.

[17] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge, "A hybrid approach to offloading mobile image classification," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, IEEE, 2014, pp. 8375–8379.

[18] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 615–629, 2017.

[19] Z. Jiang, T. Chen, and M. Li, "Efficient deep learning inference on edge devices," 2018.

[20] X. Xu, Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi, "Scaling for edge inference of deep neural networks," *Nature Electronics*, vol. 1, no. 4, p. 216, 2018.

[21] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2017, pp. 1396–1401.

[22] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3709–3716, 2018.

[23] S. R. Young, D. C. Rose, T. P. Karnowski, S.-H. Lim, and R. M. Patton, "Optimizing deep learning hyper-parameters through an evolutionary algorithm," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, ACM, 2015, p. 4.

[24] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[25] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, IEEE, 2017, pp. 1–12.

[26] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014, pp. 269–284.

[27] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *MICRO*, IEEE Computer Society, 2014, pp. 609–622.

[28] Z. Du *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 43, 2015, pp. 92–104.

[29] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015, pp. 161–170.

[30] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[31] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, ACM, 2017, pp. 27–40.

[32] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, IEEE, 2016, pp. 243–254.

[33] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *SIGPLAN Not.*, vol. 53, no. 2, pp. 461–475, Mar. 2018.

[34] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G.-Y. Wei, "14.3 a 28nm soc with a 1.2 ghz 568nj/prediction sparse deep-neural-network engine with¿

0.1 timing error rate tolerance for iot applications," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, 2017, pp. 242–243.

[35] A. Samajdar, P. Mannan, K. Garg, and T. Krishna, "Genesys: Enabling continuous learning through neural network evolution in hardware," *arXiv preprint arXiv:1808.01363*, 2018.

[36] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[37] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[38] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[39] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, p. 201 611 835, 2017.

[40] *Neat python*, `https://github.com/CodeReclaimers/neat-python`, 2017.

[41] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator," *arXiv preprint arXiv:1811.02883*, 2018.