VLADIMIR PLES

Implementation of a model design framework in a GPU Server Platform

2023

Vladimir Ples

Implementation of a model design framework in a GPU Server Platform

MSc. in Computer Engineering

Supervisor:
Prof. Dr. António Ruano

Statement of Originality

Title

Declaration of authorship of work: I hereby declare to be the author of this original and unique work. Authors and references in use are properly cited in the text and are all listed in the included reference section.

Candidate:

____

(Vladimir Ples)

Copyright ©Vladimir Ples

The University of Algarve has the right, perpetual and without geographical boundaries, to archive and make public this work through printed copies reproduced in paper or digital form, or by any other means known or to be invented, to broadcast it through scientific repositories and allow its copy and distribution with educational or research purposes, noncommercial purposes, provided that credit is given to the author and Publisher.

Acknowledgements

Firstly, I would like to thank Dr. António Ruano for his continued support and guidance throughout the thesis process. His insight and expertise were invaluable in shaping this thesis, and I am grateful for the opportunities he provided me to grow as a researcher.

I would like to acknowledge the financial support of Operational Program Portugal 2020 and Operational Program CRESC Algarve 2020, grant number 72581/2020, “Boosting energy communities: from home energy management systems to intelligent energy aggregators

I would like to thank the ”Serviços de Informática da Universidade do Algarve” department, and in particular, Nuno Costa for the help and insights into setting up the various virtual machines and servers on the university’s infrastructure.

I would like to express my gratitude to every one of my laboratory colleagues for their support and the stimulating work environment they provided me.

Lastly, I want to thank my family and friends for their emotional support and encouragement to both start and finish this Master’s program.

Abstract

This thesis deals with the development and implementation of a model design framework in a Graphical Processing Unit (GPU) Server Platform powered by a multi-objective genetic algorithm with some novel components. With GPUs suffering rapid improvements in performance, functionality and energy efficiency they have become an attractive choice in contexts where it is necessary to process large amounts of data. The objective behind this work is to make use of the processing capabilities of the GPU server platform to accelerate the process of model design by parallelizing the workload in the GPU Server Platform and making use of its specific capabilities to provably outperform an existing legacy platform using a cluster of machines that rely mostly on Central Processing Units (CPUs).

This thesis will first provide a background on the GPU and the Multi-Objective Genetic Algorithm that will be used, as well as the Legacy Platform that will be employed as a basis. The following section will provide an analysis of the state of the art regarding constructive algorithms and the general paradigm in which GPUs are used. The next sections details the initial contact with the legacy model design framework and the documentation efforts; the creation of a methodology to extract processing times and memory values; the prototyping of the GPU implementation along with the challenges presented, and the solutions found. The final parts of the thesis address the statistical validation of the newly produced models as well as a comparison of the performance between the legacy platform and the new one.

Keywords: Model Design, Neural Networks, Multi-Objective Genetic Algorithm, GPU Acceleration

Resumo

Esta tese lida com o desenvolvimento e implementação de uma ‘framework’ de desenho de modelos de inteligência artificial numa plataforma que tem ao seu dispor uma Unidade de Processamento Gráfico (GPU). Esta plataforma implementa um algoritmo genético multi-objetivo com componentes inovadoras.

Com as melhorias rápidas no desempenho, funcionalidade e eficiência energética, as GPUs vieram a tornar-se numa escolha atractiva em contextos onde é necessário processar grandes quantidades de dados. O principal objetivo deste trabalho é aproveitar as capacidades de processamento da GPU existente na plataforma previamente mencionada. Ao utilizar a GPU é esperado que exista uma aceleração do processo de desenho de modelos.

Esta plataforma encontra-se me uso há alguns anos, tendo sido utilizada por vários investigadores mas não tendo sido ativamente mantida nos anos mais recentes. Antes do trabalho neste projeto se iniciar, foi decidido usar a plataforma original como base e não fazer alterações radicais a não ser que fossem necessárias, isto de forma a manter o foco no desenvolvimento e implementação de código que utilizasse a GPU.

Os primeiros passos tomados foram a adaptação ao sistema e a documentação do mesmo, visto não existir documentação relevante. Procedeu-se então com a exploração do código e documentação do mesmo dentro que fora compreendido. Um conjunto de diagramas que detalham as principais interações do sistema foram também produzidos como uma forma de documentação. Após este processo foi necessário atualizar as várias componentes da plataforma de forma criar uma base atualizada a partir da qual se pudesse partir começar a trabalhar livremente. Este processo de atualização foi realizado numa par de máquinas virtuais de teste cujo propósito era agirem como uma versão condensada da plataforma original, o que permitiu testar de forma mais completa a versão atualizada do código.

Com o par de máquinas de teste a utilizar a versão atualizada do código, o processo de criar a implementação em GPU poderia começar. Foram colocados em vários pontos do código marcadores que iriam temporizar várias funções relevantes ao treino dos modelos. Um processo semelhante foi feito também, mas com o intuito de gravar a quantidade de memória utilizada pelos modelos ao longo do processo. Os dados resultantes foram compostos em ‘datasets’ que

iriam ser utilizados seguidamente com dois objetivos: compreender que partes do código poderiam ser alteradas para serem mais rápidas, com recurso à GPU, e tentar fazer previsões dos valores de tempo e memória com base na configuração inserida. A análise do tempo que as operações levam durante a execução levou à conclusão que o passo relativo à previsão é o que mais tempo consome, seguido do algoritmo de Levenberg Marquardt. Estas operações seriam as principais candidatas a serem aceleradas via GPU. A previsão dos valores de tempo e de memória seria processada por um algoritmo chamado ASMOD, usado para modelar empiricamente dados observáveis com recurso a funções 'basis spline'. Os resultados obtidos seriam dececionates, pelo que seriam procuradas alternativas para a estimação dos valores de tempo e memória.

Após encontrar as partes que mais iriam beneficiar do use de GPU, seria necessário criar um protótipo, alguns testes seriam efetuados de forma a validar a escolha da biblioteca que seria utilizada para a implementação em GPU com resultados positivos. A implementação consistiu na criação de uma função de cálculo da jacobiana que efetua os cálculos em blocos, ao invés de sequencialmente, o que iria acelerar a função substancialmente, outra funções foram alteradas para as suas funções equivalentes na biblioteca 'CuPy', utilizada para acelerar o código. Foi tentada a aceleração de uma função que trata da ativação sem sucesso, sendo esta utilizada na sua forma original. Após implementar as alterações, testes foram realizados sem sucesso. Notou-se que a GPU relatava que os processos lançados para o treino de modelos estavam a consumir mais memória que a GPU tinha, o que levou à criação de um mecanismo que tentaria treinar um modelo, enquanto grava o valore máximo de memória utilizado. Após obter o valor máximo, seria feito um cálculo de forma a descobrir quantos processos poderiam ser lançados sem passar dos limites da memória da GPU. Este mecanismo age como uma solução ao problema de previsão de memória mencionado previamente. Um último passo foi necessário antes da nova implementação funcionar corretamente. A ativação de uma funcionalidade da GPU chamada 'Multi-Process Service' (MPS) que permite que a GPU interaja com múltiplos processos facilmente.

Com uma implementação funcional, o passo seguinte seria a validação estatística da mesma. Isto permitiria entender se os novos modelos gerados estariam ao nível dos anteriores. Como tal, o mesmo problema foi executado na plataforma antiga e na nova, sendo extraídos os modelos resultantes e respetivos erros. Após uma análise estatística auxiliada por gráficos, concluiu-se que os modelos eram de qualidades semelhantes apesar de existirem algumas discrepâncias em termos estatísticos, os quais foram atribuídos à natureza do algoritmo.

O último passo consistiu na execução de múltiplos problemas de ambos os tipos na plataforma antiga e na nova. Os tempos tendo sido gravados com recurso a marcadores, de forma semelhante a testes anteriores, tendo gravado o tempo de início e fim do problema, assim como o tempo que cada modelo demorou a ser treinado. Os principais resultados a comparar foram os tempos totais, que indicam que para ambos os tipos de problemas. No geral pode concluir-se

que a implementação em GPU na nova plataforma pode ser considerada um sucesso.

Contents

Acknowledgements . . . iii
Abstract . . . iv
Resumo . . . v
List of Figures . . . xi
List of Tables . . . xii

Nomenclature **xiii**

1 Introduction **1**
1.1 Introduction . . . 2
1.2 Scope . . . 2
1.3 Research Objective and Expected Contribution . . . 3

2 Background **4**
2.1 Graphics Processing Units . . . 5
2.2 Multi-Objective Genetic Algorithm . . . 7
2.2.1 Individual Representation . . . 7
2.2.2 MOGA Algorithm . . . 8
2.2.3 Radial Basis Function Artificial Neural Network . . . 10
2.2.4 Levenberg Marquardt Algorithm . . . 10
2.3 The Legacy MOGA Platform . . . 13
2.4 MOGA Platform applications . . . 15
2.5 ASMOD . . . 16

3 Literature Review **18**
3.1 Constructive Algorithms . . . 19
3.2 GPU paradigm . . . 21

4 Methodology **24**
4.1 Documentation . . . 25
4.2 Hardware . . . 29
4.3 Legacy Platform Updating . . . 29

4.3.1 Development Platform . . . 32
4.4 Performance Data Collection . . . 33
4.4.1 Methodology . . . 34
4.4.2 Results . . . 34
4.4.3 ASMOD . . . 38
4.5 GPU Implementation . . . 39
4.5.1 Library Choice . . . 40
4.5.2 Prototyping . . . 40
4.5.3 Implementation . . . 41
4.5.4 Process Load Balancing . . . 42
4.5.5 Mini MOGA . . . 44

5 Evaluation **47**
5.1 Validation of Results . . . 48

6 Results - Execution time **56**

7 Discussion and Conclusion **73**

List of Figures

2.1 High-level view of A100 GPU, built on the Ampere architecture. (Source: Nvidia) 5
2.2 A100 GPU Streaming Multiprocessor Detail. (Source: Nvidia) 6
2.3 Chromosome and input space lookup table [1] 7
2.4 Crossover Recombination operator [1] 9
2.5 Typical flow of operation in a MOGA [1] 9
2.6 Overview of the legacy MOGA platform 13

3.1 The Cascade architecture, after two hidden units have been added. The vertical lines sum all incoming activation. Boxed connections are frozen, X connections are trained repeatedly. Source:[2] 20

4.1 Flowchart Legend 26
4.2 Flowchart example 1 27
4.3 Flowchart example 2 28
4.4 Code acceleration on sequential CPU code 34
4.5 All models training time, net_id x time (sec) 35
4.6 250 models sample training time with outliers, net_id x time (sec) 35
4.7 Classification problem training time (sec) 36
4.8 Classification problem training time (%) 36
4.9 Regression problem training time (sec) 37
4.10 Regression problem training time (sec) 37
4.11 ASMOD input data example for regression problems (memory) 38
4.12 ASMOD input data example for classification problems (time) 39
4.13 Mini Moga GUI use - CPU 45
4.14 Mini Moga GUI results - CPU 45
4.15 Mini Moga GUI use - GPU 46
4.16 Mini Moga GUI results - GPU 46

5.1 Overview of errors (CPU) 48
5.2 Overview of errors (GPU) 49
5.3 Forecast Error (CPU) 49

5.4 Forecast Error (GPU) 50
5.5 CPU vs GPU median error per step PH:48 50
5.6 Training Error (CPU) 51
5.7 Training Error (GPU) 51
5.8 Testing Error (CPU) 52
5.9 Testing Error (GPU) 52
5.10 Validation Error (CPU) 53
5.11 Validation Error (GPU) 53

6.1 dbc1 (CPU) - Model Execution Time 58
6.2 dbc1 (GPU) - Model Execution Time 58
6.3 dbc1 (CPU) - Number of models by execution time 59
6.4 dbc1 (GPU) - Number of models by execution time 59
6.5 fd8 (CPU) - Model Execution Time 60
6.6 fd8 (GPU) - Model Execution Time 60
6.7 fd8 (CPU) - Number of models by execution time 61
6.8 fd8 (GPU) - Number of models by execution time 61
6.9 sc17 (CPU) - Model Execution Time 62
6.10 sc17 (GPU) - Model Execution Time 63
6.11 sc17 (CPU) - Number of models by execution time 63
6.12 sc17 (GPU) - Number of models by execution time 64
6.13 atgambelas (CPU) - Model Execution Time 65
6.14 atgambelas (GPU) - Model Execution Time 65
6.15 atgambelas (CPU) - Number of models by execution time 66
6.16 atgambelas (GPU) - Number of models by execution time 66
6.17 kbotp9 (CPU) - Model Execution Time 67
6.18 kbotp9 (GPU) - Model Execution Time 68
6.19 kbotp9 (CPU) - Number of models by execution time 68
6.20 kbotp9 (GPU) - Number of models by execution time 69
6.21 p1pcsr (CPU) - Model Execution Time 70
6.22 p1pcsr (GPU) - Model Execution Time 70
6.23 p1pcsr (CPU) - Number of models by execution time 71
6.24 p1pcsr (GPU) - Number of models by execution time 71

List of Tables

4.1 Least Squares (LSTSQ) operation with different libraries on GPU Server 40
4.2 Matrix Multiplication (MatMult) operation with different libraries on GPU Server 41

5.1 Median values CPU vs GPU 54

6.1 All problem configurations 57
6.2 All problem times CPU & GPU 57
6.3 dbc1 problem configuration 58
6.4 fd8 problem configuration 60
6.5 sc17 problem configuration 62
6.6 atgambelas problem configuration 64
6.7 kbotp9 problem configuration 67
6.8 p1pcsr problem configuration 69

Nomenclature

ANN	Artificial Neural Network
ANN-PSO	Artificial Neural Network-Particle Swarm Optimization
API	Application Programming Interface
APT	Advanced Package Tool
ASMOD	Adaptive Spline Modelling of Observational Data
B-Spline	Basis Spline
CNNE	Constructive Neural Network Ensemble
CPU	Central Processing Unit
CT	Computed Tomographic
CUDA	Compute Unified Device Architecture
CVA	Cerebral Vascular Accidents
GNU Compiler Collection	GCC
GPU	Graphics Processing Unit
GUI	Graphical User Interace
HVAC	Heating Ventilation and Air Conditioning
I/O	Input Output
IMBPC	Intelligent Model Based Predictive Control
IP	Internet Protocol
LM	Levenberg Marquardt
MOGA	Multi-Objective Genetic Algorithm
MOGWO	Multi-Objective Grey Wolves Optimization
MPS	Multi-Process Service

N/A	Non-Applicable
NAR	Non-linear Autoregressive
NARX	Non-linear Autoregressive with Exogenous inputs
NFS	Network File System
OS	Operating System
PH	Prediction Horizon
PIP	Package Installer for Python
RBFANN	Radial Basis Function Artificial Neural Network
REN	*Rede Eletrica Nacional*
RMSE	Root Mean Square Error
SM	Streaming Multiprocessors
VPN	Virtual Private Network

Chapter 1

Introduction

1.1 Introduction

Graphics Processing Unit (GPU) technology has come a long way since its early days powering video game graphics. Today, it powers the *Frontier* supercomputer, managed by the Oak Ridge National Laboratory. *Frontier* features 74 nodes, containing more than 9400 3rd Gen AMD EPYC™ CPUs and 37000 purpose built AMD Instinct 250X™ GPUs connected by *Slingshot*, a high-performance Ethernet fabric designed for HPC and AI solutions. This system can achieve a theoretical performance of 1.6 Exaflops[1], being the first supercomputer to pass the exascale barrier.[3][4]

In 2001 one of the first attempts of non-graphical computations on a GPU was a matrix-matrix multiply and in 2005 a GPU implementation outperformed a Central Processing Unit (CPU) in solving Dense Linear Systems [5][6][7]. Hardware advancements and improvementes in the programmability, such as NVIDIA's Compute Unified Device Architecture (CUDA), a parallel computing platform and programming model for general computing on graphics processing units (GPUs). This facilitated development with the introduction of threads, vector processing, data chaches and shared memory, replacing old concepts specific to the development of graphics applications like pixels, buffers and fragments, paving the way for more accesible use of GPUs for acceleration.

What resulted as a necessity to accelerate the display of 2 dimensional (2D) and 3 dimensional (3D) graphical applications for entertainment purposes in the past, has evolved into a critical component for accelerating complex computations in modern day applications such as deep learning, artificial intelligence and scientific simulations. Although CPUs are used in these areas, GPUs can outperform them given an appropriate problem and implementation. CPUs possess higher clock speeds but relatively few cores compared to GPUs, whose strength comes from its high parallelism, as they possess a high number of cores, usually in the order of thousands. While a CPU may perform a task faster, its throughput will be lower than a GPU's if the task can be performed in parallel by its high number of cores.

1.2 Scope

This work is integrated in the research project 72581-HEMS2IEA/SAICT/2020, Boosting energy communities: from home energy management systems to intelligent energy aggregators, sponsored by CRESC Algarve 2020.

[1] $1.6 * 10^{18}$ floating operations per second.

1.3 Research Objective and Expected Contribution

A framework, written in C, Python, Matlab and PostgreSQL for designing neural network models exists in the university's platform, running on a cluster of virtual machines. This platform deals with large datasets and must train multiple Neural Networks (NNs). As a result, it takes large amounts of time to design all the models. Currently, it uses Radial Basis Function Artificial Neural Network (RBFANN) models which are trained and fed into the Multi-Objective Genetic Algorithm (MOGA).

The objective of this project is to implement this framework in a server that will use GPUs to accelerate the process of model design where possible. The first step will be to get a full grasp of the system and its functions, this including going over limited existing documentation, analyzing the code and creating relevant documentation and diagrams to outline the main use cases. Furthermore, it is necessary to update legacy codebase to up-to-date versions of the used libraries, the one that stands out the most is Python 2, which was sunset on January 1st, 2020 and is not receiving any more updates. Afterwards, an analysis will be performed to discern which parts of the code are possible to accelerate, and lastly the implementation of the new framework on the GPU server.

The new framework is expected to be faster, more modern and reliable than the existing legacy framework. With the use of GPUs to accelerate the design of models it is expected to substantially reduce the time researchers need to wait for their results. This coupled with a more complete documentation would assist future developers in further extending the functionality of the platform.

From an academic standpoint, the area of general purpose computing using GPUs has already been explored to some extent. Despite this, research has been limited regarding the application of GPUs to accelerate this specific type of framework (MOGA). It is therefore expected that this work will provide some insights for development of this type of framework.

Chapter 2

Background

2.1 Graphics Processing Units

Graphics Processing Units were initially developed to handle complex graphical tasks in graphical applications. However, as their power and capabilities increased over time GPUs began to be used for high performance computation as they were well suited to handle large amounts of data.

In essence, a GPU is a powerful programmable processor designed for a specific class of applications with some of the following characteristics [8]:

- Large computational requirements. GPUs must deliver an enormous amount of computation performance to satisfy the demand of complex real-time applications.
- Substantial Parallelism. Operations on vertices and fragments are well-matched to fine-grained closely coupled programmable parallel compute units, which in turn are applicable to many other computational domains.
- Throughput is more important than latency. GPU implementations of the graphics pipeline prioritize throughput over latency.

Figure 2.1: High-level view of A100 GPU, built on the Ampere architecture. (Source: Nvidia)

Figure 2.2: A100 GPU Streaming Multiprocessor Detail. (Source: Nvidia)

The NVIDIA GPU architecture, which we will be focusing on, was built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors [7]. For an algorithm to execute efficiently on the GPU it must be cast into a *data-parallel* form with many thousands of independent threads of execution running the same lines of code, but on different data [9]. As most of the execution is done asynchronously, it is necessary that threads do not rely on results from other threads as this can lead to inconsistent results.

2.2 Multi-Objective Genetic Algorithm

A multi-objective genetic algorithm (MOGA) is an optimization technique that benefits from a set of procedures and operators inspired on the process of natural evolution and on the notion of survival of the fittest, in order to perform a population based search for the Pareto set of solutions of a given multi-objective problem [1]. The next subsections will go over the details of the MOGA used throughout the project.

2.2.1 Individual Representation

Each individual in the MOGA has its representation, the *chromosome*, each of these is an encoding for the topology of a Radial Basis Function Artificial Neural Network (RBFANN). It is possible for other types of *feed-forward* Artificial Neural Networks (ANN) to be used with some changes to the representation and implementation.

The topology of the RBFANN mentioned must contain the number of neurons used and input features that will be employed. These will correspond to a string of integers, where the first one represents the number of neurons used and the remaining represent a subset of input terms taken from the total feature space of our data. Figure 2.3 shows a representation of the chromosome and the table used to represent the input terms.

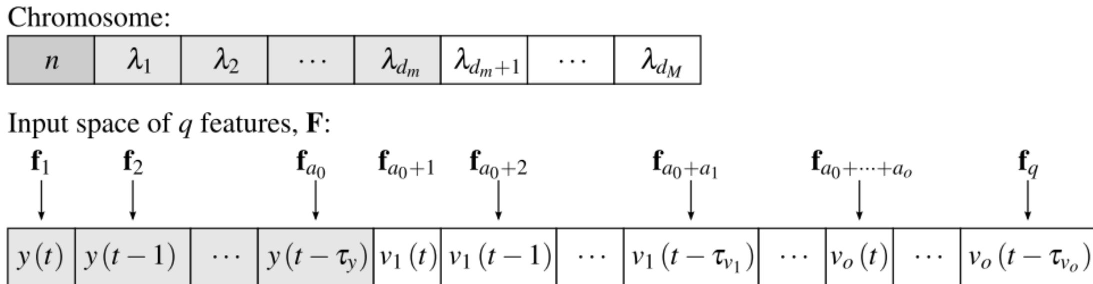


Figure 2.3: Chromosome and input space lookup table [1]

The first component corresponds to the number of neurons, those highlighted by a light gray background represent the minimum number of inputs, and the remaining are a variable number of input terms up to (in total) d_M. The λ_j values are the indices of the features f_l (the columns of **F**), the input features.

In situations where the ANN is used for predicting, its input-output structure can be formulated as a non-linear autoregressive (NAR) with exogenous inputs (NARX). In these situations, **F** is composed of a_0 output delayed terms with a maximum lag of τ_y and a_i input terms for each exogenous variable v_i, each having τ_{vi} as maximum lag. The inputs corresponding to delayed output values are highlighted in a light gray background in the lower part of figure 2.3 [1].

$$\begin{aligned} y(t+1) =& g(y(t), y(t-1), \ldots, y(t-\tau_y), \\ & v_i(t), v_i(t-1), \ldots, v_i(t-\tau_{vi}), \\ & \ldots, \\ & v_0(t), v_0(t-1), \ldots, v_0(t-\tau_{v0})) \end{aligned} \tag{2.1}$$

Equation 2.1 describes how a 1-step ahead prediction ($y(t+1)$) would be calculated using the general mapping (g) of the delayed output values (y) and exogenous variables (v_i). As a NARX model, the predicted $y(t+1)$ value would be τ_{v0}. The values used for this calculation will be $y(t), y(t-1), \ldots, y(t-\tau_y)$. At the next time step, the arguments of the model inputs are increased by one sample, creating this was a sliding window approach.

2.2.2 MOGA Algorithm

The MOGA algorithm starts with the generation of a set of individuals, called the initial population. Each individual will have a random number of neurons and a random set of inputs from the input features, each within a range set beforehand. The population is designated as a collection of individuals and each iteration of the algorithm is called a generation, where each generation has its associated population.

After having generated the initial population, the individuals will be subjected to an iterative cycle composed of evaluation, selection, recombination and mutation. The evaluation step consists of evaluating the individual, a procedure which assigns to each individual a fitness value describing how well the solution performs in the multiple objectives. A stopping criteria can be implemented to make the algorithm stop early if the solutions fulfill set criteria. This however does not occur in the MOGA algorithm used. Instead, a maximum number of iterations is set manually as a parameter. The user also has the option to end the algorithm manually at any time.

The selection step involves selecting a number of individuals from a population based on their fitness values. The selected individuals will be used to generate the next population. Roulette wheel selection is used in this implementation. This method of selection attributes a percentage chance to each individual based on their fitness value, where the higher the fitness value, the more likely they are to be chosen. A value is then generated, and the corresponding individual is selected. This simulates the behavior of spinning a roulette wheel, where the wheel's sectors correspond to the percentages of the individuals. All the selected individuals will be used to create a mating pool for the recombination step.

Each pair of individuals in the mating pool will generate two offspring, based on a manually set *crossover probability*. Figure 2.4 illustrates the process of crossover. The chromosome is

equivalent to the individual's representation within the algorithm. Firstly, the inputs part of the chromosomes of both parents are reorganized. This is done by swapping the common terms between them to the leftmost positions, the remaining terms are then shuffled. A *crossover point* is then selected and the elements to its right are exchanged. This procedure is known as *full identity preserving crossover*, this guarantees offspring with no duplicate terms. [1]

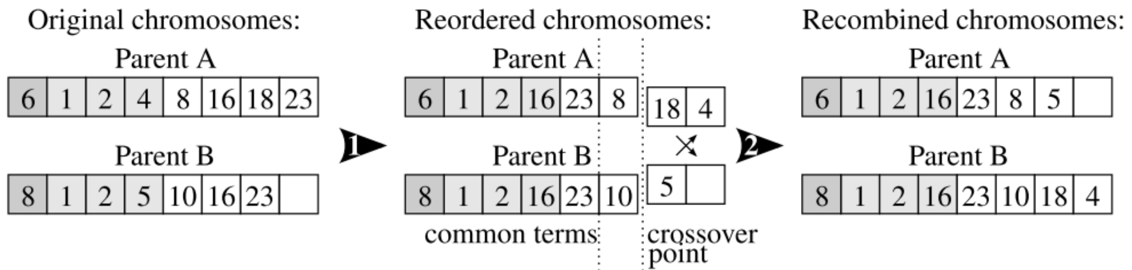


Figure 2.4: Crossover Recombination operator [1]

The mutation operator is then applied to the new population after recombination. The number of neurons in the hidden layer of the ANN can have one neuron added or subtracted to its value with a set probability, within the range specified for the number of neurons. The input terms can suffer on of three operations: replacement, addition or deletion. With a given probability, each term is tested and can be deleted or replaced by another term from the set of input features that do not exist in the chromosome. Deletion can only occur if the chromosome has more terms than the minimum number of terms. After these operations, if the chromosome has not reached the maximum number of terms, one term may be added, from the set of those outside the chromosome.

After all the operations are completed the algorithm proceeds to the evaluation step and the cycle repeats itself for the new population of individuals.

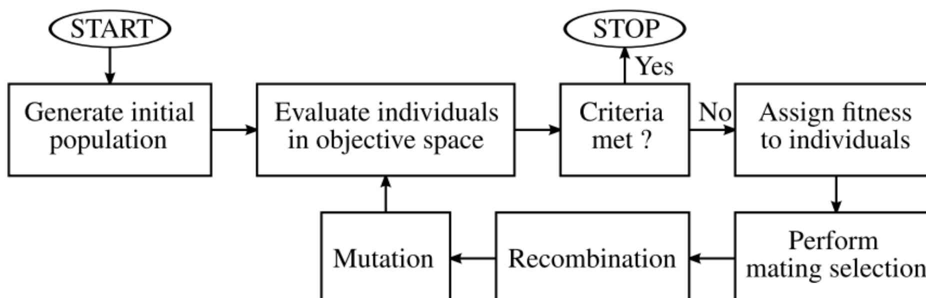


Figure 2.5: Typical flow of operation in a MOGA [1]

2.2.3 Radial Basis Function Artificial Neural Network

Every individual's representation corresponds to an ANN's topology in MOGA. The ANN used was an RBFANN with a *Gaussian* function as the basis function.

An RBFANN is a feed-forward artificial neural networks with one hidden layer. Its purpose is to perform non-linear mapping between an input space and an output space. The output of this ANN can be represented by equation 2.2, where x_k is an input vector from a set of input patterns X and $w = [\alpha, \beta]^T$, where $\alpha = [\alpha_0, \alpha_1, \ldots, \alpha_n]$ is the vector of scalar linear parameters, and $\beta = [\beta_1, \ldots, \beta_n]$ is composed of n β_i vectors of non-linear parameters, each associated with a neuron.

$$\hat{y}(x_k, w) = \alpha_0 + \sum_{i=1}^{n} \alpha_i \phi_i(x_k, \beta_i) \tag{2.2}$$

Equation 2.3 formulates the Gaussian function, where $\beta_i = [c_i, \sigma_i]$, the non-linear parameter vector is composed of c_i, a point in the input space that represents the center of the Gaussian function and σ_i that represents the corresponding spread.

$$\phi_i(x_k, \beta_i) = exp(-\frac{1}{2\sigma_i^2}||x_k - c_i||^2) \tag{2.3}$$

2.2.4 Levenberg Marquardt Algorithm

Before the first iteration of the training algorithm the model parameters have to be initialized. The approach taken is the use of Optimal Adaptive K-means Clustering to find the centers c_i. The goal of k-means clustering is to partition the domain of the input pattern vectors into K clusters [10]. This algorithm is a variation of the K-means clustering algorithm, as it employs a dynamic adjustment of learning rate. The learning rate varies, within the range $[0, 1]$, based on the difference between the quality of the partition and that of an optimal partition. The farther the current partition is from the optimal the higher learning rate, helping the partition improve quickly, while the closer they are, the lower the learning rate. As the center discovery through k-means clustering has some randomness, namely in the initial center choice, the resulting model can be better or worse. This leads to the possibility of initializing a model multiple times and keeping the one with best results.

The spread values σ_i can then be calculated using equation 2.4

$$\sigma_i = \frac{z^{\max}}{\sqrt{2n}} \tag{2.4}$$

Where z^{max} is the maximum Euclidean distance among the initial centers c_i, and n is the number of neurons.

Afterwards, a training criterion must exist such that for a given set of input patterns, the ANN will be trained to find values for w such that equation 2.5 minimizes the sum of squared errors between the expected and predicted values.

$$\Omega(X,w) = \frac{1}{2}||y - \hat{y}(X,w)||^2 \tag{2.5}$$

Since the model output is a linear combination of the neuron activation functions output, it is possible to write 2.5 as:

$$\Omega(X,w) = \frac{1}{2}||y - \phi(X,\beta)\alpha||^2 \tag{2.6}$$

The computation of the optimal value α^* of the linear parameters α with respect to the non-linear parameters β, as a least-squares solution

$$\alpha^* = \phi^+(X,\beta)y \tag{2.7}$$

In equation 2.7 "+" denotes a pseudo-inverse operation. By replacing equation 2.7 in 2.6, the training criterion to compute the non-linear parameters is obtained:

$$\psi(X,\beta) = \frac{1}{2}||y - \phi(X,\beta)\phi^+(X,b)y||^2 \tag{2.8}$$

The criterion is independent of the linear parameters α. Whatever values β takes, the $\alpha*$ used will always be the optimal ones. Using the criterion it is possible to iteratively minimize 2.8 to find β^*, which corresponds to searching for the best non-linear mapping. It is then possible to solve 2.7 using β^* to obtain the complete optimal parameter vector w^*.

The training is done using the Levenberg Marquardt (LM) algorithm, as it is recognized as the best method for solving non-linear least-squares problems since it exploits the sum-of-squares characteristic of the problem.[11]

The LM method is of the *restricted step* type. A search direction p_k is computed such that $\Omega(W_k + p_k) < \Omega(w_k)$. It defines a neighborhood of w_k, where a quadratic function agrees with $\Omega(w_k + p_k)$. The step pk is restricted by the region of validity of the quadratic function which is obtained by formulating in terms of pk a truncated Taylor series expansion of $\Omega(wk + pk)$. Then, it may be shown that pk can be obtained by solving the following system‘ [1]:

$$(J_k^T J_k + v_k I) p_k = -g_k \tag{2.9}$$

g_k and J_k are, respectively, the gradient and Jacobean matrix of $\Omega(w_k), v_k \geq 0$, is a scalar value controlling the magnitude and p_k controlling the direction. The gradient g_k can be obtained as such:

$$\begin{aligned} g_k = \frac{\partial \Omega(w_k)}{\partial w_k} = \\ = -J_k^T e_k \end{aligned} \tag{2.10}$$

Where the Jacobean matrix has the form:

$$J_k = \begin{pmatrix} \frac{\partial y_i}{\partial w_1} & \cdots & \frac{\partial y_i}{\partial w_l} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_N}{\partial w_1} & \cdots & \frac{\partial y_N}{\partial w_l} \end{pmatrix} \tag{2.11}$$

The previous equations were extracted from [1].

In the following algorithm ratio r[k] is employed to measure the accuracy to which the quadratic function approximates to the actual function. It is calculated using the following equations [12]:

$$r[k] = \frac{\Delta\Omega[k]}{\Delta\Omega^p[k]} \tag{2.12}$$

$$\Delta\Omega[k] = \Omega(w[k]) - \Omega(w[k] + p[k]) \tag{2.13}$$

$$\Delta\Omega^p[k] = \Omega(w[k]) - \frac{(e^p[k])^T (e^p[k])}{2} \tag{2.14}$$

$$e^p[k] = e[k] - J[k]p[k] \tag{2.15}$$

For a given iteration **k** of the LM algorithm the following steps would be executed:

1: Calculate $J[k]$ using 2.11
2: Calculate the error vector, $e[k]$
3: Solve 2.9 to obtain p_k
4: **while** $(J^t[k]J[k] + v[k]I)$ is not positive definite **do**
5: $\quad v[k] := 4v[k]$

6: Evaluate $\Omega(w[k]+p[k])$
7: Calculate $r[k]$ using 2.12
8: **if** $r[k] < 0.25$ **then**
9: $\quad v[k+1] := 4v[k]$
10: **else if** $r[k] > 0.75$ **then**
11: $\quad v[k+1] := \frac{v[k]}{2}$
12: **else**
13: $\quad v[k+1] := v[k]$
14: **if** $r[k] \leq 0$ **then**
15: $\quad w[k+1] := w[k]$
16: **else**
17: $\quad w[k+1] := w[k]+p[k]$

This results in either accepting or rejecting $w[k]+p[k]$. This algorithm is initialized with $v[0] = 1$.[12]

2.3 The Legacy MOGA Platform

This section details the structure and functionality of the legacy platform. It exists as a cluster of virtual machines set up in a Virtual Private Network (VPN) working as a distributed system. It is currently composed of one master, four servers and a gateway virtual machine that is used to access the system. The master is the central point of communication of its cluster. Any requests made by a user will be received and further processed by the master. It is also responsible for performing part of the MOGA algorithm and storing information regarding application users and generated models. The slave machines exist to perform the training of the models. Each possesses an instruction queue that the master will connect with and convey instructions to start or stop. The gateway machine can be accessed via Remote Desktop Protocol (RDP) by a user. It contains an interface that allows the user to interact with the platform.

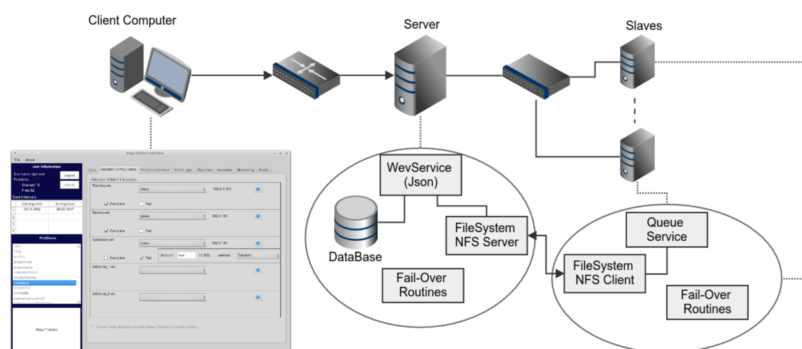


Figure 2.6: Overview of the legacy MOGA platform

An interaction with the platform will start by accessing the gateway virtual machine and opening the interface. It is then necessary to authenticate as a user and set the IP address of the master virtual machine that will be accessed. Theoretically, multiple masters can exist, making it possible to access multiple clusters. However, this would require a user account for each master as each would store its users, problems and results. After authenticating a user can then create and execute *problems*. A problem, in the platform's context is a structure, associated with one user at a time, that contains all the specifications of a problem:

- Variable configuration - Configures what parts of the train, test and validation sets will be used.
- Type of problem - Static Mapping and One-Step Ahead Prediction are supported. For Prediction it is possible to set the input delays we wish to use for the specific problem, for multiple variables if so desired. The set of input lags provided will be used to generate the models in the MOGA algorithm.
- Model specific configurations
  - Upper and lower bounds for the number of neurons.
  - Upper and lower bounds for the number of input terms.
  - Initialization method for the Radial Basis Function Neural Network.
  - Number of training trials for the model.
- Objectives to optimize - These can be selected at will
  - Root Mean Square of Training error.
  - Root Mean Square of Testing error.
  - Forecasting error within a Prediction Horizon for time-series.
  - Number of False Positive Training.
  - Number of False Positive Testing.
  - Number of False Negative Training.
  - Number of False Negative Testing.
- Execution configurations
  - Maximum number of iterations
  - Number of Runs
  - Population size
  - Proportion of random immigrants

- Selective Pressure
- Crossover Rate

A user would input a data file and configure the problem. The platform would then verify the configurations and if the user has allotted time to execute the problem. After successfully finishing the verifications the Master generates the initial population and stores it in a local database, created specifically for the problem. At the same time, each worker will launch as many processes as the number of CPU cores it possesses and start polling for models that were not trained. When a process finds one, it will change a specific database column representing its status and retrieves its representation. It will then proceed to create the ANN based on the retrieved representation and starts training it as discussed in the previous subsections. After the training is performed, the resulting weights, initializations, training, testing and validation errors are stored in the database and the status column is changed to trained.

After all the models in a generation are trained the Master then proceeds with the next steps of the MOGA: Evaluation; Fitness Assignment; Mating Selection; Recombination; Mutation. When the new generation is created, the models will be stored in the database, where the workers will continue training them. This happens iteratively until a generation limit is reached or, the problem is stopped manually via the interface.

Finally, after the problem has finished executing, the user has access to a part of interface that permits analysis of the results. This part of the interface contains tables that allow the user to access the minimum, maximum and mean error values for training, testing, validation and prediction, as well as tables the data for each individual model. This data can be further analysed by downloading it from the interface and using Matlab or Python scripts.

2.4 MOGA Platform applications

The legacy MOGA platform has been used in the past for multiple works. This section will describe the influence of MOGA in these works.

- In [1] the authors utilize MOGA in two benchmark prediction problems. The first deals with the prediction of energy consumption of Portuguese electricity consumption profile for the company *Rede Eletrica Nacional* (REN), within a horizon of 48 hours. Where, compared to previous results, better prediction accuracy was achieved over a longer prediction horizon, and faster convergence (in number of generations) was observed in the MOGA execution. The second problem deals with the prediction of global solar radiation. The features of all-sky images were used, along with cloudiness values as the exogenous input to estimate and predict solar radiation at a given ground location. The results show that the selected ANNs outperformed other methods such as Ridler, Calvard and Trussel (RCT) algorithm and Otsu's method.

- In [13] ensemble techniques are used to improve the forecasting performance of ANN models designed my MOGA for load demand prediction. When compared to other works in the literature results achieved were either comparable or better.
- In [14] a RBFANN model designed by the MOGA framework is use in automatic identification of Cerebral Vascular Accidents (CVA) through the analysis of Computed Tomographic (CT) images. The best results were obtained from an ensemble of models, a value of 98.01% was achieved for specificity and 98.22% for sensitivity at pixel level.
- In [15] Intelligent Model Based Predictive Control (IMBPC) is used to control Heating, Ventilation and Air Conditioning (HVAC) systems using models generated by MOGA. It was installed in a building of *Universidade do Algarve* enabling the control of three classrooms. Dependent on outside temperature and occupation, it is claimed that savings in the order of 50% are expected.
- In [16] a comparison is made between ANN models designed using statistical and analytical methods, where a group of ANNs was designed using MOGA. The models were used to predict electric power demand at the solar energy research center CIESOL in Spain. The experimental results show that the models obtained through MOGA performed comparably to the model obtained through a statistical and analytical approach, while using only 0.8% of the data samples and having a lower model complexity.

2.5 ASMOD

Adaptive Spline Modelling of Observational Data (ASMOD) is an algorithm used to empirically model observable data. It uses basis spline (B-spline) basis functions to construct a mapping between an input variable $x \in R^n$ and an output variable $y \in R$. The model can be described as:

$$m(x) = \sum_{i=1}^{K} c_i b_i(x) = c^T b(x) \tag{2.16}$$

Where $b(x)$ is a vector of B-spline basis functions, and c is a coefficient vector. In a B-spline model, a set of basis functions is defined as a *knot vector*, $\tau = \{\tau_1, ..., \tau_{k+p+1}\}$, consisting of an ordered sequence of real values. Each basis function is a composition of a set of polynomials of a given degree p, each defined over disjoint subintervals of the input space.

ASMOD tries to overcome the curse of dimensionality problem of modeling in high dimensional input spaces by utilizing an iterative model refinement procedure, that, for each step, either grows to better model the data or is pruned to generate a simpler model. Detailed in [17], a general ASMOD algorithm would be:

1: Let the initial model structure be the current model structure
2: Let $i = 0$, and let the *stop refinement criterion* be False
3: **while** *stop refinement criterion* is False **do**
4: Let $i = i + 1$
5: From the current model structure, generate a set $\mu_i = \{M_1, M_2, \ldots, M_{N_i}\}$ of candidate model structures
6: Estimate the parameters in each model structures in μ_i. Denote the estimated parameter vectors by $\hat{c}_{i,j}, j = 1, \ldots, N_i$ i.e.

$$\hat{c}_{i,j} = \arg \min_{c \in \mathrm{A}_{M_j}} I_{emp}(c, M_j, l)$$

7: Compute a criterion function $g(M)$ for all candidate model structures.
8: Select the model structure with the smallest value of the criterion function $g(M)$ as the new current model structure. Denote this model structure by $\hat{M}_i$, and denote the corresponding parameter vector from 3c by $\hat{c}_i$. That is:

$$\hat{M}_i = \arg \min_{M \in \mathrm{M}_i} g(M)$$

$$\hat{c}_i = \arg \min_{c \in \mathrm{A}_{\hat{M}_i}} I_{emp}(c, \hat{M}_i, l)$$

9: Compute the *stop refinement* criterion
10: The identified model structure $\hat{M}$ is the one which gives the minimum value of $g(\hat{M}_j), j = 1, \ldots, i$, and the identified model within this structure is given by the corresponding parameter vector $\hat{c}$, i.e.

$$\hat{M} = \arg \min_{M \in \{\hat{M}_1, \ldots, \hat{M}_i\}} g(M)$$

$$\hat{c}_i = \arg \min_{c \in \mathrm{A}_{\hat{M}}} I_{emp}(c, \hat{M}_i, l)$$

Where i represents the iteration number in the refinement procedure. In the last step of the algorithm, the model that minimizes g is selected. ASMOD will be used in Section 4.4.

Chapter 3

Literature Review

As mentioned in the introduction chapter, this work explores the use GPUs to accelerate the process of generating RBFANN models. This chapter will present a literature review and go over the state of the art of the main two components, namely the constructive algorithms and the paradigm for accelerating code with GPUs.

3.1 Constructive Algorithms

ANN models can be put to use in solving problems of the classification and regression types. There is, however no systematic method for designing ANNs, and while the manual design of ANNs is possible for problems where prior knowledge exists, it would commonly result in a trial-and-error process. Thus, an algorithm that is capable of finding an appropriate architecture for the ANN is quite desirable.

In [18] Kwok and Yeung provide a review of the constructive algorithms for structure learning in feedforward neural networks for regression problems, where they create a taxonomy for these algorithms. It is stated that, regardless of the weight training mechanisms used, they are only effective if a proper architecture is chosen. A comparison is also made between the size of the architecture of an ANN and the use of polynomials for data approximation, where too few coefficients can't capture the underlying function, while a polynomial with too many coefficients will fit the noise in the data and result in a poor representation.

The focus of the paper was mainly the review of constructive algorithms, where the algorithms start with a small network and systematically increment the number of hidden units and weights until an acceptable solution is found. Aside from constructive algorithms, pruning algorithms are mentioned, where a large network is first trained and unused hidden units and weights are iteratively removed until a satisfying solution is achieved. Lastly, the genetic approach is mentioned, where, by converting the search for the optimal architecture to a search space, the genetic algorithm could provide good results, although with an increase in time and space complexity compared to the previous algorithms.

In [2], Fahlman and Lebiere introduce the *Cascade-Correlation* learning architecture, one of the earlier appearances of constructive algorithms in the literature. At the time, the existing learning algorithms were slow, this happened due to simultaneously updating the network weights, thus making a constantly changing environment. The *cascade architecture* proposes adding hidden units, one at the time, in an effort to maximize the magnitude of the correlation between the unit's output and residual. The new hidden units are connected to the original inputs and each pre-existing hidden units, forming a cascade, visible in figure 3.1. The newly added hidden unit's input weights are frozen at the time it is added, and only its output connections are trained repeatedly.

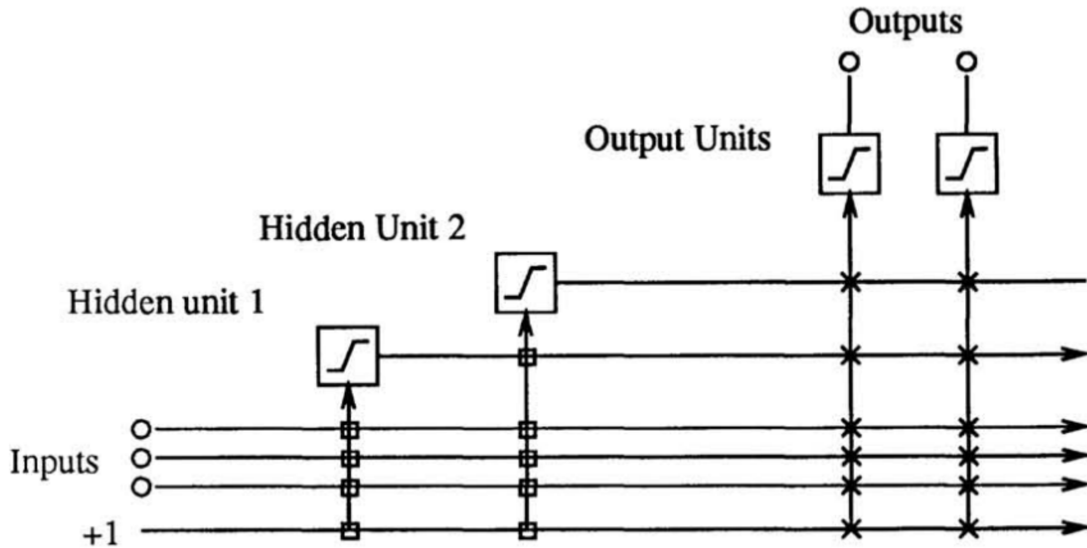


Figure 3.1: The Cascade architecture, after two hidden units have been added. The vertical lines sum all incoming activation. Boxed connections are frozen, X connections are trained repeatedly. Source:[2]

In [19], Subirats, Franco and Jerez propose *C-Mantec* as neural network constructive algorithm, that combines competition between neurons with a stable modified perceptron learning rule. It uses the thermal perceptron learning rule, which is a modification of the original perceptron learning rule [20]. It aims to provide a successful and stable linearly separable approximation to a non-linearly separable problem. This modification is necessary as the original perceptron learning rule guarantees convergence in a finite amount of time if the problem is linearly separable. When compared over a number of datasets with DASG algorithm [21], that produced the best known results at the time, it outperformed it by 39.08%.

The authors of [22], presents an interesting approach to constructive algorithms by using ensembles with the goal of solving complex real-world problems. An algorithm called constructive neural network ensemble (CNNE) is proposed for training cooperative neural networks. The algorithm determines the number of neural networks necessary and the number of hidden nodes for each. Negative correlation learning is used in the networks' training stage promoting diversity in the ensemble, while individually a constructive approach was taken.

While genetic algorithms tend to not be referenced much in works on the topic of constructive algorithms, such as [18], they still fulfill the same role as the 'generic' constructive algorithms in designing neural networks. In [23], Yao and Shi introduce an algorithm for designing artificial neural networks using co-evolution, one of earlier works using evolutionary algorithms. They conceptually convert the problem into a space search one, where each point represents an architecture, where the highest point would correspond to the optimal architecture design. In this work, each individual corresponds to a rule and the population corresponds to a set of rules

that are used to generate one ANN architecture cooperatively.

In a more recent work [24] Behnood and Golafshani introduce an algorithm called *Multi-Objective Grey Wolves Optimization* (MOGWO) for creating ANN models to predict the trend of the compressive strength of silica fume concrete. Although the work leans more towards the field of material engineering, it still provides interesting insights due to some similarities with the MOGA used throughout this thesis. The MOGWO algorithm attempts to construct various ANN models with different architectures and accuracies. The individual (wolf) is encoded in two parts, the first uses binary values to indicate if the ith hidden layer is active, the second part indicates the number of neurons in the ith layer. The algorithm proposed uses four types of wolves, $\alpha, \beta, \delta, \omega$. In the optimization algorithm, the α, β and δ wolves locations are the first, second and third best solutions in the search plane, these influencing the ω wolves to converge towards them. Afterwards the individuals' ANN models are evaluated, the roles reassigned ($\alpha, \beta, \delta, \omega$) and the algorithm progresses iteratively, where afterwards a pareto front is formed with the best solutions. An interesting detail in the training of the ANN models is the that the Levenberg-Marquardt algorithm is used, and due to the random weight initialization, it is run 10 times, where the best model is saved as the individual. This mechanism is similar to the one used in MOGA, although in MOGA it is possible to manually tune the parameter.

In [25] Lin and Hu propose *Artificial Neural Network-Particle Swarm Optimization* (ANN-PSO), a novel algorithm for meta-heuristically designing ANNs, considering topology, activation functions and training algorithm as the main design factors. The individuals (particles) are codified in three parts, the first consists of a string of bits that represent whether the neurons exist in the hidden layer of the ANN, the second contains the synaptic weights and the last is a representation of one of six types of activation functions (sigmoid function, hyperbolic tangent function, sinusoidal function, Gaussian function, linear function, or hard-limit function). The approach taken in the algorithm is similar to [24] and possesses some similarities to MOGA. In the algorithm, a set of particles is generated, each with its speed and position. As each particle contains the representation an ANN, the ANNs are generated, trained and evaluated and the best local and global values are stored. The particles are then updated based on their position and speed, as well as the local and global best values. This process is repeated iteratively until a minimum error tolerance is achieved.

3.2 GPU paradigm

In the book [26] an introduction is provided on how to program CUDA enabled GPUs. The initial chapters of the book provide insight on the differences between the CPU and GPU. The concepts of *latency* as the time a task takes to be completed and *throughput* as the amount of tasks completed in a given time frame are introduced. It is explained that CPUs are designed to optimize latency, in part due to their generally higher clock rates and GPUs are designed to

optimize throughput by executing multiple instructions simultaneously due to its large number of computational units.

At a hardware level, it is explained that the GPU possesses multiple parallel processing units called *cores*. These units are divided into streaming processors and streaming multiprocessors, that exist on a grid. When a CUDA program is executed, it runs in parallel on a number of threads, where each thread is executed on a different core.

On a software level, there exists a distinction between the GPU and the rest of the system. The system is called *host* and the GPU is called *device*, these two act as independent parts of the system with different memory, memory pointers and clocks. Any CUDA program would first allocate memory on the host and device, then copy the relevant data from the host memory to device memory, launch the *kernel*, a function that will run on the device, perform the necessary computation on the data, copy the results back to host memory and free all the memory used.

It is further explained that the any GPU kernel runs on CUDA C, a programming model developed by NVIDIA that allows a programmer to interface with the GPU. Although the book explains the initial chapters in CUDA C, it then suggests the use PyCuda for Python, one the many wrapper libraries that exist for executing GPU on other programming languages. The one that was used mainly in this thesis was CuPy. An introduction as well as some examples are provided in [27]. CuPy is a library based on NumPy, a popular library used in scientific computing, as it provides multidimensional arrays as well as many functions. It was developed so that it is similar to NumPy in use and functionality but is capable of executing on GPU.

In [28], PyTorch is used to train ResNet models using GPUs. The ResNet models use repeated well-designed residual blocks, allowing training of very deep networks to high accuracy while maintaining high GPU utilization. The authors refined the existing ResNet50 model, creating three variants with differing depths and number of channels. A number of code optimizations were also performed to enchance the GPU throughput of the models. The just in time compilator from PyTorch was used to pre-compile parts of the network to C++. Some operations such as ‘mean’ and ‘view’ were given custom implementations and the use inplace operations that directly change the content of a tensor, without making a copy all contributed to improving throughput. This work claims that their TResNet models surpasses the accuracy of state-of-the-art works with improved GPU training and inference throughput on commonly used datasets.

In [29], the authors propose a deep neural network training system called HetPipe (Heterogenous pipeline). Using a heterogeneous GPU cluster that possibly includes whimpy GPUs that, as a standalone, could not be used for training HetPipe is implemented as a solution. Using four different GPUs: TITAN V, TITAN RTX, GeForce RTX 2060, QuadroP4000, with different specifications they exploited model parallelism to train large deep neural network models. The implementation is centered around virtual workers, groups of multiple GPUs that process minibatches in a pipelined manner. To synchronize these workers, the authors propose

a novel algorithm for parameter synchronization, Wave Synchronous Parallel (WSP). WSP is adapted from the Stale Synchronous Parallel model [30] to accommodate both pipelined model parallelism and data parallelism for virtual workers composed of heterogeneous GPUs. The referenced implementation results of modifying TensorFlow 1.12 version with CUDA 10.0 and cuDNN 7.4. The authors further claim that HetPipe is capable of efficiently training large deep neural network models with fast convergence.

Chapter 4

Methodology

4.1 Documentation

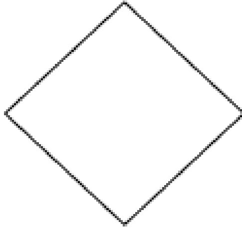
The original MOGA platform suffered many changes from its inception until reaching the state in which it was when this work started. At least four developers had worked on the platform before, in disjointed time periods ranging from 1993 to 2015. These developers created code according to their styles and often with minimal or no documentation. This resulted in a platform that is complicated to operate. All of this on top of the already complex algorithms related to the genetic algorithm, neural network training and prediction algorithms.

The first step to fulfill the objectives set for this work is the understanding of the platform and how the different components interact with each other. It was deemed beneficial the creation of documentation as well as diagrams that could express the actions of the code. At this point in time some works in the literature were consulted previous to the start [1][31] related to MOGA and [26] [32] which were later used as reference for the GPU implementation component of this work.

The creation of such documentation allows for the possibility of consulting it further down in development for reference. It would also create a base to facilitate work for other developers that would eventually work on the system's code. Through this process a general understanding of the system would be acquired to further complete the other objectives of this work.

The documentation is done in the form of flowchart diagrams; these diagrams go over the code files of the project and explain the functions in the form of images accompanied by text and comments. A set of diagrams exists that details major actions across various files, such as the execution of a problem. Where necessary the functions would also have their individual documentation written in the code. These two actions would be performed in parallel as well as making minor changes to the readability of the code where deemed necessary.

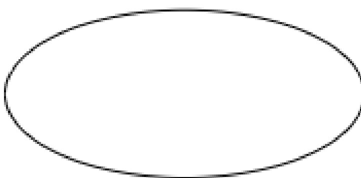
Represents an action or a set of actions that happen in the code.



Represents the 'if' clause in the program, will have multiple arrows exiting it depending on the assertion.

Represents a function call to our code this does not account for external libraries.

Represents an input-output operation. This is mostly used for operations related to logging.



Represents an entry or exit point for the function such as return.

Figure 4.1: Flowchart Legend

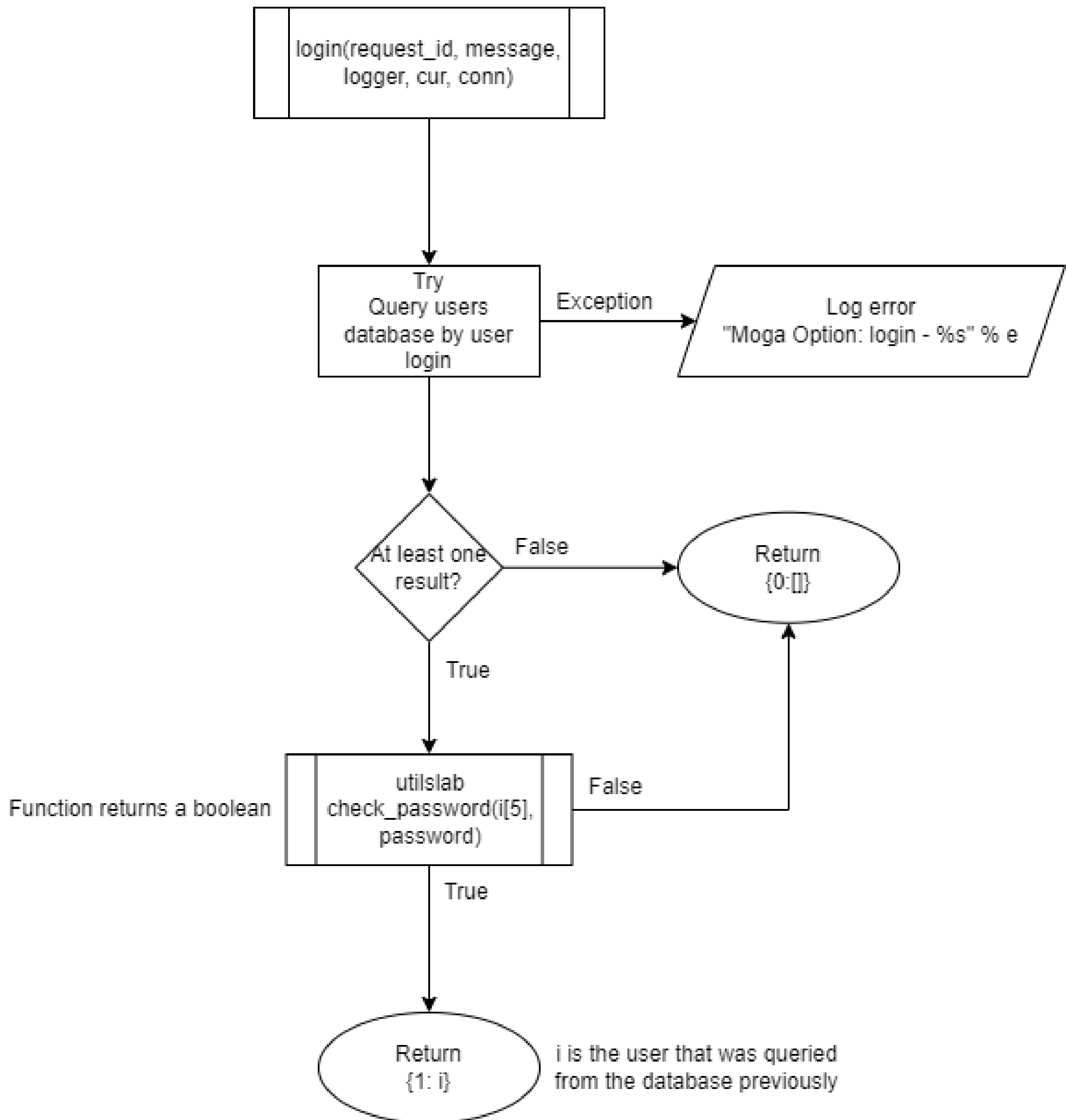


Figure 4.2: Flowchart example 1

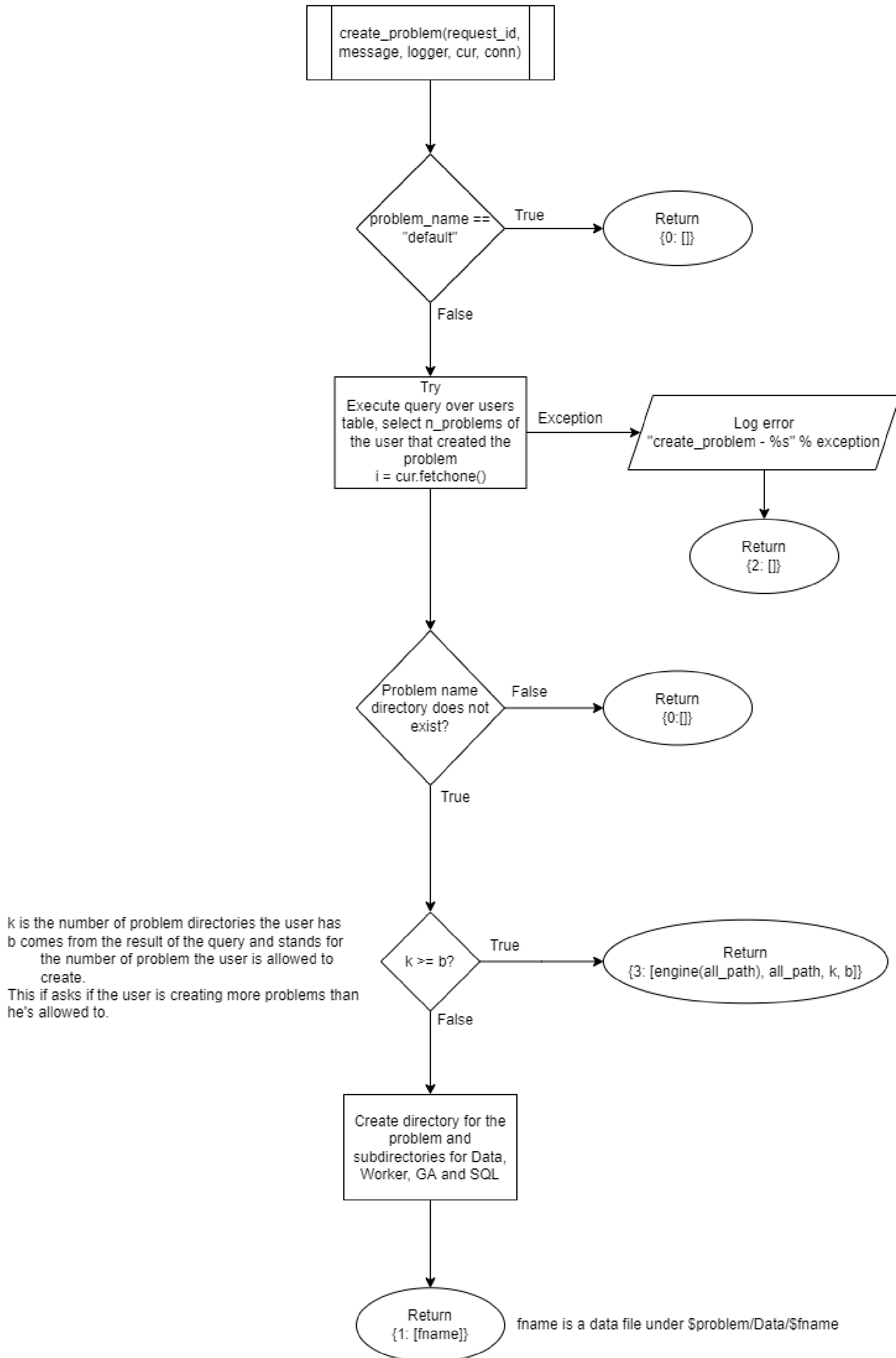


Figure 4.3: Flowchart example 2

Figure 4.1 shows the legend for the diagrams and figures 4.2, 4.3 show two example diagrams. The first one shows how a user's login would be processes, and the second one shows how a problem would be created.

4.2 Hardware

This section details the hardware used in this project:

A SuperMicro AS -2024US-TRT server, which was called 'GPU Server' and installed in the university's data center. The specific configuration of this server contains two NVIDIA A100 Tensor Core GPUs that would be leveraged to speed up the MOGA platform. Its specifications are as follows:

- CPU - Dual AMD EPYC 7282 Series processors 16 Cores @ 2.8Ghz
- RAM - 256 GB of 3200MHz ECC DDR4 RDIMM, LRDIMM
- GPU - 2x NVIDIA A100 Tensor Core GPU (40GB each)

The platform that was used to run MOGA used 6 virtual machines to form a cluster to train the models. In this cluster 1 was the master and the remaining 5 were workers. These virtual machines existed in a cloud provided by the university for research. They used a set of virtualized hardware resources that were limited and allocated to that specific cloud. The virtual machines were all using:

- CPU - 8 cores Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz
- RAM - 16 GB

One of these machines was eventually destroyed to free up resources to create a pair of development virtual machines detailed in the following section.

A personal computer was used for development and as way to compare the performance of GPU Server with a consumer level computer with the following specifications:

- CPU - AMD Ryzen 5 5600X 6-Core Processor 3.70 GHz
- RAM - 16 GB
- GPU - NVIDIA GTX 1050 Ti (4GB)

4.3 Legacy Platform Updating

The first step to fulfilling this work's objectives was to create a steady based. Initially what was provided as a codebase was a set of files that contained both the master and worker components of the MOGA platform as well as the source code for the graphical interface. The interface

was written exclusively in Python2, while the platform code was a mix of MATLAB, C, SQL and Python2, with the majority of it written in Python2. For the platform components it was necessary to update the Python2 to Python3, as Python2 was no longer supported since January 1st, 2020. The MATLAB portion of code did not need too many changes as MATLAB itself hadn't suffered major changes since the code's creation. It was necessary however to recompile a set of MEX files. A MEX file in MATLAB is a file that details high-performance functions written in C/C++ that executes just like regular MATLAB functions. A recompilation was deemed necessary to ensure no problems with the more recent versions of MATLAB and GNU Compiler Collection (GCC). The C code was written as to be compiled by the Extension library from Python, the library allows for the extending of Python with C or C++. This portion of code defined the activation function and the calculation of the Jacobian matrix for the RBFANN. This portion of code suffered minor changes as it was only necessary to get the equivalent libraries for Python3 and replace them in the code, as well as change the file that compiled the code into a dynamic library to use Python3 over Python2.

The codebase written in Python had an extensive number of files of varying lengths. The use of a library to automatically convert the code was deemed optimal. In the Python documentation on porting Python2 code to Python3 [33] much advice is given on the topic, as well as explaining some major differences between the versions. Some of the most notable differences are:

```
On a syntactic level:
Python2:
print 'Hello'
raise ValueError, "dodgy value"

Python2 and 3:
print('Hello')
raise ValueError("dodgy value").with_traceback()
```

As well as some structure differences such as the 'try/catch' block:

```
Python2:
try:
    ...
except ValueError, e:
    ...

Python2 and 3:
try:
    ...
except ValueError as e:
```

...

The change that caused the most problems in this conversion step from Python2 to Python3 was how the language handled strings. In Python2, the ‘str’ type was used for two different types of values, *text* and *bytes*. In Python3 *text* and *bytes* exist as separate types. As explained in [34], **text** contains human-readable messages, represented as sequence of unicode codepoints. In Python3 this exists as ‘str’ and in Python2 as ‘unicode’. In all the codebase, written in Python2 ‘str’ is used interchangeably for string and byte, where if run in Python3, they would be interpreted as unicode strings instead of a string of bytes. Due to the system’s need to communicate between servers, transfer data and manipulate files (I/O), problems would arise since the majority of these operations are processed using bytes.

To tackle the first two changes just mentioned, among other minor changes, a Python library called ‘2to3’ was used. This library provides automated Python2 to 3 code translation. After the conversion library was called it would iterate through the code in all the codebase and convert the Python2 specific structures and functions to Python3. This procedure did not however make the code immediately functional as the ‘2to3’ library couldn’t fix dependency problems between the Python versions. Thus, it was necessary to go over the libraries used throughout the code, check for the ones that didn’t exist for Python3 or suffered changes and find current libraries that succeeded those, while applying the necessary changes throughout the code.

The previously mentioned graphical user interface (GUI), written exclusively in Python was also updated in the same timeframe. A similar process was applied, where, by using the ‘2to3’ library the code was automatically translated to be run on Python3. It would also need a manual revision of the libraries, the main problem stemming from the ‘PySide’ library, a cross-platform GUI toolkit. This library suffered significant structural changes from the last Python2 compatible version ‘PySide1.2.4’ to the version that was used for updating, ‘PySide6’. As a result, many components used previously didn’t exist in the same form or were changed within the library, provoking the necessity for manual debugging. The simultaneous updating of the GUI’s and the platform’s codebases would allow for easier debugging by using the GUI to interact with a testing platform and experiment with real use cases. A more detailed introduction of the test platform will be provided in the following subsection.

To address the third change mentioned, and as a way to test and debug the updated code, the new code was implemented in the testing platform and the interface was executed used in a personal computer. The use cases for the platform were tried, one by one, and debugged if necessary. This step was necessary to assert if the communication between the GUI and the platform, as well as the communication within itself was working as expected. This was necessary to make sure the platform was interpreting ‘strings’ as ‘strings’ and ‘bytestrings’ as ‘bytestrings’. The task was executed, along with a Secure Socket Shell (SSH) connections to the test platform’s computers, allowing for access to logs generated by the various parts of the platform in real-

time.

4.3.1 Development Platform

The updated codebase required a testing environment. As such, it was decided that two virtual machines would be created to act as master and worker in order to ensure the updated codebase would perform in the same way as the legacy one. These virtual machines were created using spare resources existent in the cloud where the legacy MOGA platform was hosted. The machines were created as to replicate the ones in the existing cluster in terms of hardware capabilities.

Both virtual machines were created using the same method. By using System Center 2019 and accessing the cloud where the cluster resides, one can create and manage the virtual machines. An internal document exists detailing the procedure but in general terms it would go as follows:

1. Installing the Operating System (OS). *DEBIAN 11* was chosen as it is a more up-to-date version of the OS ran in the other cluster machines.
2. Setting up the machines' Internet Protocol (IP) addresses, gateway, name resolution servers and adding each other to the host names file.
3. Checking if the virtual machines can reach each other through the 'ping' command.
4. Update the package manager and install an ssh server, as well as start the service. This enables ssh connections allowing one to connect without using System Center software.

At this point in the process the installations would differ for each. For the master machine Python, Package Installer for Python (PIP), MATLAB, Postgres SQL and the MOGA components relevant to the master would be installed. Most of the software necessary can be installed through the package manager, Advanced Package Tool (APT) and the Python dependencies through PIP. For MATLAB, however, a GUI is required, which is not possible in the terminal based environment that was provided. As such, a workaround using X11 forwarding was used to mirror the MATLAB installer on the computer the developer was connecting from, allowing one to use the regular MATLAB installation GUI. The MOGA components would be copied to the machine and installed using a custom script that would install the different libraries, as well as create the main location where problems would be stored. With the existence of this main location, a Network File System (NFS) server kernel would be installed and configured. The NFS would allow whitelisted worker virtual machines to access the main location of MOGA. This approach was taken due to the architecture of the platform, where many workers would need to access a problem's configuration and data files stored in the master machine. Afterwards it is necessary to configure the Postgres SQL database, creating a whitelist of all the servant's IPs that should be able to interact with it.

The servant machine goes through a similar installation process, although with less software. Each worker would have Python, PIP and the MOGA servant components installed as well as the Python dependencies. It would then connect to the NFS server in the master machine as to access the main location of MOGA.

The last step to have a functioning platform would be to have the necessary scripts running at all times to receive input from the GUI and convey it to the servants as necessary. To achieve this, daemons would be used. Daemon refers to a program that is run as a background process to perform specific tasks or provide services, without direct interaction from a human user. In the legacy platform these daemons existed under the Linux Standard Base specification format that was discontinued for Debian in 2015, according to [35]. The daemons were recreated in a *systemd* version and would now be controlled through the they *systemctl* command. The master has two daemons, one that would run and restart a server that would communicate with GUI clients that connected to it and each servant has one daemon that would handle instructions sent by the master. Below is an example of a *systemd* daemon to start a python script on boot and restart it on failure:

Listing 4.1: *Systemd* daemon example

```
[Unit]
Description=MOGA Interface Server
After=network.target

[Service]
ExecStart=/bin/bash -c '/bin/python3 $WORKPATH/SLAVE/moga_work_handler.py'
Restart=on-failure
User=$RUN_USER
Group=$RUN_USER

[Install]
WantedBy=multi-user.target
```

4.4 Performance Data Collection

After establishing a functioning testing platform, the next stage would be to perform an analysis of the performance of the system. This section will go over the data collection process, results and analysis of the performance time and memory usage of the testing platform for various test cases. The objective of this part is to help discern how much time certain operations take in the training of the RBFANNs and how much memory the whole process would use. The data generated would be used to try and create models that could estimate the time taken and mem-

ory used based on a problem's configuration using ASMOD, detailed in one of the following subsections. It would also provide insight on the operations that need to be specifically ran on GPU. Figure 4.4 illustrates this concept.

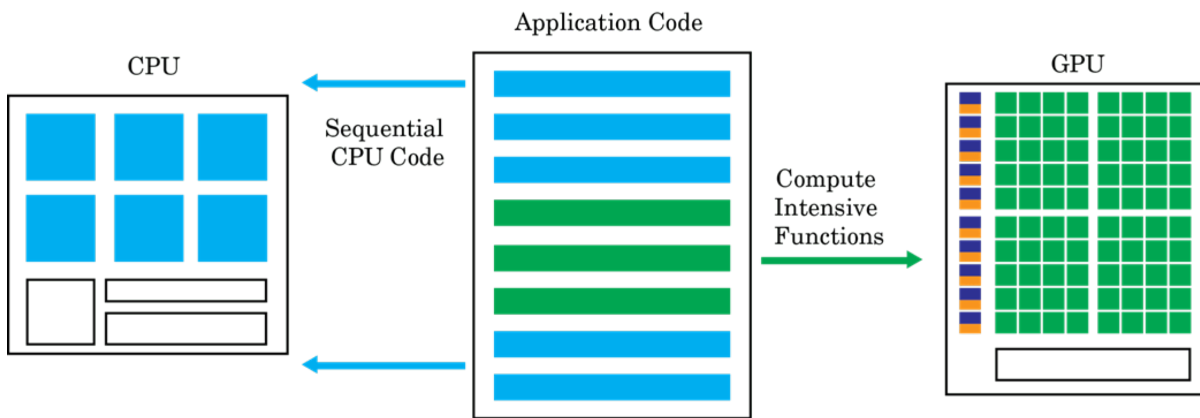


Figure 4.4: Code acceleration on sequential CPU code

4.4.1 Methodology

After deliberation, it was decided that the part of the code that was the best candidate for acceleration was the training of the RBFANNs in the servants' side. Through analysis, this part of the code can be divided in these operations:

- RBFANN Creation and initialization (clustering and weight assignment)
- The Levenberg Marquardt algorithm
- Prediction, if the problem was a regression problem.

The training is usually performed on multiple initializations; as such, at the start and end of each, a timestamp was logged with the initialization's number. Then, for each of the operations mentioned, a timestamp was logged along with the operation's name, this for both time and memory values. The logged values would then be analyzed, through the use of a script to produce readable results. This whole process of data extraction was performed on ten regression problems and five classification problems.

4.4.2 Results

Data from about 170 000 models was acquired, resulting from running five classification and ten regression problems using the recording method previously mentioned. This section will present some of the more meaningful results acquired with regard to performance time.

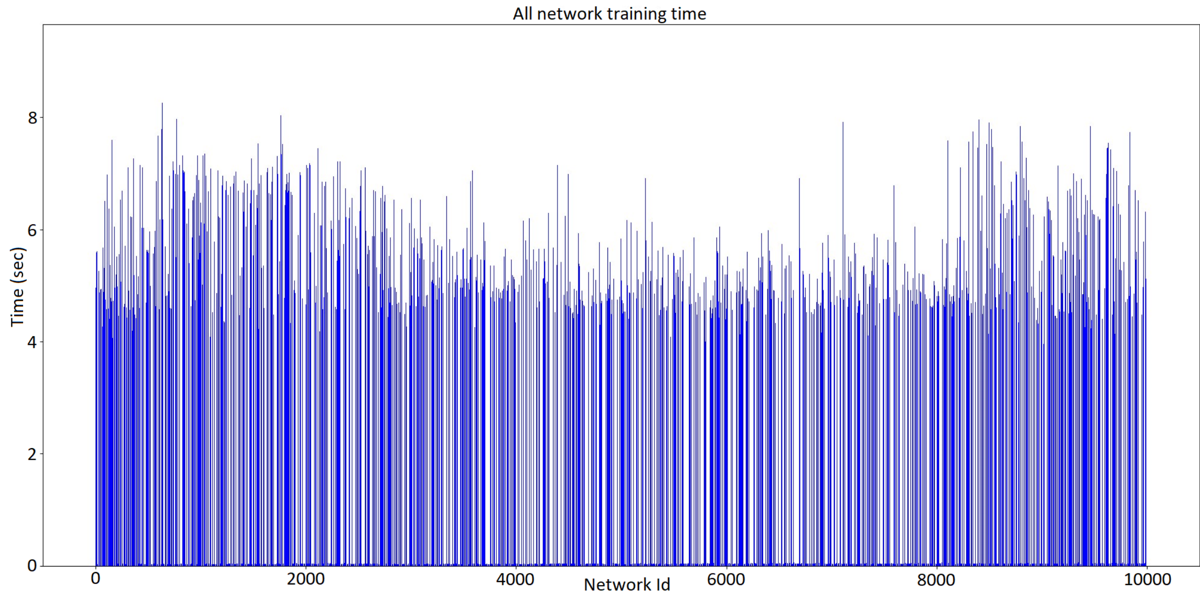


Figure 4.5: All models training time, net_id x time (sec)

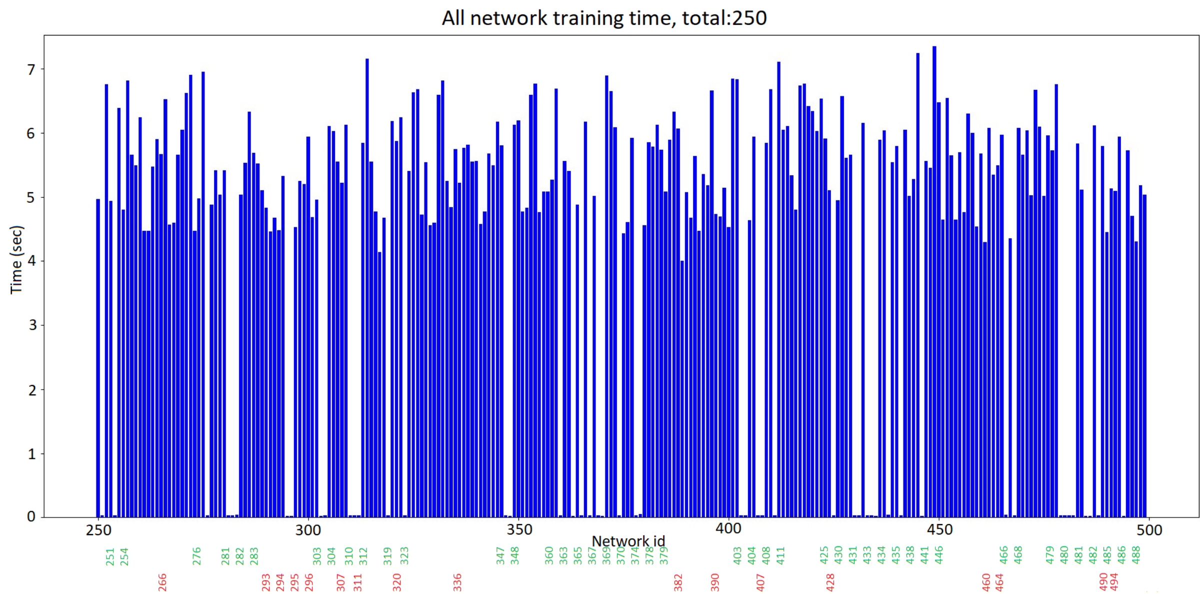


Figure 4.6: 250 models sample training time with outliers, net_id x time (sec)

Figures 4.5 describes the time it took to train each model for a specific problem, each denoted by its specific 'net_id' tag. The values tend to be in the 4 to 8 seconds range, the differences attributed to the different number of neurons and inputs for each model. It is noticeable however that some models have really low training times. Upon investigating, these were associated with 2 specific mechanisms existent in the MOGA platform. One would prevent the training entirely if the model's configuration was similar to one already existent in the ANN models database. It would then copy it is values and be declared as trained, thus taking significantly less time. The other mechanism would abort a model's training if at any point the training process generated

an exception. The model would then be declared as trained and stored in a table specific for bad models. In figure 4.6, a sample of 250 models with sequential 'net_id' values was taken that exposes these two mechanisms better. Below the x-axis, there are two rows of values, where the green ones denote models affected by the first mechanism and the red ones denote models affected by the second mechanism. These models will be filtered in the dataset used for ASMOD in the next sub-section.

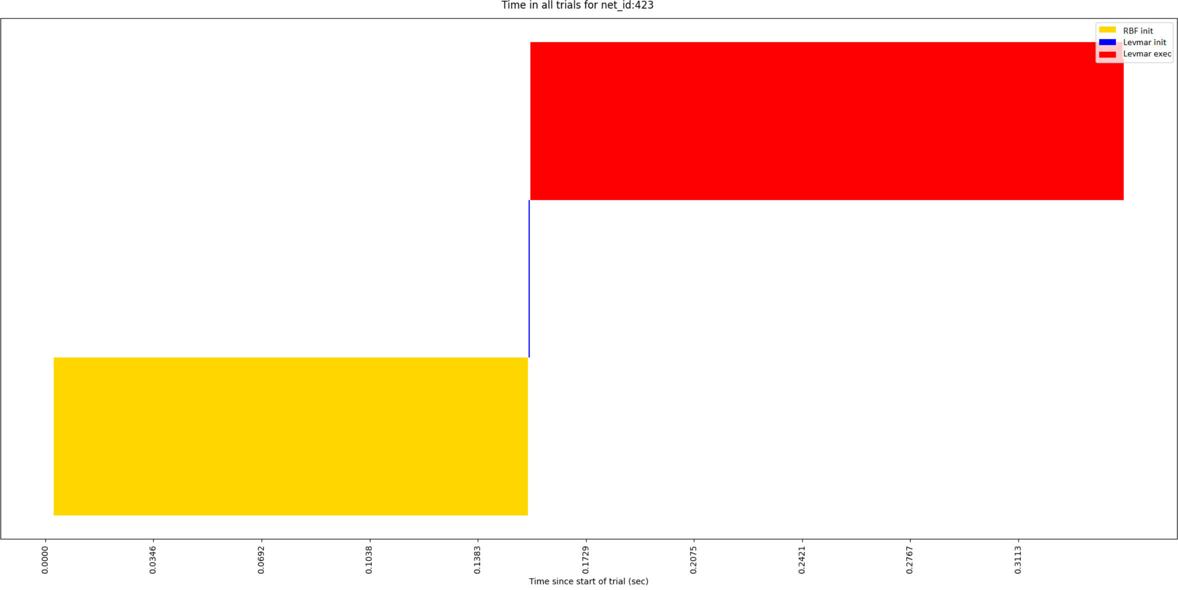


Figure 4.7: Classification problem training time (sec)

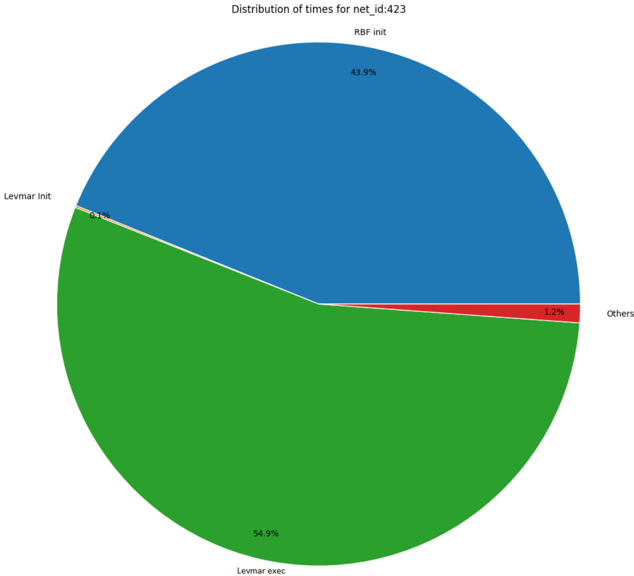


Figure 4.8: Classification problem training time (%)

Figures 4.7,4.8 provide insight on the general distribution of time in the classification problems execution. Within this problem class it is visible that the vast majority of time is distributed

evenly between two operations, the execution of the Levenberg Marquardt algorithm (Levmar exec) and the initialization of the RBFANN (RBF init).

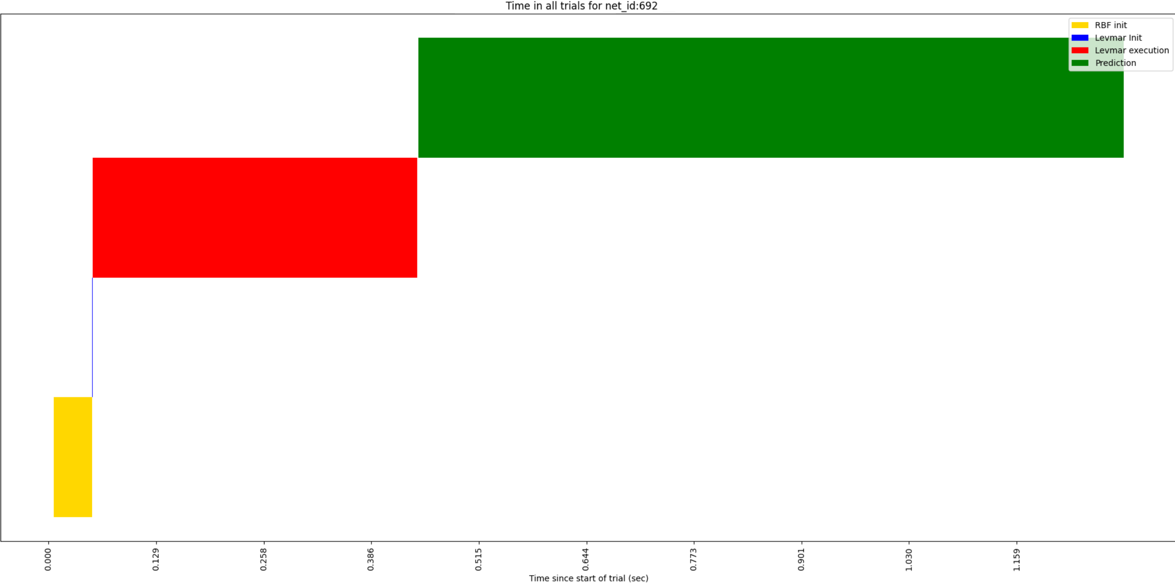


Figure 4.9: Regression problem training time (sec)

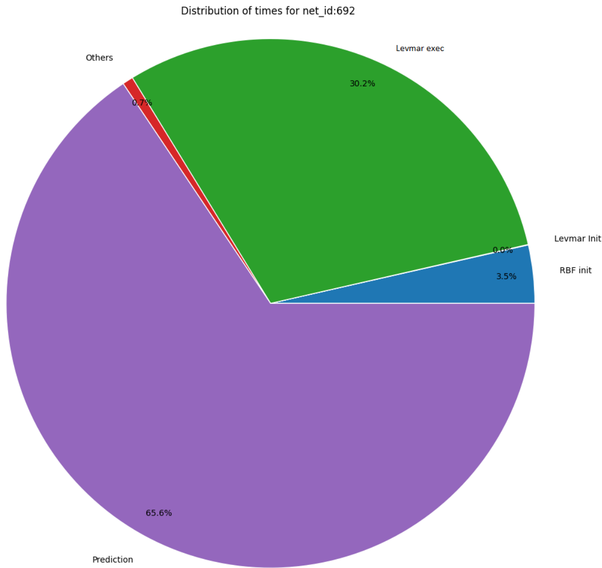


Figure 4.10: Regression problem training time (sec)

Figures 4.9,4.10 provide insight on the general distribution of time in the regression problems execution. Within this problem class it is visible that the overwhelming majority of time is used in the prediction operation, followed by the execution of the Levenberg Marquardt algorithm (Levmar exec) and the initialization of the RBFANN (RBF init). In the regression problems, training is performed similarly to the classification ones, after which the prediction is performed.

It can be concluded that, in general the prediction is going to take the most time, if applicable, otherwise, the next operation to focus on would be the Levenberg Marquardt algorithm. In that order, these would be the main candidates for acceleration, although the calculation of the prediction can not be performed in parallel, as it relies on previously calculated data. The part of the code that performs the activation function could be a candidate for GPU acceleration, possibly improving the performance of the prediction operation. In the Levenberg Marquardt algorithm, the parts of the code related to the calculation of the jacobian matrix, also written in C, as well as the gradient and update calculations could be accelerated.

4.4.3 ASMOD

As previously stated, the capacity to predict how much time and memory a problem would need would be beneficial. Knowing how long a problem would take to complete would be beneficial for a researcher using the platform, and the memory data would prove useful in deciding how many processes can access the GPU's memory at the same time. For this purpose, an implementation of ASMOD was provided and was employed, along with the previously gathered data to create models that could estimate time and memory for each class of problem. Figures 4.11 and 4.12 show a sample of the data that was input into ASMOD.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Id	problem	net_id	num_training	num_testing	num_validation	num_levmar_iter	num_init	num_neurons	num_inputs	rss	PH
2	0	vladimir_rhsala1	1	3656	1218	1220	50	5	5	24	150.56	48
3	1	vladimir_rhsala1	2	3656	1218	1220	50	5	16	17	141.01	48
4	2	vladimir_rhsala1	3	3656	1218	1220	50	5	10	21	138.11	48
5	3	vladimir_rhsala1	4	3656	1218	1220	50	5	10	23	141.01	48
6	4	vladimir_rhsala1	5	3656	1218	1220	50	5	15	31	134.73	48
7	5	vladimir_rhsala1	6	3656	1218	1220	50	5	16	34	150.56	48
8	6	vladimir_rhsala1	7	3656	1218	1220	50	5	11	20	141.09	48
9	7	vladimir_rhsala1	8	3656	1218	1220	50	5	9	33	136.67	48
10	8	vladimir_rhsala1	9	3656	1218	1220	50	5	11	32	141.01	48
11	9	vladimir_rhsala1	10	3656	1218	1220	50	5	6	11	141.09	48
12	10	vladimir_rhsala1	12	3656	1218	1220	50	5	17	26	141.01	48
13	11	vladimir_rhsala1	13	3656	1218	1220	50	5	13	34	134.73	48
14	12	vladimir_rhsala1	14	3656	1218	1220	50	5	17	21	148.26	48
15	13	vladimir_rhsala1	15	3656	1218	1220	50	5	14	28	138.01	48
16	14	vladimir_rhsala1	16	3656	1218	1220	50	5	15	31	138.01	48
17	15	vladimir_rhsala1	17	3656	1218	1220	50	5	15	30	138.11	48
18	16	vladimir_rhsala1	18	3656	1218	1220	50	5	5	30	141.09	48
19	17	vladimir_rhsala1	19	3656	1218	1220	50	5	6	29	138.01	48
20	18	vladimir_rhsala1	20	3656	1218	1220	50	5	9	22	148.26	48
21	19	vladimir_rhsala1	21	3656	1218	1220	50	5	11	7	136.67	48
22	20	vladimir_rhsala1	22	3656	1218	1220	50	5	9	17	136.67	48
23	21	vladimir_rhsala1	23	3656	1218	1220	50	5	14	29	134.73	48
24	22	vladimir_rhsala1	24	3656	1218	1220	50	5	9	19	138.01	48
25	23	vladimir_rhsala1	25	3656	1218	1220	50	5	13	19	148.26	48
26	24	vladimir_rhsala1	26	3656	1218	1220	50	5	8	23	141.01	48
27	25	vladimir_rhsala1	27	3656	1218	1220	50	5	5	7	136.67	48
28	26	vladimir_rhsala1	28	3656	1218	1220	50	5	18	20	150.56	48
29	27	vladimir_rhsala1	29	3656	1218	1220	50	5	15	32	141.09	48
30	28	vladimir_rhsala1	30	3656	1218	1220	50	5	17	26	136.67	48
31	29	vladimir_rhsala1	31	3656	1218	1220	50	5	7	22	150.56	48
32	30	vladimir_rhsala1	32	3656	1218	1220	50	5	14	24	134.73	48

Figure 4.11: ASMOD input data example for regression problems (memory)

	A	B	C	D	E	F	G	H	I	J	K
1	Column1	problem	net_id	num_training	num_testing	num_validation	num_levmar_iter	num_init	num_neurons	num_inputs	time
2	0	vladimir_sc16new	1	2585	510	510	50	5	6	24	1.295798
3	1	vladimir_sc16new	2	2585	510	510	50	5	29	17	10.893564
4	2	vladimir_sc16new	3	2585	510	510	50	5	16	21	5.30867
5	3	vladimir_sc16new	4	2585	510	510	50	5	16	23	6.239949
6	4	vladimir_sc16new	5	2585	510	510	50	5	26	31	31.136131
7	5	vladimir_sc16new	6	2585	510	510	50	5	29	34	51.384556
8	6	vladimir_sc16new	7	2585	510	510	50	5	18	20	6.467055
9	7	vladimir_sc16new	8	2585	510	510	50	5	14	33	8.516181
10	8	vladimir_sc16new	9	2585	510	510	50	5	18	32	13.504138
11	9	vladimir_sc16new	10	2585	510	510	50	5	8	11	1.241809
12	10	vladimir_sc16new	12	2585	510	510	50	5	30	26	31.078173
13	11	vladimir_sc16new	13	2585	510	510	50	5	21	34	22.913769
14	12	vladimir_sc16new	14	2585	510	510	50	5	30	21	18.618664
15	13	vladimir_sc16new	15	2585	510	510	50	5	23	28	17.878434
16	14	vladimir_sc16new	16	2585	510	510	50	5	26	31	31.254559
17	15	vladimir_sc16new	17	2585	510	510	50	5	27	30	31.908541
18	16	vladimir_sc16new	18	2585	510	510	50	5	5	30	1.11479
19	17	vladimir_sc16new	19	2585	510	510	50	5	8	29	2.454312
20	18	vladimir_sc16new	22	2585	510	510	50	5	15	17	3.719493
21	19	vladimir_sc16new	24	2585	510	510	50	5	14	19	3.637049
22	20	vladimir_sc16new	25	2585	510	510	50	5	21	19	7.255597
23	21	vladimir_sc16new	27	2585	510	510	50	5	6	7	0.707159
24	22	vladimir_sc16new	28	2585	510	510	50	5	32	20	19.107473
25	23	vladimir_sc16new	29	2585	510	510	50	5	26	32	33.331529
26	24	vladimir_sc16new	30	2585	510	510	50	5	31	26	31.290455
27	25	vladimir_sc16new	31	2585	510	510	50	5	9	22	2.288306
28	26	vladimir_sc16new	32	2585	510	510	50	5	23	24	12.933156
29	27	vladimir_sc16new	33	2585	510	510	50	5	18	14	3.818649
30	28	vladimir_sc16new	35	2585	510	510	50	5	18	18	5.016195
31	29	vladimir_sc16new	36	2585	510	510	50	5	23	33	26.921403
32	30	vladimir_sc16new	37	2585	510	510	50	5	16	9	2.036963

Figure 4.12: ASMOD input data example for classification problems (time)

A number of models was produced, as result of using various configurations and refining the data in different ways. None of the models produced however provided a satisfactory result. The overall low model quality was attributed to the lack of diverse enough data. As seen in the previous figures, many rows posses the same values. This happens due to some columns referencing problem parameters, like the number of initializations, or training and testing data sizes, that change for every problem but are common to within its models. This leads the datasets provided to ASMOD to not have sufficient features to create correlations.

Since the ASMOD models couldn't contribute to the estimation of time and memory, a different approach was taken. To predict the time a problem would take to run, a graphical interface, usable by a researcher, was envisioned that would allow the estimation of an upper bound based on a minimal execution of a problem with a similar configuration, detailed in 4.5.5. The memory estimation would be done based on a live run of the problem, detailed in 4.5.4.

4.5 GPU Implementation

After the analysis performed previously the next step would be to prototype and implement a GPU accelerated MOGA platform on the GPU server. This machine was installed with 'Ubuntu 20.02' because it was the only Linux distribution compatible with all the software requested to exist in the machine. The installation followed a similar path as the previously detailed worker installation. And RDP server was installed as to provide a way to access the machine with a

graphical environment, allowing it to be used for other purposes. This installation in Linux was performed right before the implementation of the GPU compatible code. Another installation of the GPU server had been done before with Windows Server 2019 as the operating system. This was done during an early stage of the project when certain requirements for the system were not yet set, eventually leading to the change to Linux.

4.5.1 Library Choice

There are a few options when working with CUDA on python. Nvidia provides its own library, 'cuda' with access to low level functions and the ability to create kernels. There exist some other libraries that provide CUDA wrappers for python such as 'PyCuda', 'Numba' or 'CuPy'. Each has their own specificities, but they all rely on the CUDA programming model, essentially compiling a subset of Python code into CUDA kernels. Within these libraries, 'CuPy' stood out as it provided a set of functions similar to the 'numpy' library already in use for data manipulation and some linear algebra functions. As such, due to similarly with 'numpy', the existence of linear algebra functions that are already in use, the 'CuPy' library was chosen.

4.5.2 Prototyping

Two initial tests were conducted using the 'CuPy' library to discover what factor of acceleration can be expected when compared to current operations.

Operation	Matrix Size	Time	Speedup vs MOGA value
LSTSQ(Scypy)	100x100	5.1 ms ± 36.8 μs	(In use on MOGA)
LSTSQ(Numpy)	100x100	5.06 ms ± 21 μs	1x
LSTSQ(Cupy)	100x100	1.16 ms ± 43 μs	4.3x
LSTSQ(Cupy) + Data Copy	100x100	1.24 ms ± 947 ns	4.1x
LSTSQ(Scypy)	1000x1000	196 ms ± 868 μs	(In use on MOGA)
LSTSQ(Numpy)	1000x1000	211 ms ± 2.06 ms	0.9x
LSTSQ(Cupy)	1000x1000	15.6 ms ± 4.39 μs	12.5x
LSTSQ(Cupy) + Data Copy	1000x1000	17.1 ms ± 132 μs	11.4x
LSTSQ(Scypy)	10000x10000	29.8 s ± 250 ms	(In use on MOGA)
LSTSQ(Numpy)	10000x10000	27.4 s ± 106 ms	1.1X
LSTSQ(Cupy)	10000x10000	553 ms ± 1.46 ms	53.8x
LSTSQ(Cupy) + Data Copy	10000x10000	635 ms ± 3.48 ms	46.9x

Table 4.1: Least Squares (LSTSQ) operation with different libraries on GPU Server

Operation	Matrix Size	Time	Speedup vs Ref
Numpy MatMult	100x100	167 μ ± 874 ns	Ref.
Cupy MatMult	100x100	16.9 μ ± 107 ns	9.8x
Cupy MatMult + Data Copy	100x100	145 μ ± 736 ns	1.1x
Numpy MatMult	1000x1000	12.6 ms ± 40.6 μ	Ref.
Cupy MatMult	1000x1000	170 μ ± 435 ns	74.1x
Cupy MatMult + Data Copy	1000x1000	5.62 ms ± 158 μ	2.2x

Table 4.2: Matrix Multiplication (MatMult) operation with different libraries on GPU Server

The first test compares the time taken by the least squares function provided in three libraries, 'SciPy', used in MOGA, 'NumPy' and 'CuPy'. The results of the executions for the different matrix sizes are displayed in table 4.1.

The second test compares the time taken by the matrix multiplication operation between the 'NumPy' and 'CuPy' libraries. The results of the executions for the different matrix sizes are displayed in table 4.2.

Both tests were conducted in the GPU Server machine. The matrices used were generated using a function that randomly generates matrices of the desired size filled with random numbers in the interval $(0,1)$. The results shown in these experiments show promising results for replacing operations such as Least Squares in the algorithm directly with their CuPy counterparts. Another notable result is that, although it takes some more time, it is still faster, in all cases to copy data to the device, perform the operation, and back copy the result back to the host. This can be attributed to the extremely fast transfer rate and bandwidth of the hardware.

The activation and jacobian functions were the only ones that had to be custom-made. Both these functions were previously programmed in C due to their extensive use in the algorithm. The jacobian was successfully changed by altering how the operations were performed to the matrix. Instead of performing the operations iteratively throughout the matrix, now all the rows would be used to perform the calculations in parallel. The resulting changes allowed for a 79x speedup over the CPU implementation. The activation function would suffer a similar change, however it wasn't able to obtain better results than the CPU implementation. A kernel function implementation was attempted without success. This resulted in the use of the CPU version of the activation function in the final implementation. Due to the previous tests it was considered that even with the overhead of transferring the data back to the host, it would still perform faster than the CPU exclusive implementation.

4.5.3 Implementation

The initial implementation consisted of creating a separate library that would mirror the functions used in the training. The model's configuration would be pulled from the database in the

same way. This configuration would then be used to build the RBFANN python object that would then be initialized. The clustering, performed in the initialization would still be done in the CPU. After the clustering, the relevant data would be stored in a 'CuPy' array stored in the GPU's memory. The objective of keeping the data in the GPU during most of the execution is to avoid the overhead of transferring data back and forth as much as possible as most of the training operations can be performed exclusively on the GPU via 'CuPy'. The Levenberg Marquardt algorithm would use the GPU implementation of the jacobian function. Other operations within it, especially the linear algebra ones would use their 'CuPy' variants, such as *solve*, that solves a linear matrix equation, or system of linear scalar equations; *dot*, that produces the dot product of two arrays; *eigvalsh*, that computes the eigenvalues of a complex Hermitian or real symmetric matrix; *lstsq*, that computes the least-squares solution to a linear matrix equation. After the execution of the Levenberg Marquard algorithm data relative to the training, testing and validation root-mean-square-errors, as well as the centers, weights and spreads are transferred back to the host and stored in a Python dictionary that temporarily saves the data that will be stored in the database. If the problem is a regression problem, at this point in the algorithm the prediction would be calculated. Since it relies heavily on the activation function that is executed on the CPU, the relevant data is transferred back to the host and the prediction function called. The predictions would then be iteratively performed on the data that was set as a prediction variable in the problem's configuration.

After debugging, the implementation was tested with a single process with promising results, although slow. The following test would then use a number of processes, similar to what would've been used in the CPU cluster version, but still utilizing the GPU's capabilities. This test stopped progressing after a certain point, where, after investigating using the 'nvidia-smi' command to query the GPU's status, it was concluded that the processes were attempting to use more GPU memory than it was possible; this stopped all the processes from advancing. This raised the question of how to manage the number of processes depending on the problem's configuration, as the dataset used and configuration could change the amount of memory needed. The question is addressed in the next subsection.

4.5.4 Process Load Balancing

The CPU exclusive version of the MOGA platform would allow as many processes as the number of cores the CPU had. This is not feasible for the GPU implementation, where if we launched as many processes as the number of cores in the machine we would end up with the previously mentioned problem of trying to use more memory than the GPU allows. The initial approach to the problem was trying to check for memory leaks and free up memory that wasn't being used. After a revision of the code and some experimentation with a memory profiler, it was discovered that a large part of the memory use could be attributed to the CUDA context. This context holds the mapping of the allocated memory, loaded models and the management

data to control the device. It is initialized in each process, the first time a CUDA related library is called, in this case 'CuPy'. The memory amount used by the CUDA context is related to the architecture of the GPU.

An alternate approach was employed to solve this problem. It was decided that a *test run* would be performed before running the MOGA algorithm. This *test run* launched only one process and measured, throughout the training how much of the GPU memory was being used by the process. The measuring was performed by another process that was repeatedly calling the 'nvidia-smi' utility to assert how much memory was being used and keeping the maximum value throughout the model training. After the model was trained, the maximum value of memory used during training would be returned and a simple calculation would be performed for each GPU:

$$n_proc = \left\lfloor \frac{0.95 * gpu_memory}{max_memory} \right\rfloor \tag{4.1}$$

Where n_proc is the number of resulting processes, gpu_memory is the total memory available on the GPU and max_memory is the maximum recorded memory value of the *test run*. With the number of processes specified for each GPU, that number of processes would then be launched and associated with its specific GPU.

After testing, the memory values were now within the limits of the GPU's memory however the algorithm would not progress. This led to questioning if the problem was the amount of processes running simultaneously. In most of the literature, as well as NVIDIA's forums dedicated to CUDA, the recommended way to use GPUs for processing is to utilize one process per GPU to tackle a large task. The optimal scenario, according to this paradigm, would be a large enough problem that would benefit from parallel computing that would force the GPU to utilize the vast majority of its resources to solve it, minimizing the amount of time related with the transfers between host and device. There exists, however, a functionality called Multi-Process Service (MPS).

"The Multi-Process Service (MPS) is an alternative, binary-compatible implementation of the CUDA Application Programming Interface (API). The MPS runtime architecture is designed to transparently enable co-operative multi-process CUDA applications, typically MPI jobs, to utilize Hyper-Q capabilities on the latest NVIDIA (Kepler and later) GPUs. Hyper-Q allows CUDA kernels to be processed concurrently on the same GPU."[36] It allows the kernel and memory copy operations from different processes to be performed simultaneously, allowing for higher utilization rates in cases where smaller processing problems are presented that would not be able to fully occupy the GPU's resources. This functionality is only available in Linux based operating systems and for GPUs with a compute capability higher than 3.0, which the NVIDIA

A100 fulfills. To enable it, the following commands must be issued via terminal.

Listing 4.2: Enabling MPS

```
nvidia-smi -i 0 -c EXCLUSIVE_PROCESS
nvidia-smi -i 1 -c EXCLUSIVE_PROCESS
nvidia-cuda-mps-control -d
```

These commands, change GPU 0 and GPU 1 to 'EXCLUSIVE_PROCESS' mode, which would allow it to only assigned on process at a time. The last command enables the MPS server control daemon, which causes the 'EXCLUSIVE_PROCESS' mode to behave in a way that allows multiple clients to use the GPU via the MPS server.

After further testing and debugging, it was deemed that the GPU implementation for MOGA could be considered functional and the collection of results data could be initiated. These results will be presented in the next chapters.

4.5.5 Mini MOGA

As previously mentioned, the estimation of the time a problem would take based on its configuration is a problem for the researchers using the MOGA platform. To help solve this issue, after the implementation of the GPU code was complete, a GUI was created that would run an algorithm similar to the one already implemented. This GUI would mimic the one used to connect and interact with the platform. It would not have the concept of user but, aside from it the problem creation and overall structure would be similar. The idea behind it, is that it would exist in the GPU Server or some other machine that would be used as a worker in MOGA and execute the training of a number of models. This training would mimic the one used in the MOGA platform and would time it. Based on the problem configuration, such as number of generations and individuals per generation an upper bound estimate can be found for the total time it would take.

It functions for both CPU and GPU, whereas for CPU it allows the expected number of cores to be input in the interface, the GPU one performs a process similar to the one detailed in the Load Balancing section 4.5.4 and can estimate how processes can run in simultaneously.

$$time_est = \frac{max_model_time * n_gens * n_indivs * n_inits}{n_cores} \tag{4.2}$$

The upper bound estimate can be given by the maximum time it took to train the model multiplied by the number of generations, the individuals per generation and the number of model initializations, divided by the number of cores. For a given chromosome, the models are ini-

tialized and trained a number of times set by the user, where the best model is then saved. It is worth noting that some inaccuracy may exist due to some mechanisms present in MOGA, observable in figure 4.6. It is also worth noting that since this GUI will run the training locally, the results will be based on the specifications of the machine used.

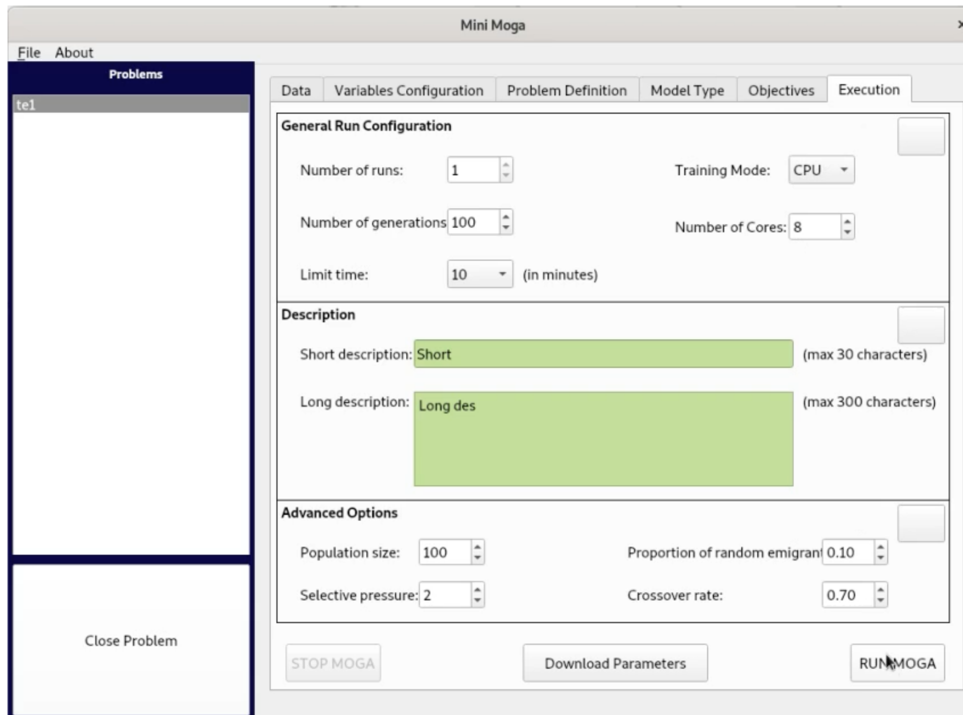


Figure 4.13: Mini Moga GUI use - CPU

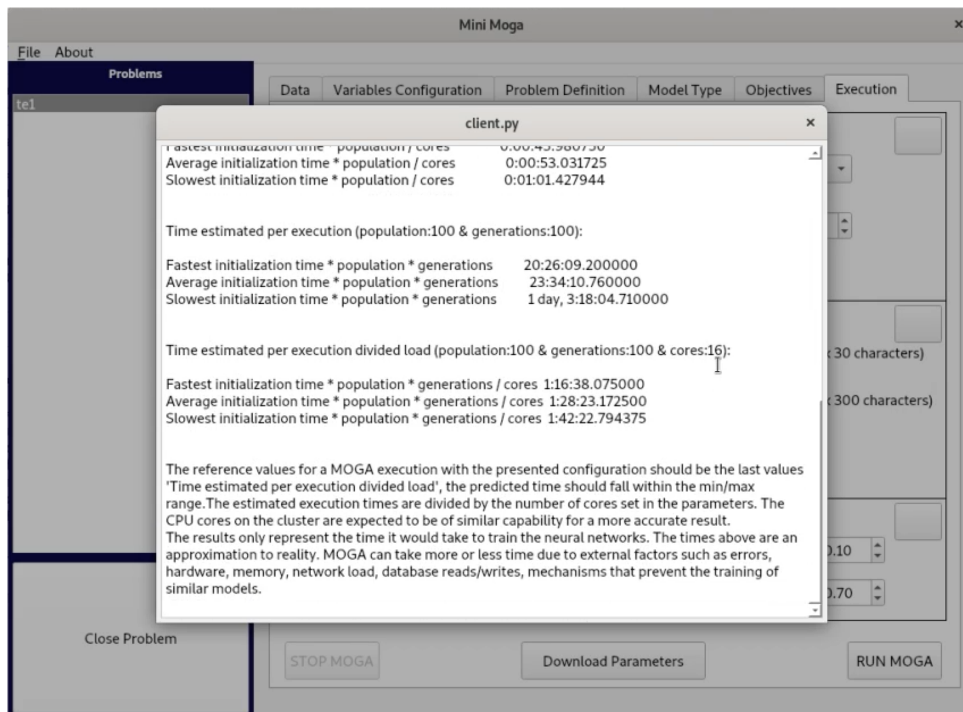


Figure 4.14: Mini Moga GUI results - CPU

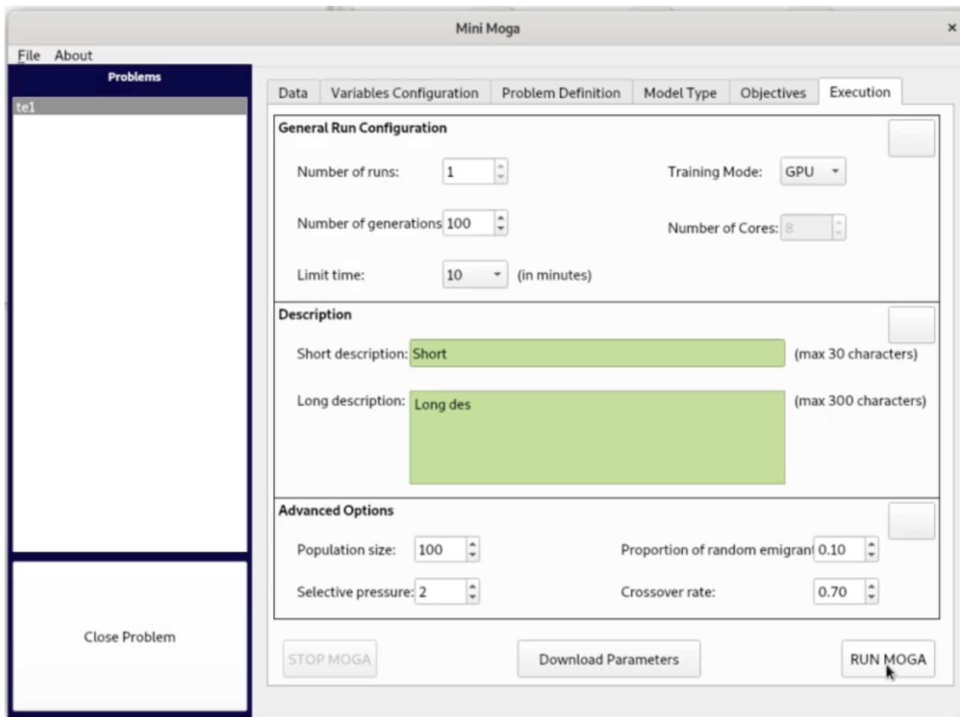


Figure 4.15: Mini Moga GUI use - GPU

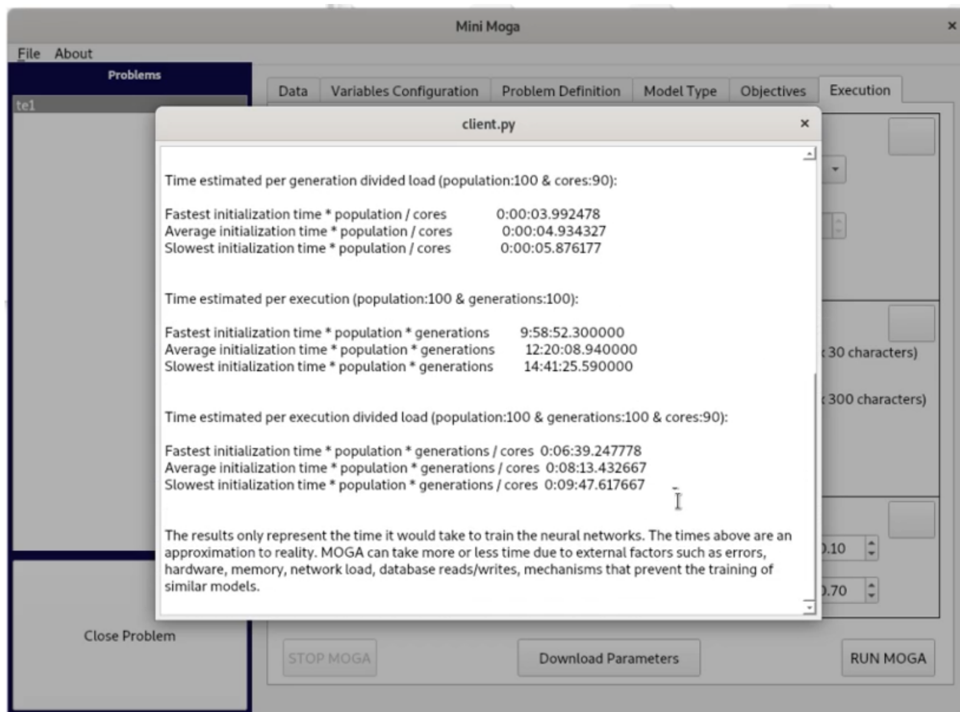


Figure 4.16: Mini Moga GUI results - GPU

Chapter 5

Evaluation

5.1 Validation of Results

This chapter deals with the validation of the GPU implementation from a statistical standpoint. It will provide an analysis of the results produced by the legacy CPU cluster and the newly implemented GPU one with the objective of verifying if there is a significant statistical coherence of the results.

A testing dataset called 'atgambelas' that had been previously used in the MOGA platform for research was used. For this experiment, the problem was executed in the legacy CPU cluster and in the GPU one. Both executions were performed with the exact same configuration and were let run until the maximum number of generations was reached. After execution, sets of files containing relevant model data can be downloaded from the GUI. This data was analyzed in MATLAB to create the following plots.

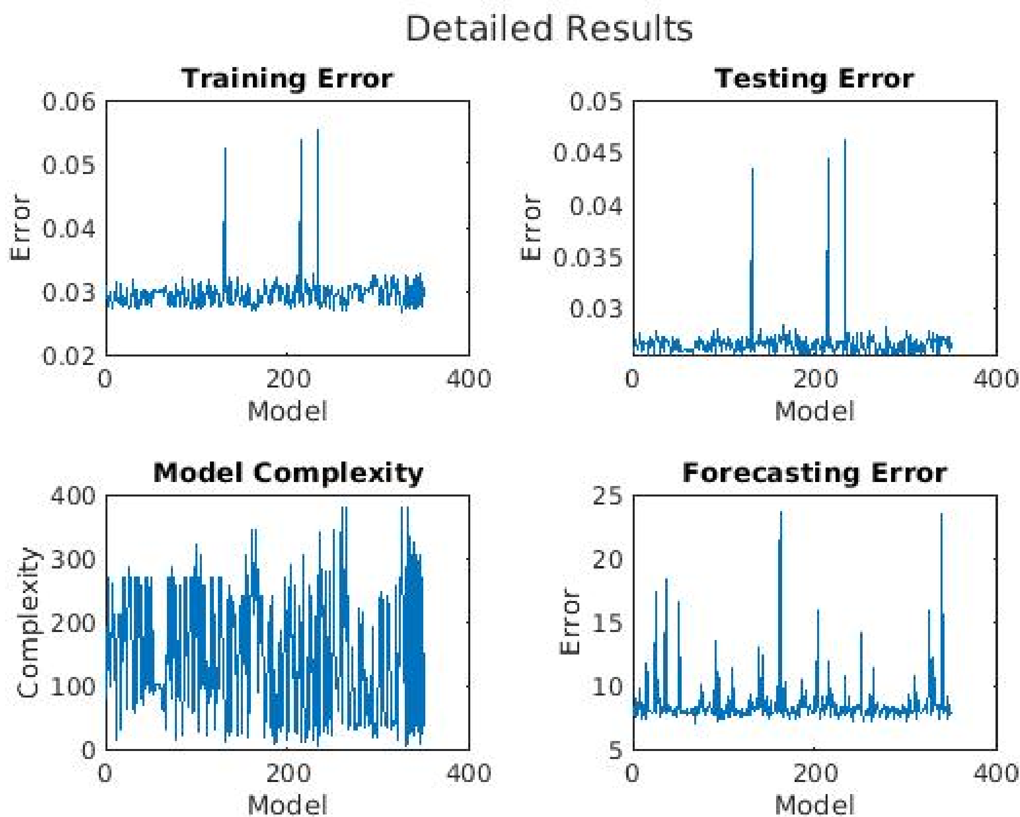


Figure 5.1: Overview of errors (CPU)

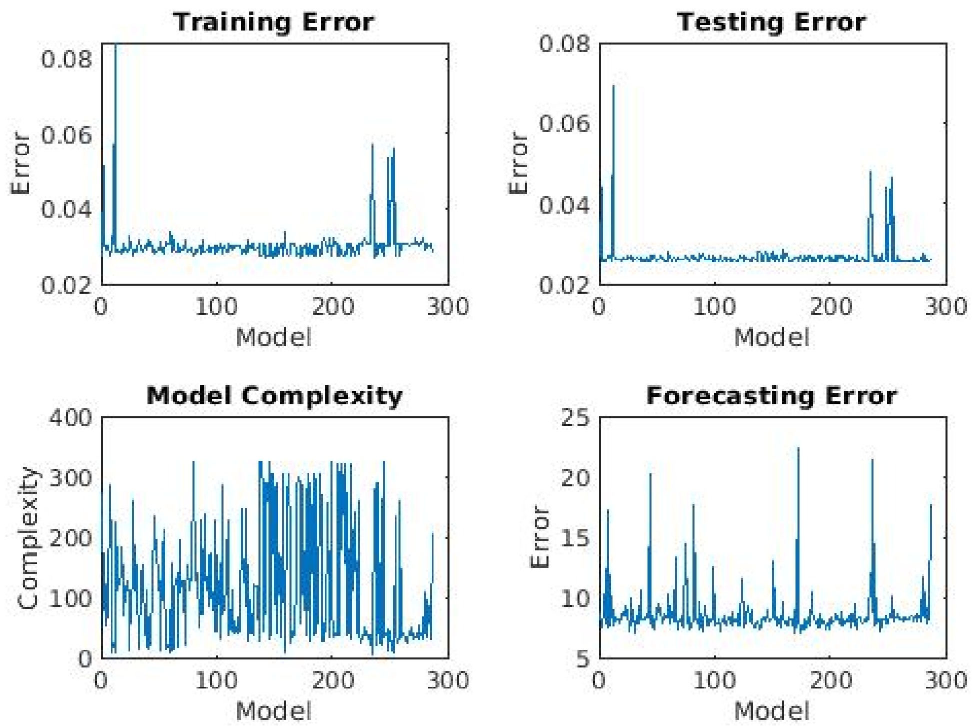


Figure 5.2: Overview of errors (GPU)

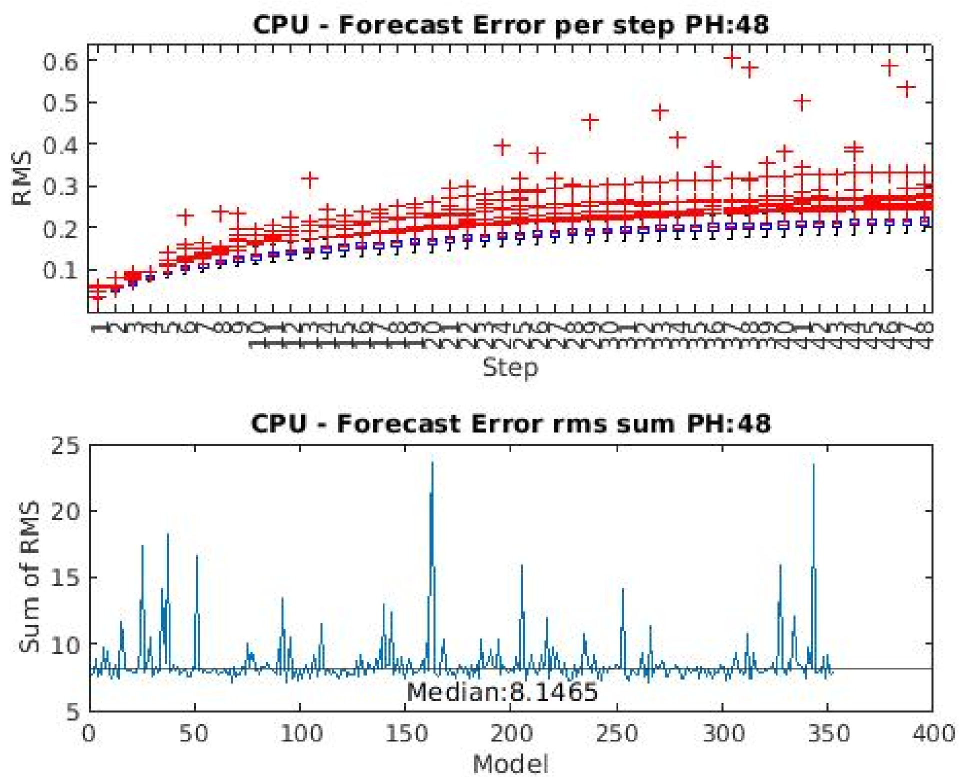


Figure 5.3: Forecast Error (CPU)

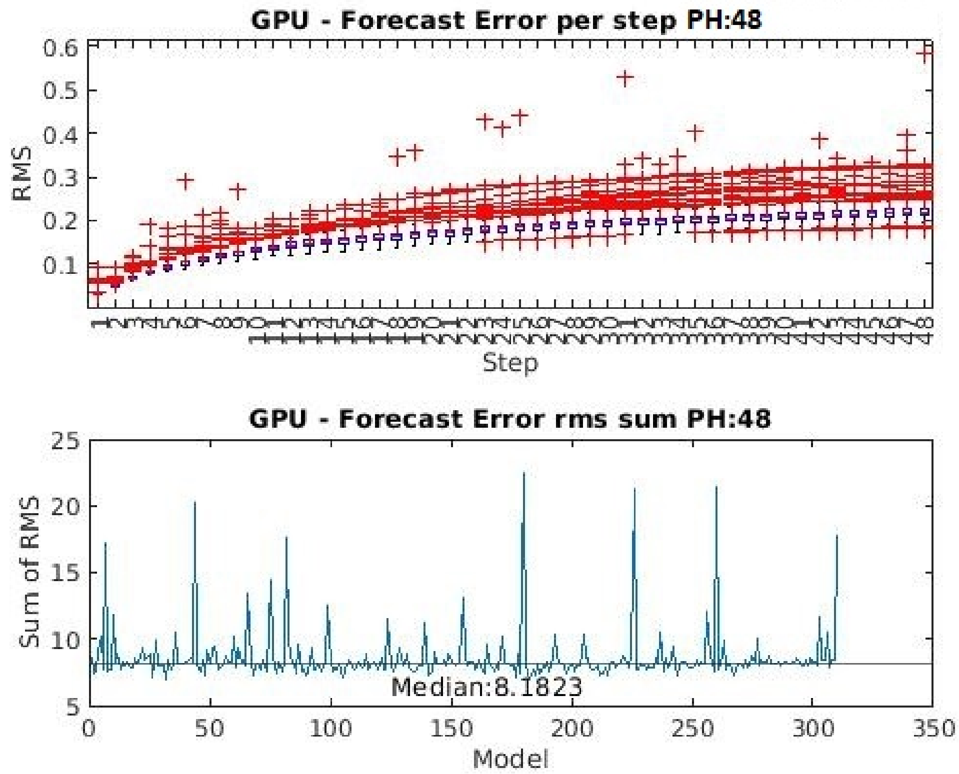


Figure 5.4: Forecast Error (GPU)

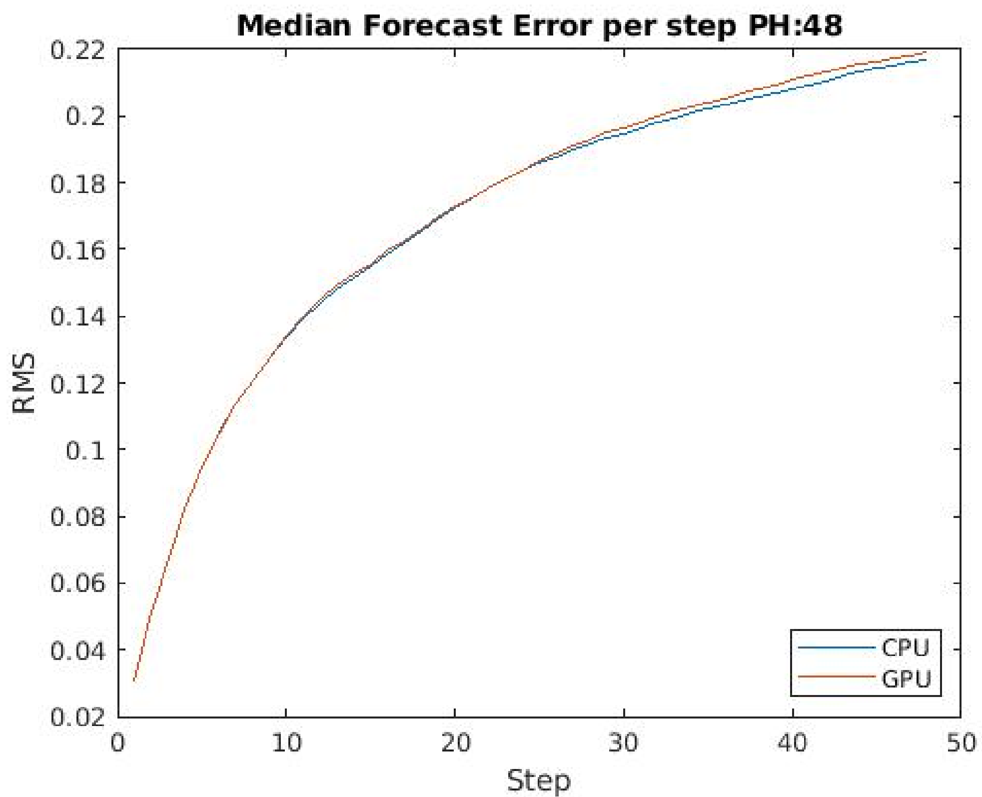


Figure 5.5: CPU vs GPU median error per step PH:48

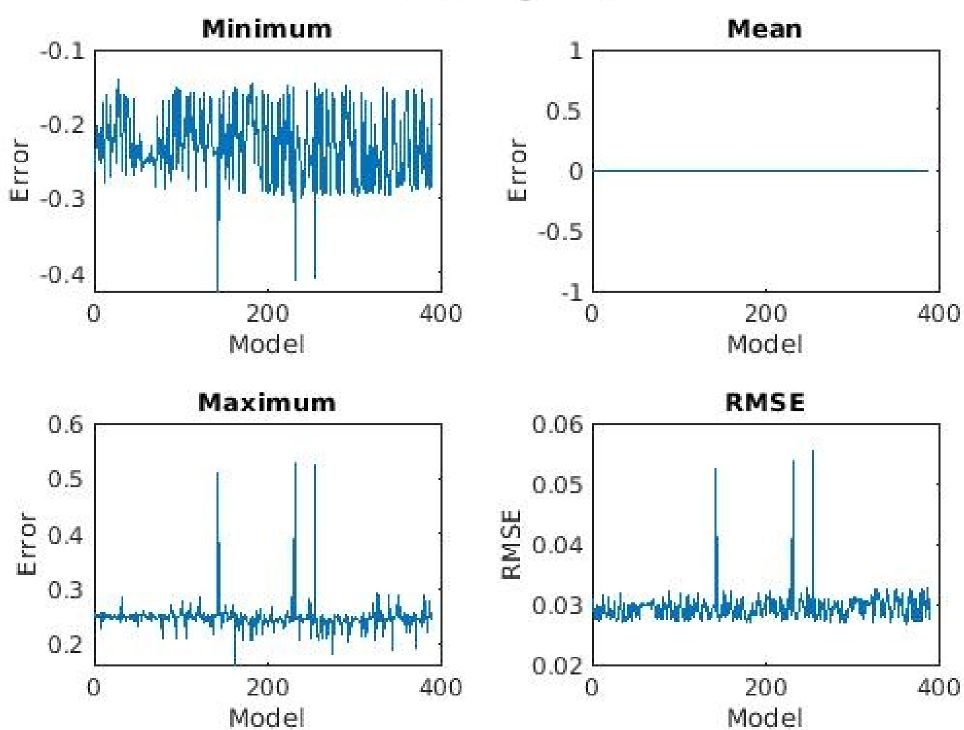


Figure 5.6: Training Error (CPU)

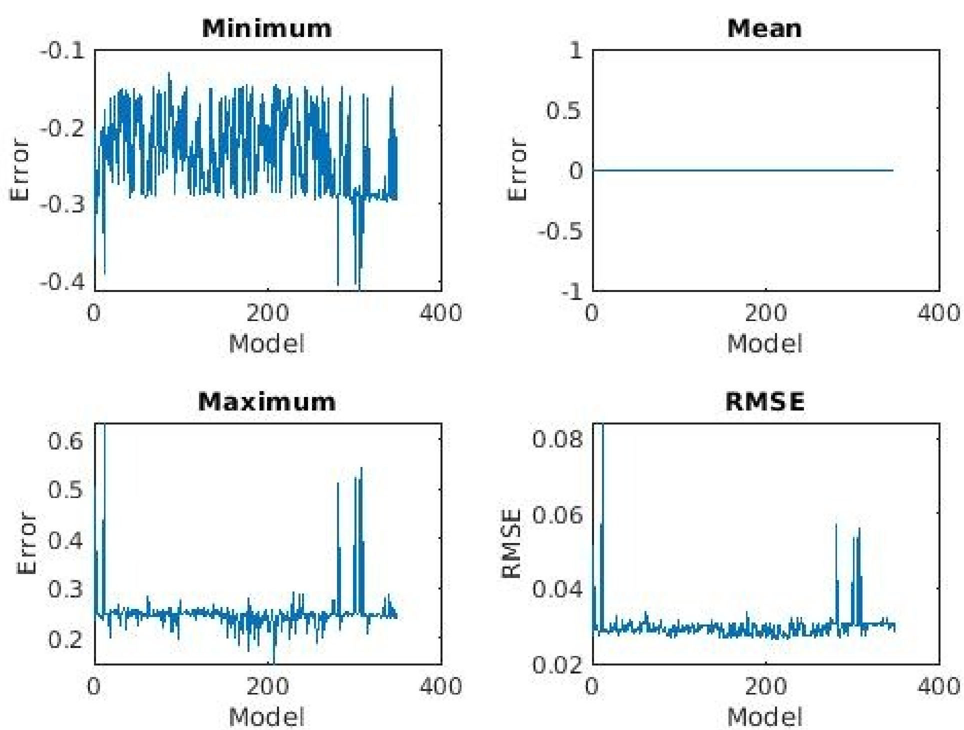


Figure 5.7: Training Error (GPU)

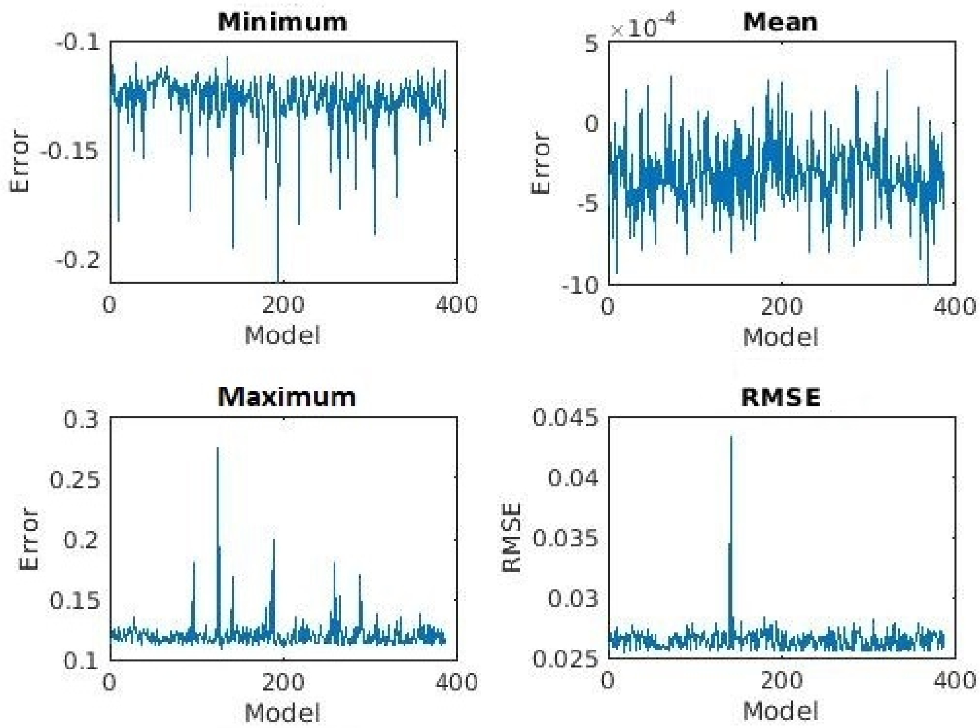


Figure 5.8: Testing Error (CPU)

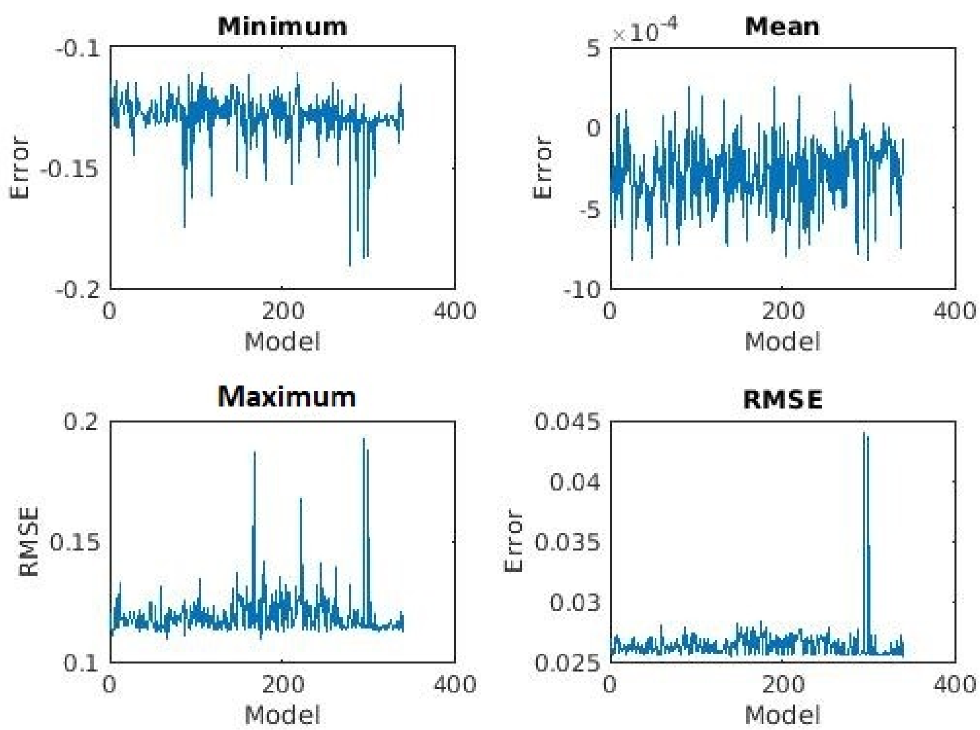


Figure 5.9: Testing Error (GPU)

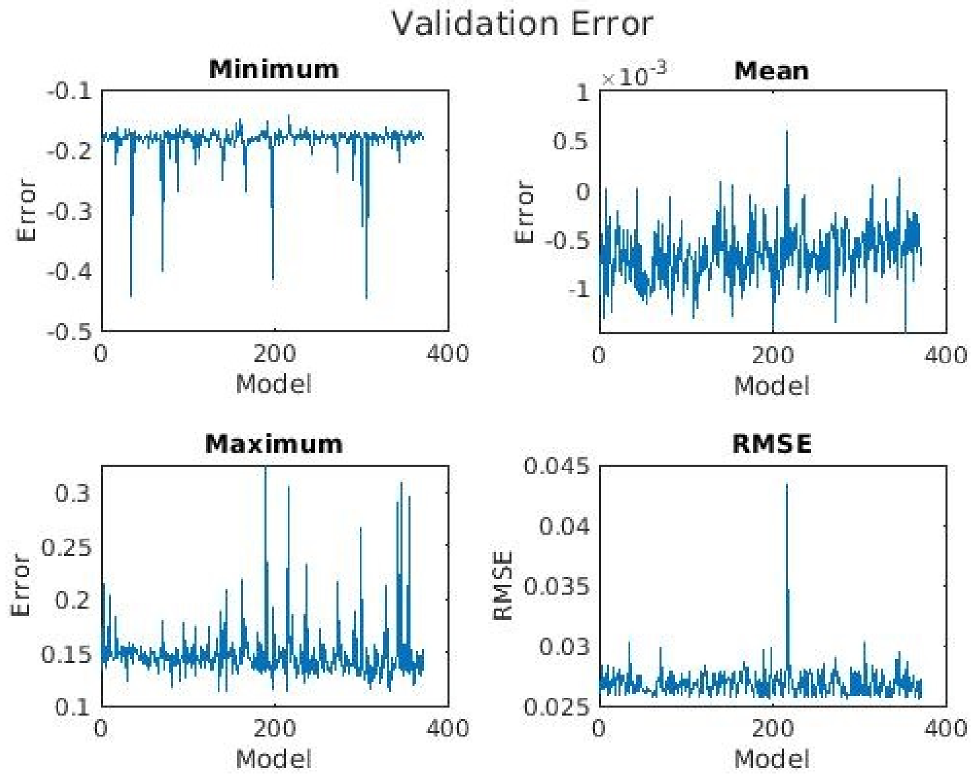


Figure 5.10: Validation Error (CPU)

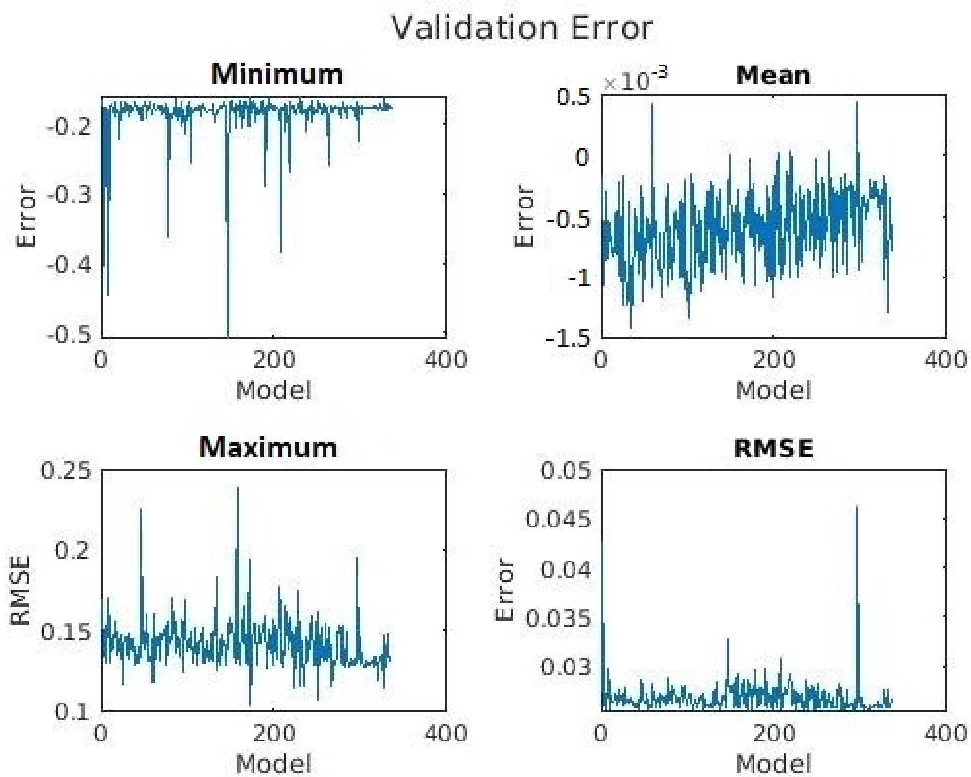


Figure 5.11: Validation Error (GPU)

Medians	CPU	GPU
Training Error	0.0293	0.0293
Testing Error	0.0264	0.0262
Validation Error	0.0267	0.0265
Prediction Error	8.1465	8.1823
Model Complexity	128	112

Table 5.1: Median values CPU vs GPU

Some information worth noting before analyzing the plots is that models generated by both executions are inherently affected by randomness, from the different numbers of inputs and neurons; the different RBFANN intializations due to clustering; the mechanisms present in a genetic algorithm, such as selection procedures for mating and the mutation operator. It is thus reasonable to expect that two executions would produce a completely different number of models, and that models from different executions would have some level of difference between themselves. As to make the plots more readable, models that had values outside two times the median value were not included as they would often have values that would make the rest of the plot unreadable.

After extracting the data, 393 non-dominated models were obtained from the CPU execution and 352 models were obtained from the GPU execution. Figures 5.1 and 5.2 give an overview of the models, the Root Mean Square Error (RMSE) training and testing errors. As seen in table5.1 the median error values for training are 0.0293 for both CPU and GPU and the median error values for testing are 0.0264 for CPU and 0.0262 for GPU. The model complexity, given by the number of neurons times the number of inputs, are also within the same range, comprised between 0 and 300 for the majority of models, with median values of 128 for CPU and 112 for GPU; this would be expected as the configuration defining the minimum and maximum values for neurons and inputs is the same. The forecasting RMSE has a median value of 8.1465 for CPU and 8.1823 for GPU. These values consist of the sum of errors for each prediction step. Since they are close to each other it would mean that the errors for each step are also similar.

In figures 5.3 and 5.4, the top plot is a vertical boxplot containing the RMSE for all the models for each step within the prediction horizon (48). The red crosses represent outliers and were attributed automatically according to MATLAB's algorithm. In both instances the RMSE starts close to 0.01 for step one and goes towards 0.22 by step 48 while having a similar curve for both CPU and GPU. This can be better observed in figure 5.5, that shows the median values for each step for both CPU and GPU. The bottom plots contain the sum of the RMSE for every step of the prediction horizon for all the models, this plot is the same as the one on the bottom right in figures 5.1 and 5.2 but with more detail. As previously noted, the median values are close. There are also occasional spikes in both the CPU and GPU instances.

Figures 5.6 and 5.7 represent the training error for all models in CPU and GPU; figures 5.8 and 5.9 represent the testing error for all models in CPU and GPU; figures 5.11 and 5.10 represent the validation error for all models in CPU and GPU. Throughout these pairs of figures, the minimum, maximum, mean and RMSE are represented for every model. All the plot pairs show similar results between the CPU and GPU implementations. Aside from the occasion outliers and some slight median differences the results produced are very similar. This can be further verified by the median values in table 5.1, where the training, testing and validation median values correspond to the median values in the RMSE plots in the previously mentioned figures.

Although the values can differ slightly, these discrepancies could be attributed to the nature of the genetic algorithm. Considering how close the results produced by the new GPU implementation are to the legacy CPU results, there is reason to believe the implementation was a success.

Chapter 6

Results - Execution time

This chapter will present the results of the execution of 6 problems, 3 regression problems and 3 classification problems. All 6 problems were executed in both legacy CPU cluster with 5 workers, with the previously referenced hardware specifications, and in the newly implemented GPU platform. The execution times were measured using the loggers in the code that would mark the start and end of the execution, as well as the time it took each model to be trained. Each problem was executed in both machines with the same exact configuration. All the problems were set to have 5 model initializations and 50 iterations limit for the Levenberg Marquardt algorithm.

Table 6 gives an overview of the configurations of the problems tested. It provides the sizes of the data used for training, testing and validation; The ranges allowed for the neurons and inputs; The Prediction Horizon (PH), the timeframe for which the forecasting will be performed. The discussion of the following results will be done in the next chapter.

Problem	Type	Training	Testing	Validation	Neurons	Inputs	PH
dbc1	Classification	1773x199	591x199	591x199	[1:10]	[1:30]	N/A[1]
fd8	Classification	6906x21	2302x21	2302x21	[2:20]	[1:20]	N/A
sc17	Classification	2586x52	510x52	510x52	[2:30]	[1:30]	N/A
atgambelas	Regression	3789x39	1262x39	1262x39	[2:20]	[1:20]	48
kbotp9	Regression	6906x21	2302x21	2302x21	[1:10]	[1:20]	48
p1pcsr	Regression	10298x70	2302x70	3419x70	[2:10]	[1:30]	36

Table 6.1: All problem configurations

Problem	Time (CPU)	Time (GPU)	Speedup [2]
dbc1	01:00:28	00:16:27	3.68
fd8	1:28:19	00:32:52	2.69
sc17	02:28:25	00:32:53	4.51
atgambelas	02:03:41	00:46:55	2.64
kbotp9	07:11:55	02:42:24	2.66
p1pcsr	02:37:43	00:32:34	4.84

Table 6.2: All problem times CPU & GPU

[1] 'N/A' stands for 'Non Applicable'. In this context, a prediction horizon doesn't make sense for a classification problem.

[2] Formula: CPU/GPU

Problem	Type	Training	Testing	Validation	Neurons	Inputs
dbc1	Classification	1773x199	591x199	591x199	[1:10]	[1:30]

Table 6.3: dbc1 problem configuration

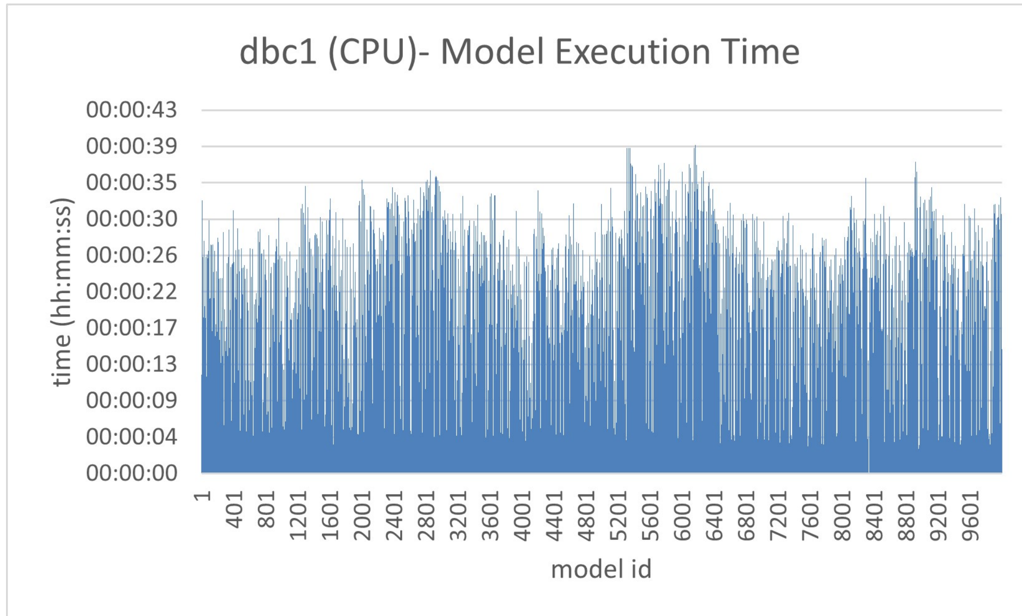


Figure 6.1: dbc1 (CPU) - Model Execution Time

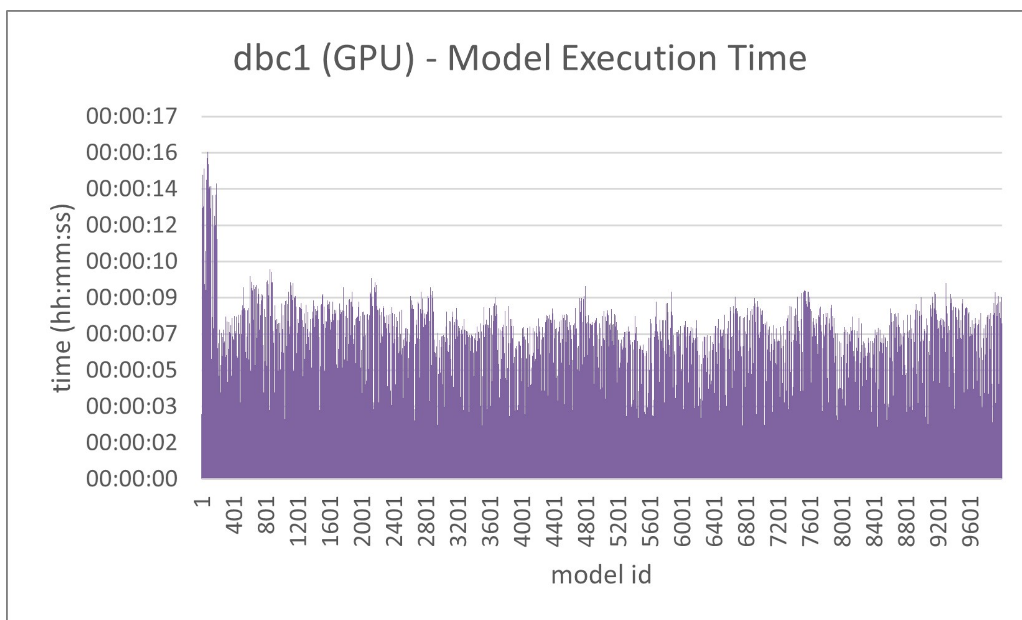


Figure 6.2: dbc1 (GPU) - Model Execution Time

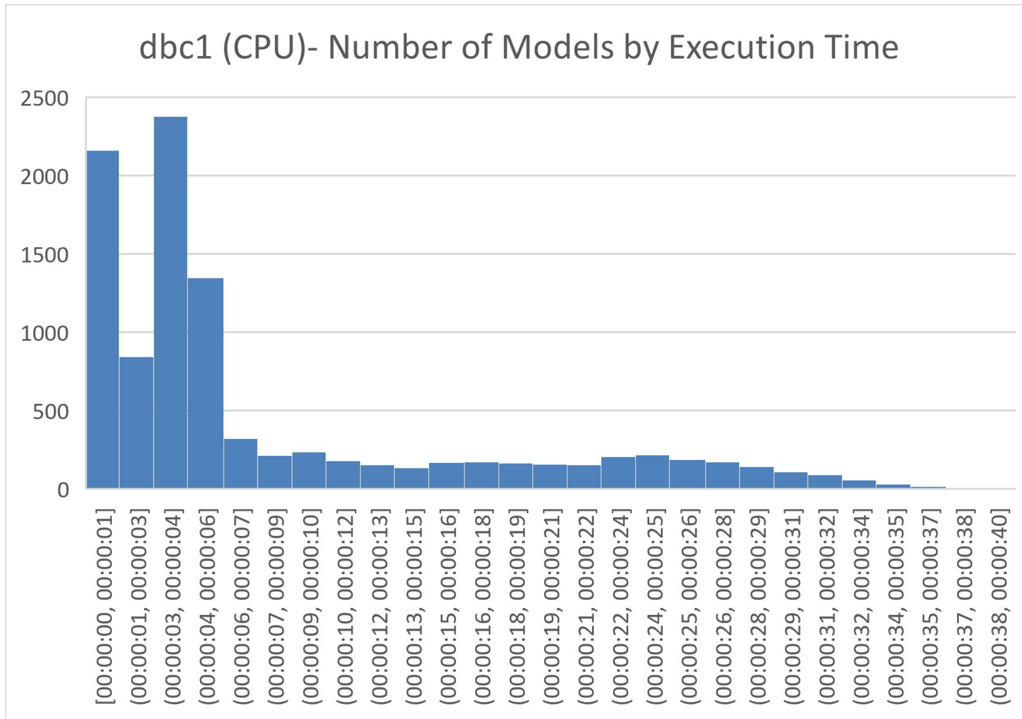


Figure 6.3: dbc1 (CPU) - Number of models by execution time

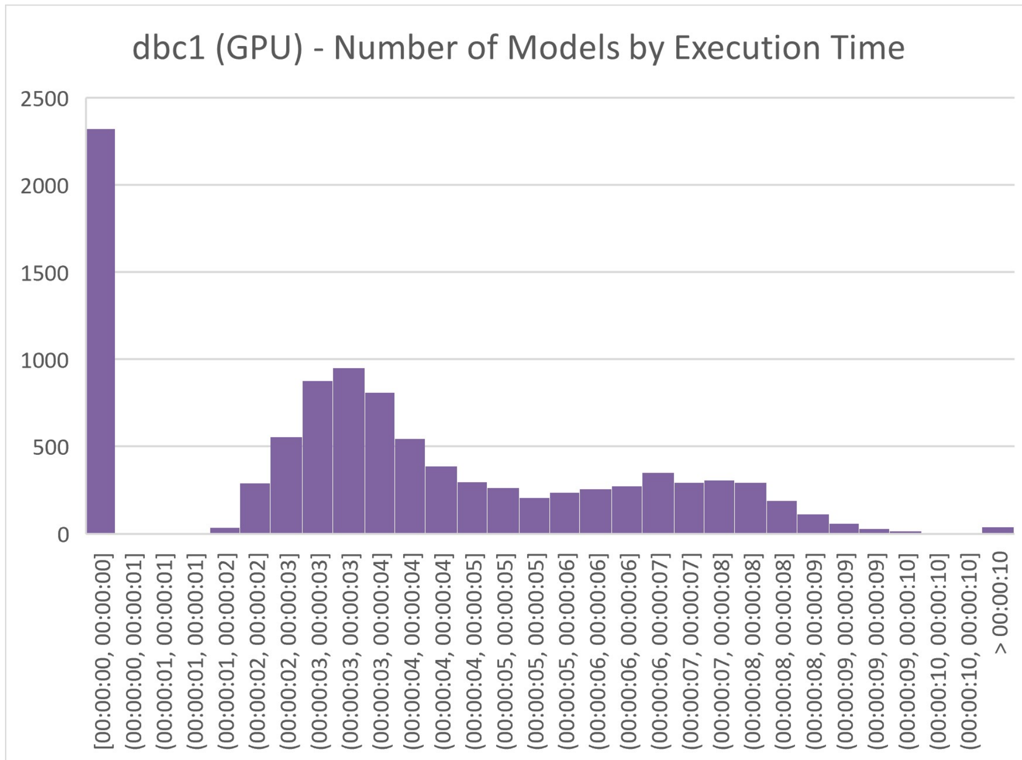


Figure 6.4: dbc1 (GPU) - Number of models by execution time

The GPU times spike early but then keep consistently lower than the CPU ones. Both histograms indicate numerous models under one second. It is also noticeable that the vast majority

of the GPU times are under 10 seconds, while the CPU ones have more tendency to go above this value.

Problem	Type	Training	Testing	Validation	Neurons	Inputs
fd8	Classification	6906x21	2302x21	2302x21	[2:20]	[1:20]

Table 6.4: fd8 problem configuration

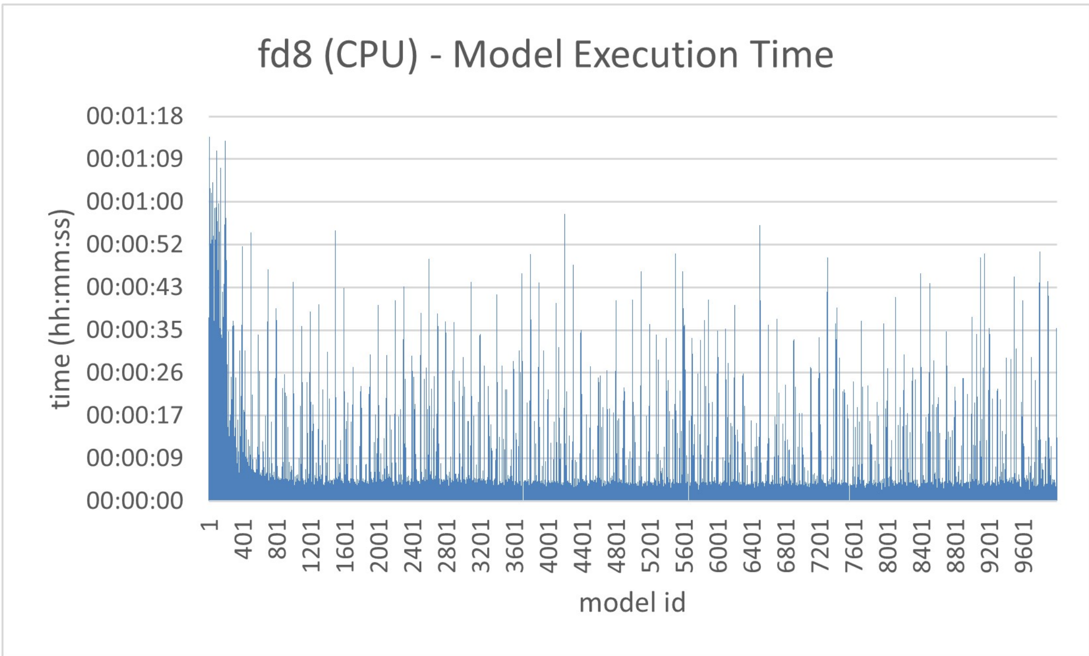


Figure 6.5: fd8 (CPU) - Model Execution Time

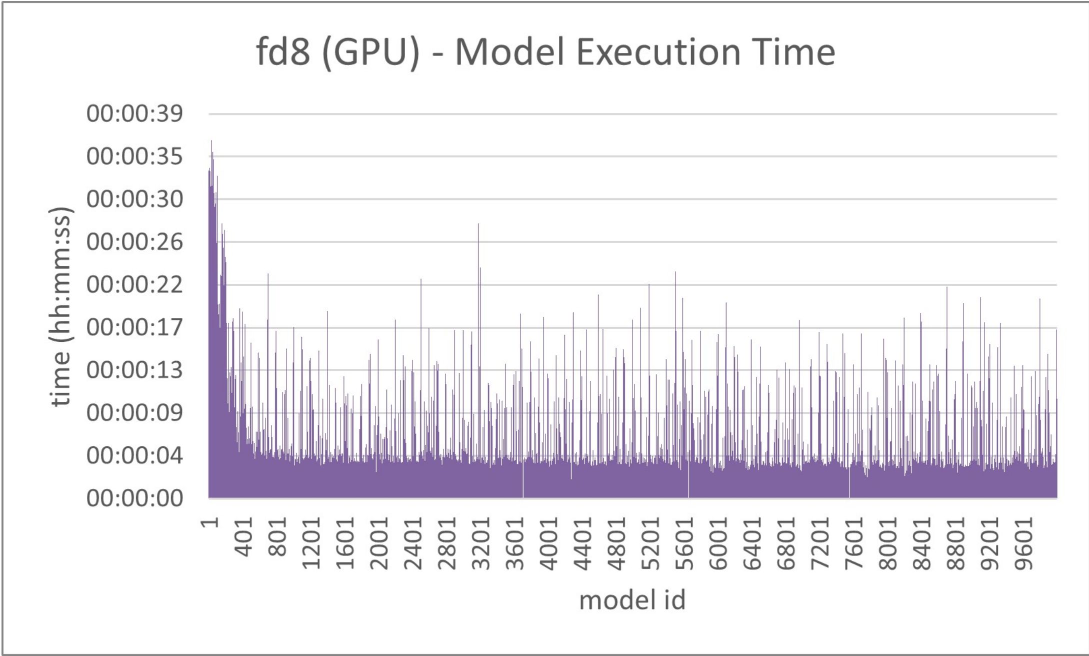


Figure 6.6: fd8 (GPU) - Model Execution Time

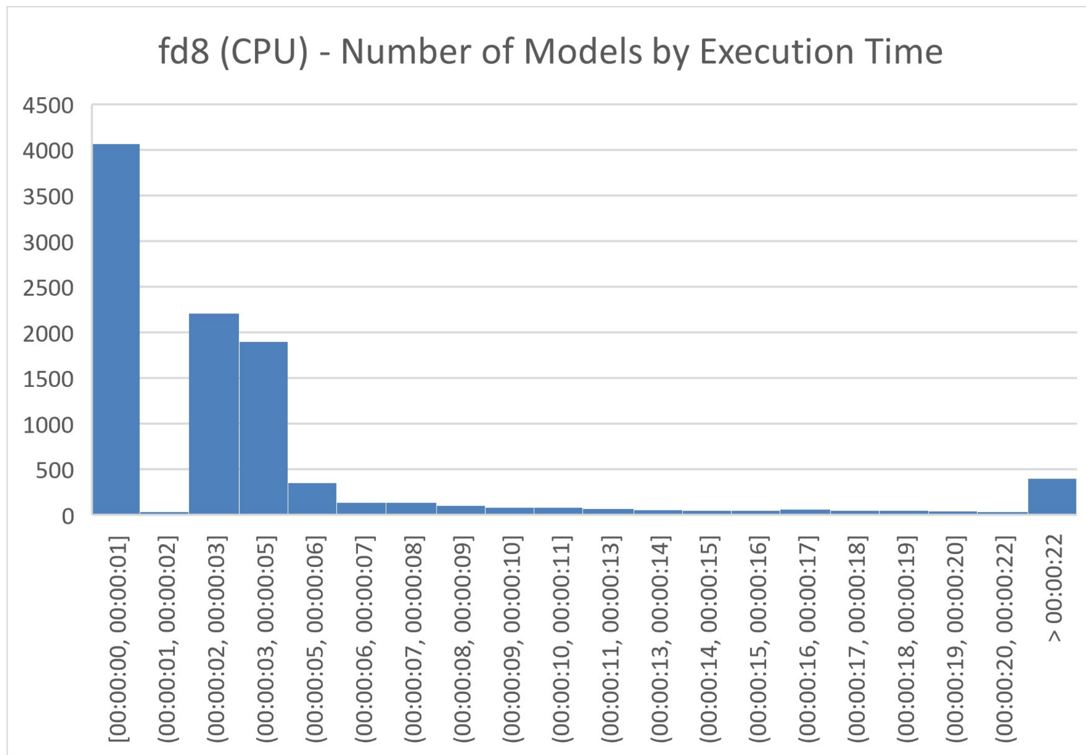


Figure 6.7: fd8 (CPU) - Number of models by execution time

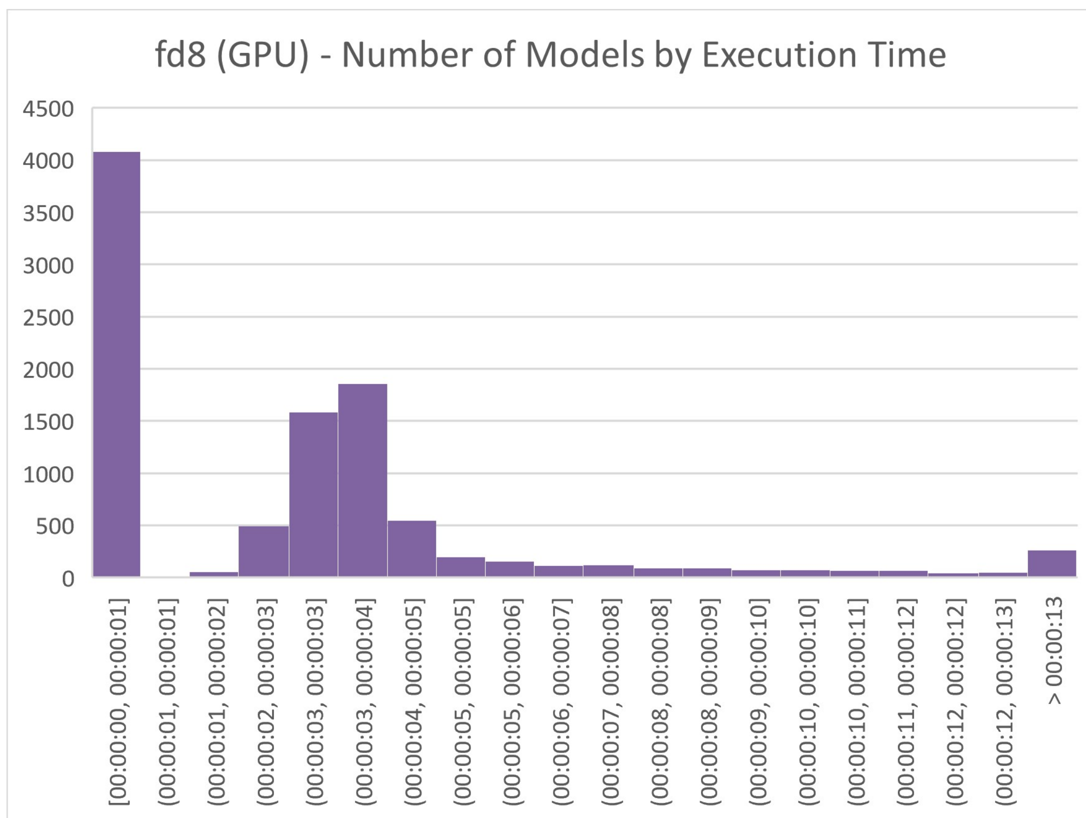


Figure 6.8: fd8 (GPU) - Number of models by execution time

Both time plots exhibit a similar pattern, starting with a large spike and then producing lower time values with occasional spikes. The histograms indicate numerous models that execute in under 1 second. They also indicate that both CPU and GPU tend to keep the time values between 2 and 5 seconds, with the CPU having outliers with larger time values.

Problem	Type	Training	Testing	Validation	Neurons	Inputs
sc17	Classification	2586x52	510x52	510x52	[2:30]	[1:30]

Table 6.5: sc17 problem configuration

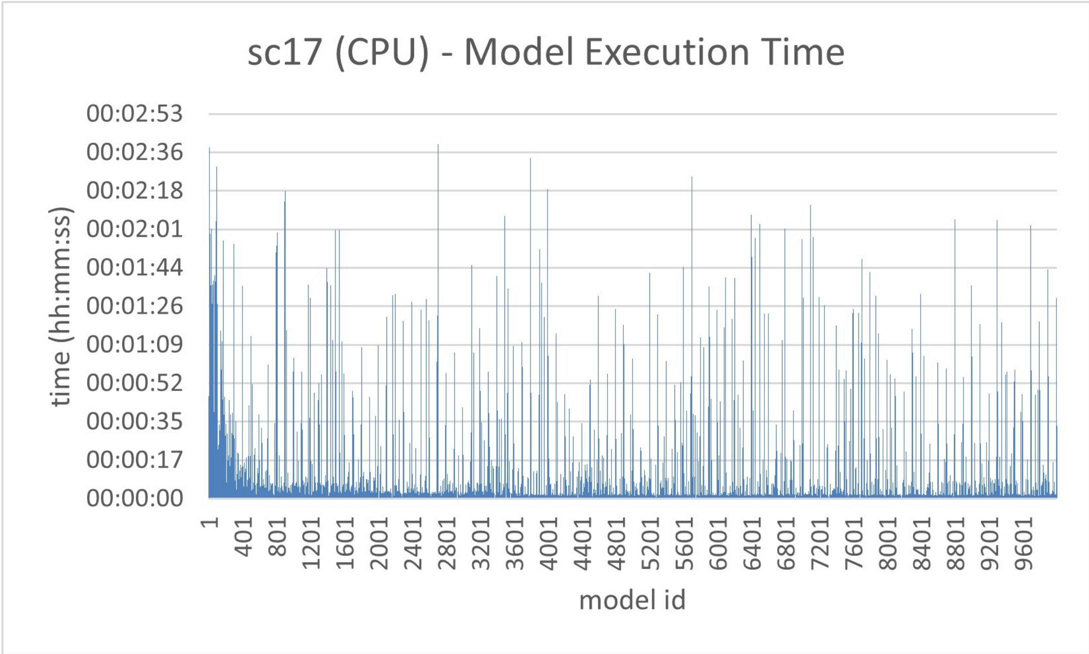


Figure 6.9: sc17 (CPU) - Model Execution Time

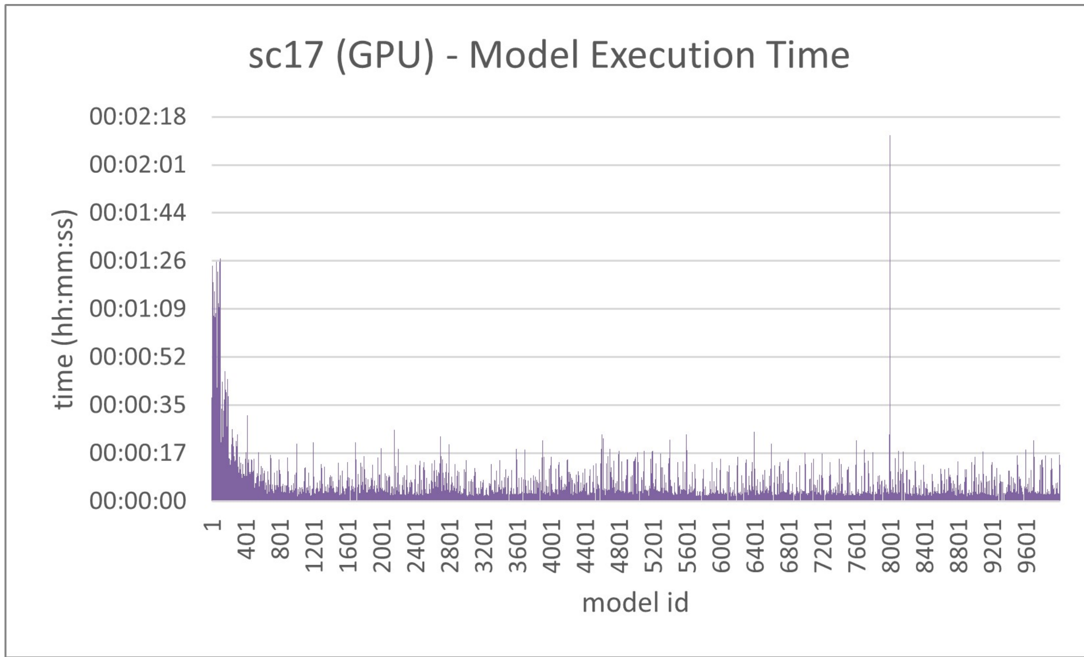


Figure 6.10: sc17 (GPU) - Model Execution Time

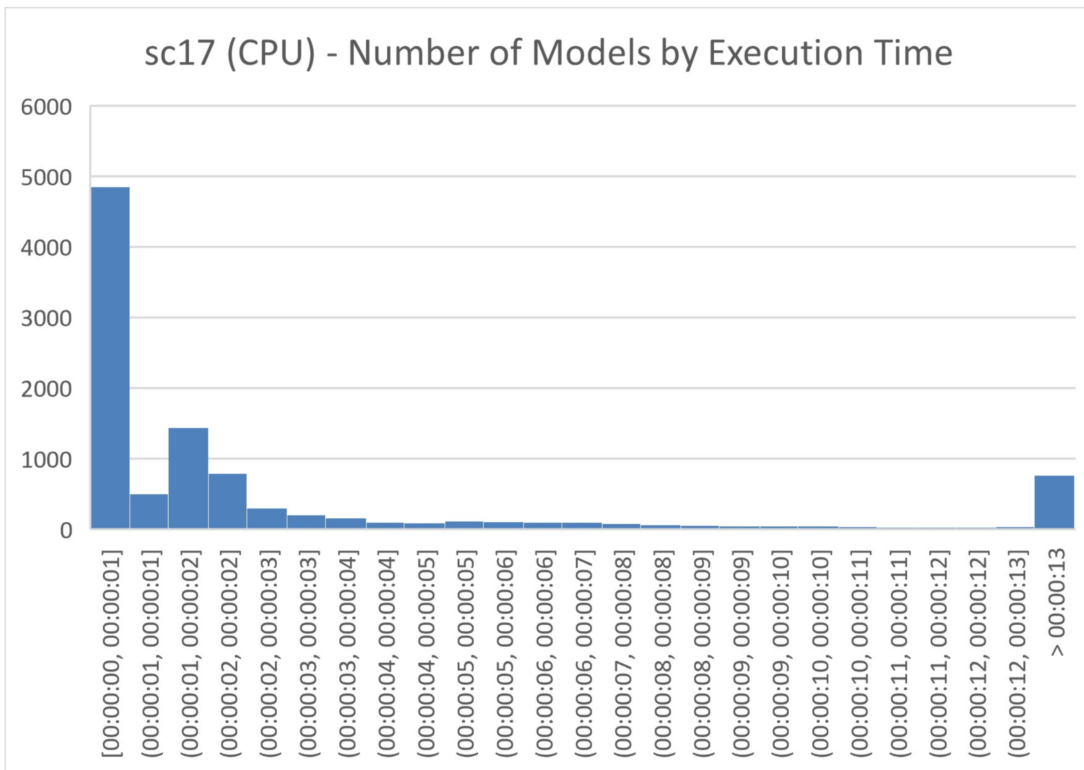


Figure 6.11: sc17 (CPU) - Number of models by execution time

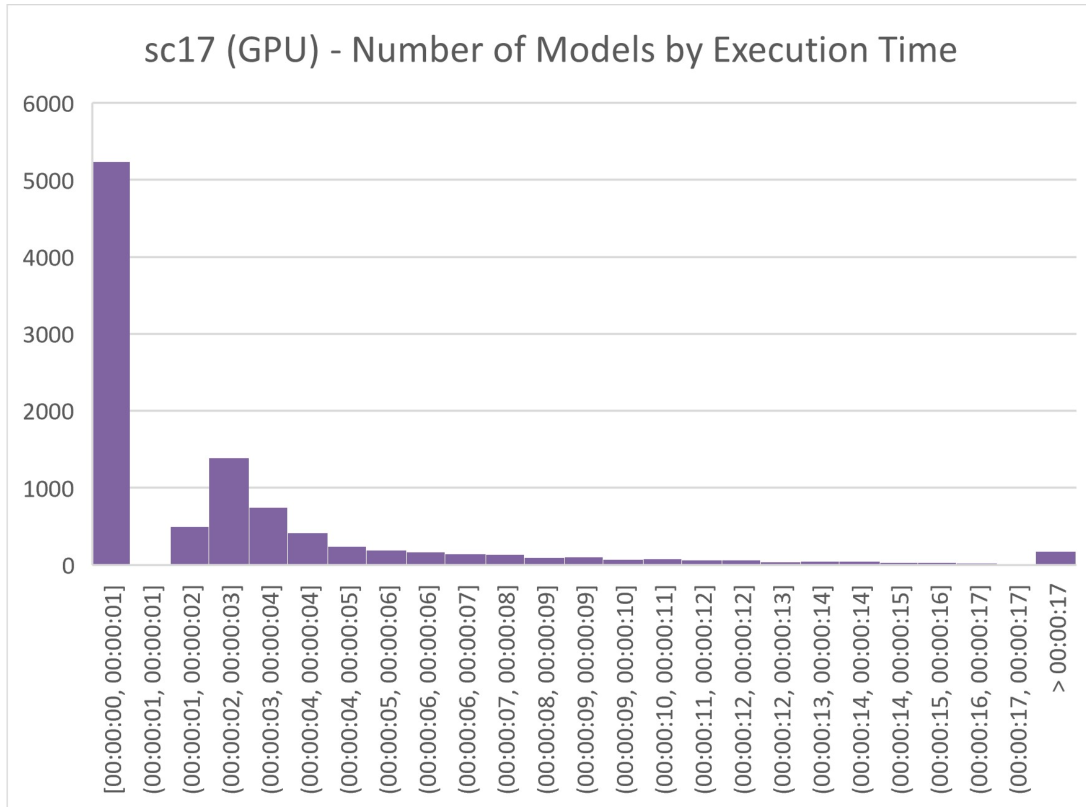


Figure 6.12: sc17 (GPU) - Number of models by execution time

Both plots indicate an early spike but then diverge. While the CPU starts having numerous spikes that exceed the 30 seconds mark, the GPU keeps to consistent times under 30 seconds, with an exception around model 8001, that exceeded 2 minutes. The histograms further reinforce this by showing consistent execution times for both between 1 and 4 seconds. However, the histogram also reveals that the CPU has a substantial number of outliers.

Problem	Type	Training	Testing	Validation	Neurons	Inputs	PH
atgambelas	Regression	3789x39	1262x39	1262x39	[2:20]	[1:20]	48

Table 6.6: atgambelas problem configuration

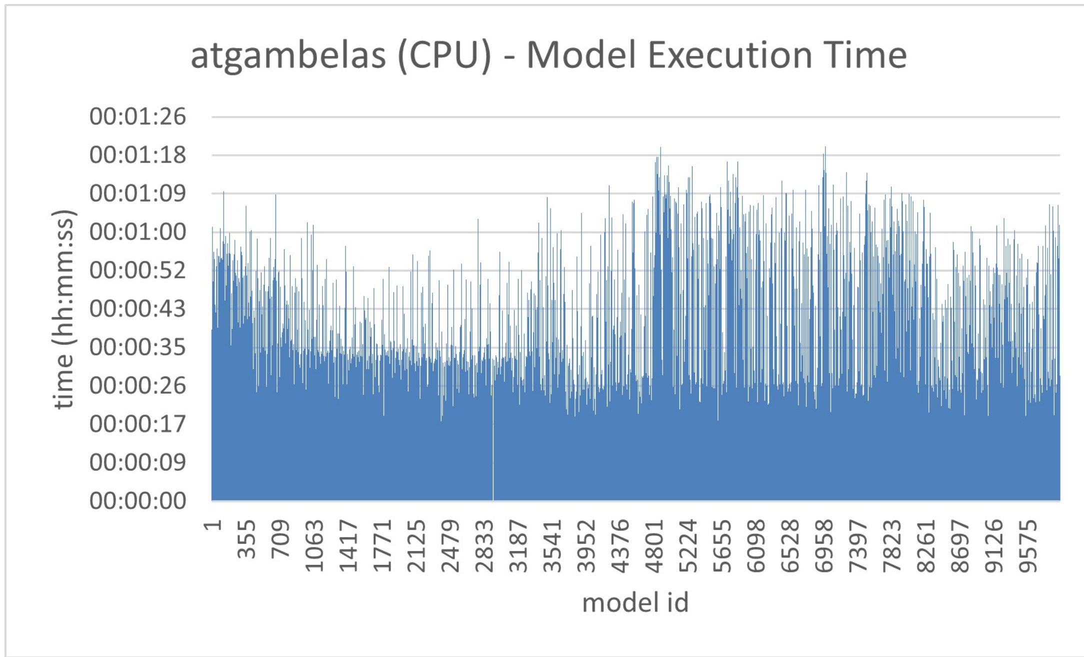


Figure 6.13: atgambelas (CPU) - Model Execution Time

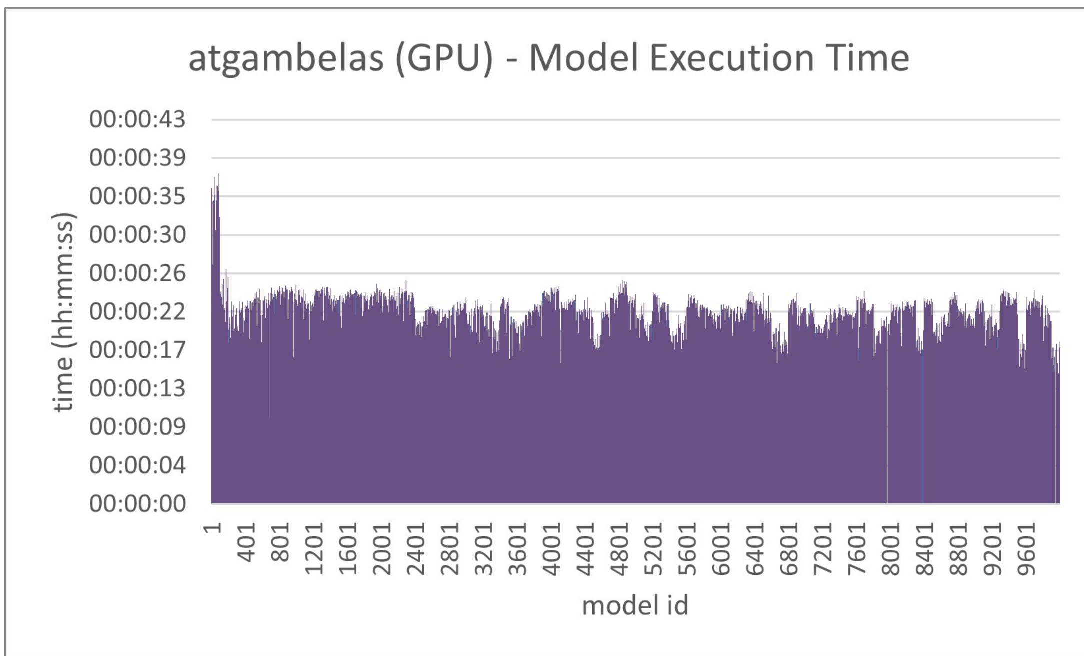


Figure 6.14: atgambelas (GPU) - Model Execution Time

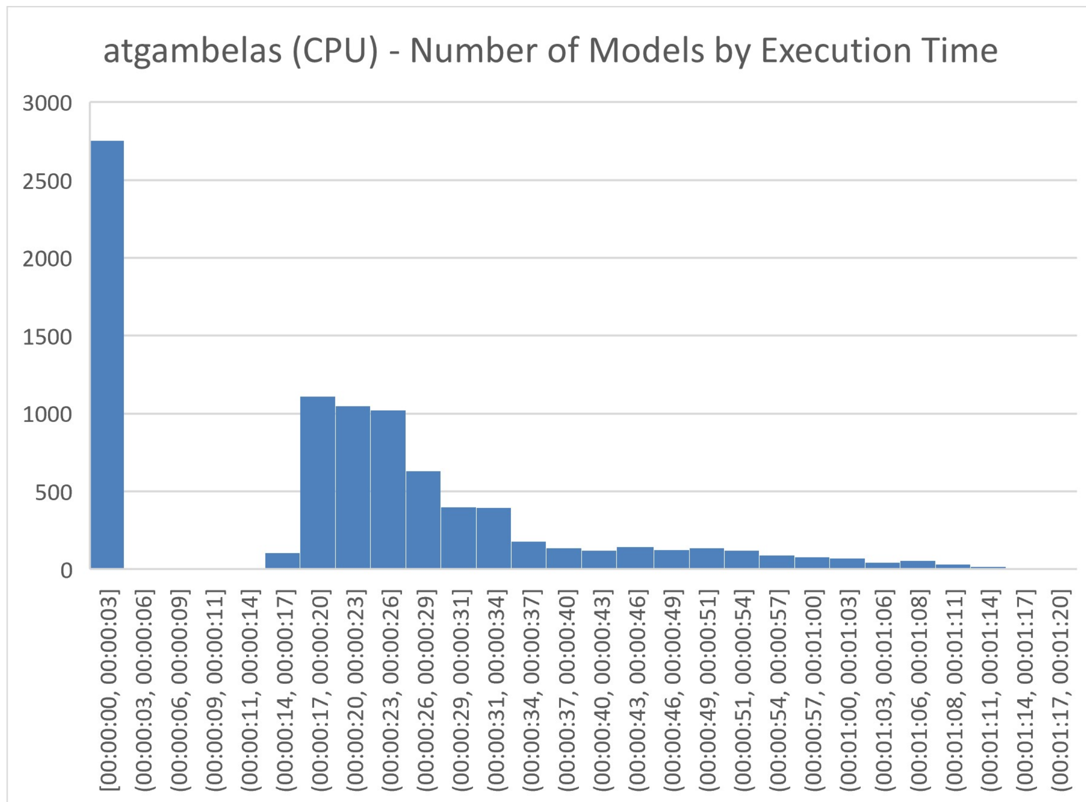


Figure 6.15: atgambelas (CPU) - Number of models by execution time

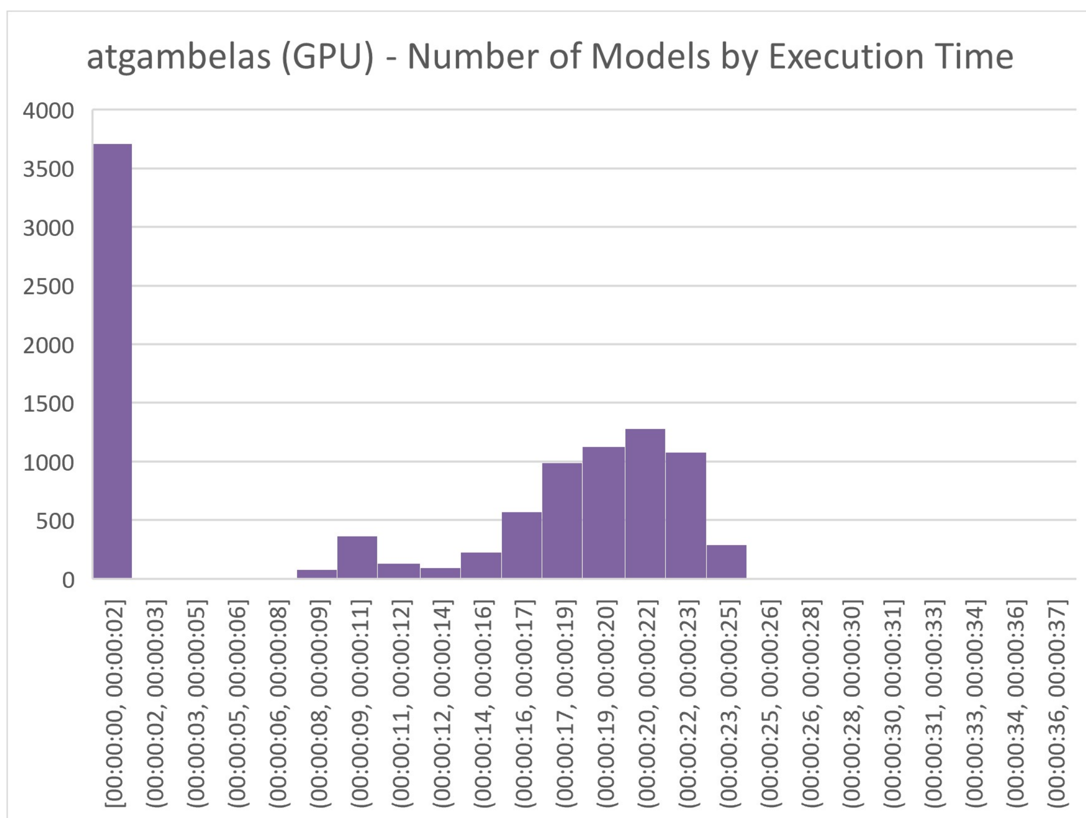


Figure 6.16: atgambelas (GPU) - Number of models by execution time

Both time plots show an initial spike. The CPU plot then lowers to execution times between 14 and 34 seconds with a substantial number of outliers. While the GPU plot consistently keeps times between 9 and 25 seconds. In both cases, numerous executions are below 1 second.

Problem	Type	Training	Testing	Validation	Neurons	Inputs	PH
kbotp9	Regression	6906x21	2302x21	2302x21	[1:10]	[1:20]	48

Table 6.7: kbotp9 problem configuration

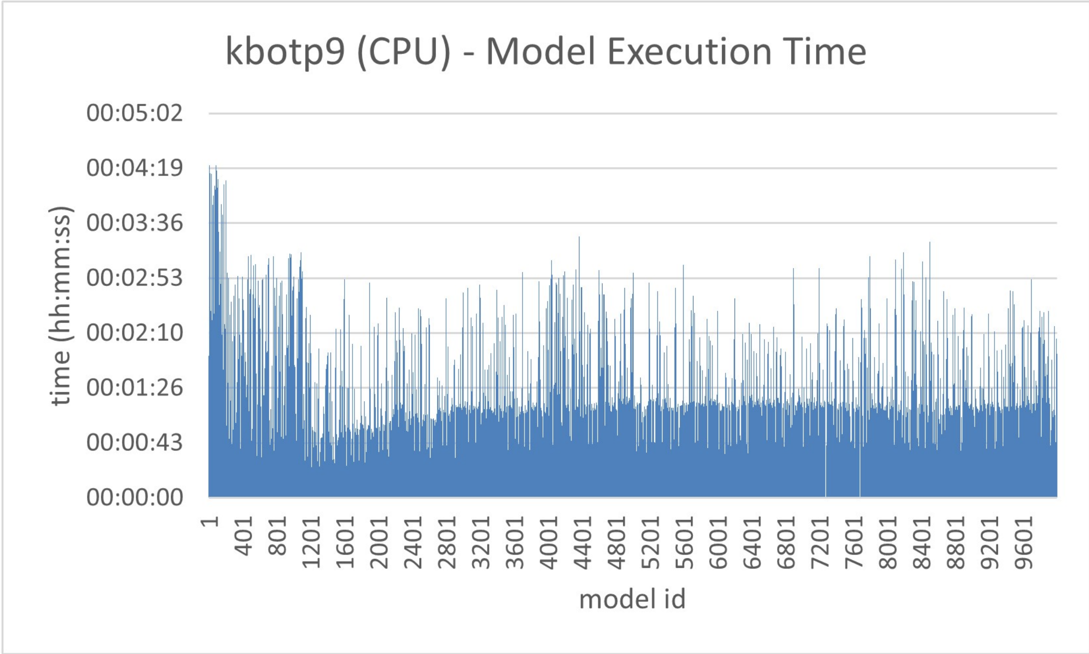


Figure 6.17: kbotp9 (CPU) - Model Execution Time

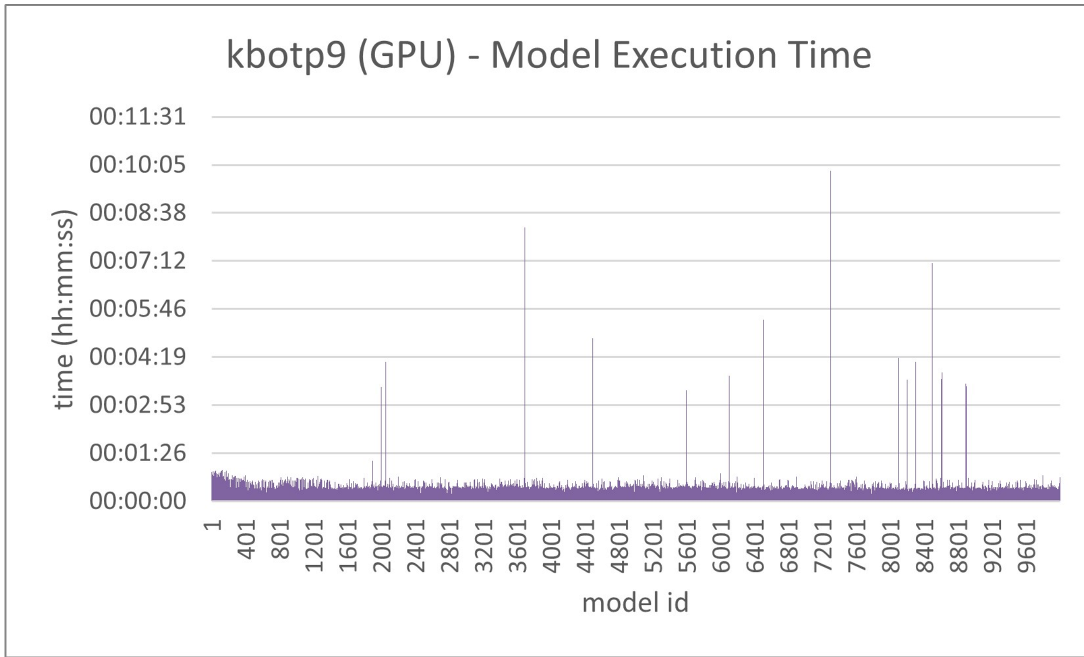


Figure 6.18: kbotp9 (GPU) - Model Execution Time

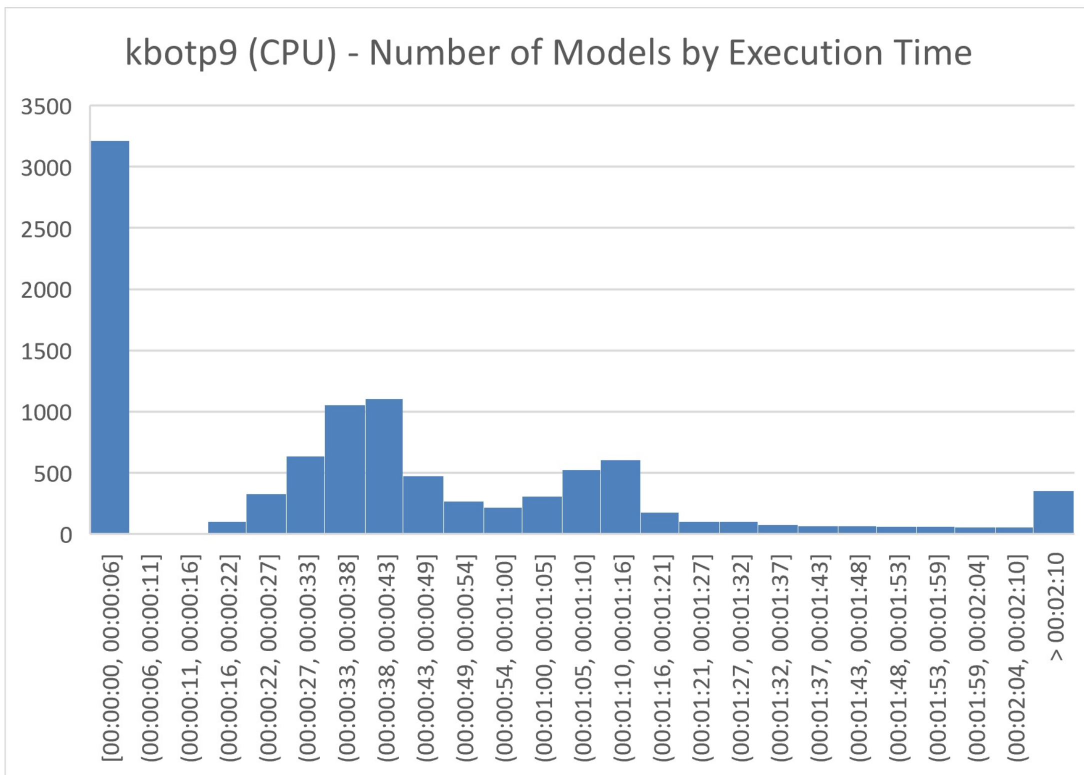


Figure 6.19: kbotp9 (CPU) - Number of models by execution time

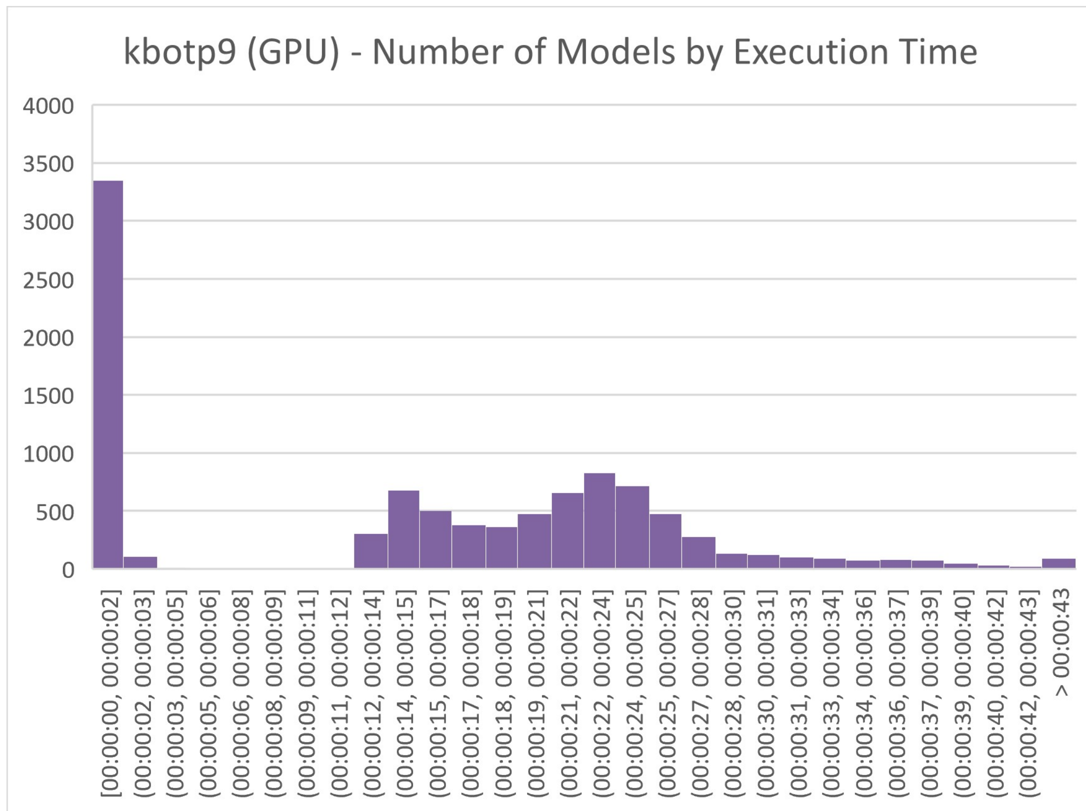


Figure 6.20: kbotp9 (GPU) - Number of models by execution time

This problem took the longest to execute in both CPU and GPU. The CPU execution starts with a notable spike, later stabilizing with executions between 16 seconds and 1 minute and 16 seconds with numerous outliers throughout the execution reaching values over 1 minute and 26 seconds. The GPU execution keeps the execution times somewhat consistent throughout the execution with values between the 12 and 27 seconds. It does however exhibit occasional spikes above the 2 minute and 53 second mark with one case reaching close to 10 minutes. Both CPU and GPU have numerous models executed in under 1 second.

Problem	Type	Training	Testing	Validation	Neurons	Inputs	PH
p1pcsr	Regression	10298x70	2302x70	3419x70	[2:10]	[1:30]	36

Table 6.8: p1pcsr problem configuration

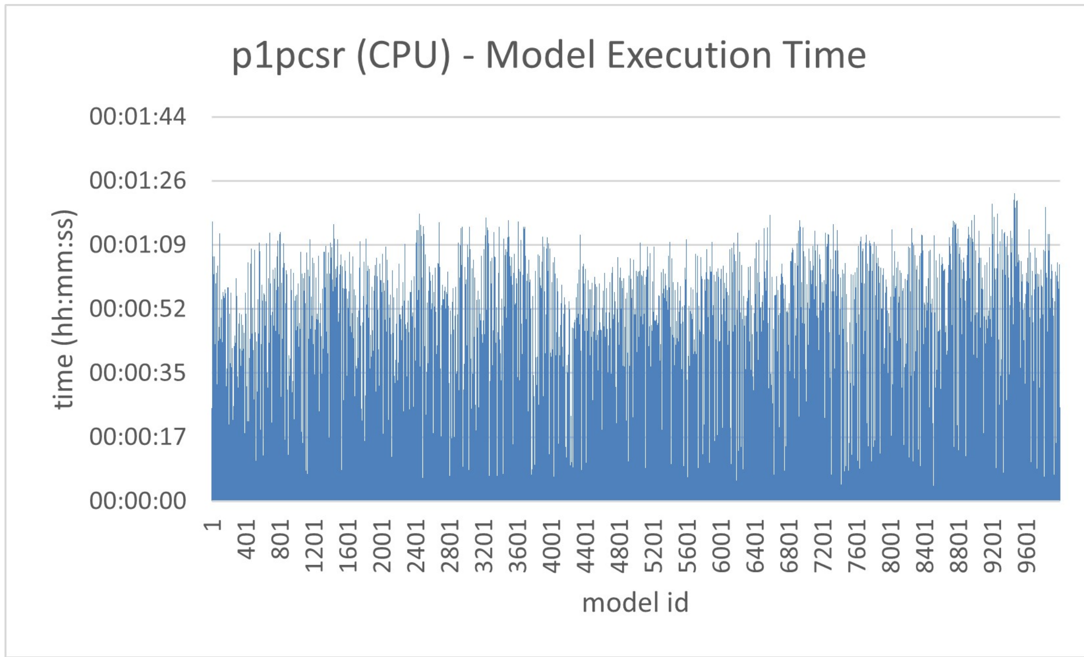


Figure 6.21: p1pcsr (CPU) - Model Execution Time

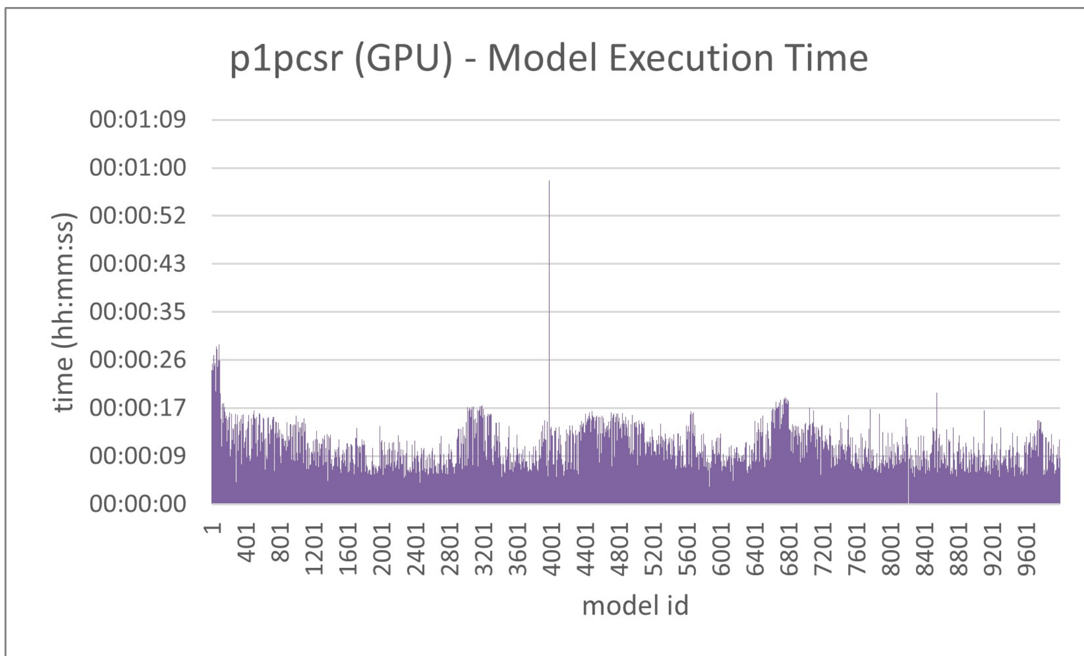


Figure 6.22: p1pcsr (GPU) - Model Execution Time

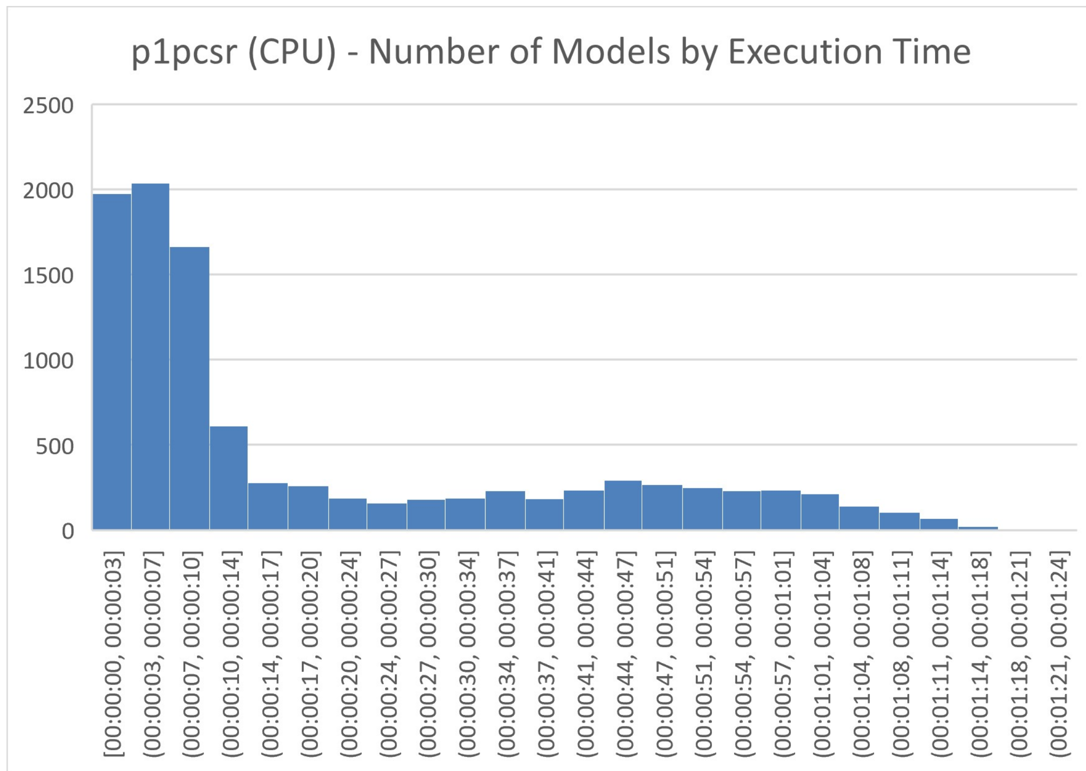


Figure 6.23: p1pcsr (CPU) - Number of models by execution time

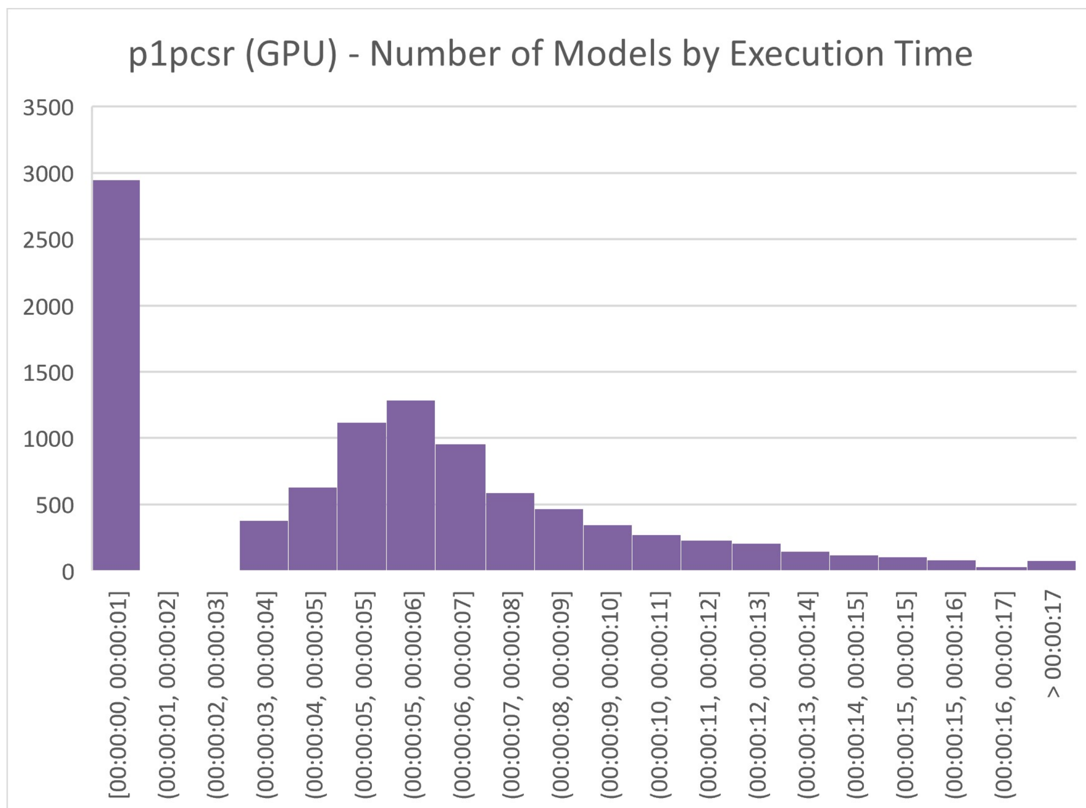


Figure 6.24: p1pcsr (GPU) - Number of models by execution time

The CPU execution has a large distribution of execution times for this problem, as noted in the histogram. Most of the execution time is concentrated in the 3 to 14 second range. With a smaller number of executions falling in the 14 to 1 minute 18 seconds range. The GPU plot exhibits a small spike at the beginning, after which, it keeps its execution times between 3 and 16 seconds. It is worth noting a particular outlier around model 4001, that took close to 1 minute to execute. A number of models were executed in under 1 second in both variants. As mentioned in previous sections, these low model execution times can be mainly attributed to one of two mechanisms: One where the model is copied if it already exists in the problem's database and one where, if the model is deemed unviable, or some problem occurs during training, it is discarded, and placeholder values are saved.

Chapter 7

Discussion and Conclusion

It is impressive how far GPUs have come, from a device whose purpose was to perform rendering in mostly consumer applications, to the fastest architectures in processing large amounts of data. With the constant improvements of both the hardware and software, as well as the ever-increasing amount data that is generated, it is a field that would likely see considerable growth in the years to come.

The aim of this thesis was the creation of model design framework in a GPU Server Platform that could outperform the legacy platform. The work performed should be of use to the researchers that will use the platform in the future. The implemented MOGA platform shows clear signs of being faster than the legacy platform and equally capable of producing models based on the specified configurations, based on the previously performed validation. The creation of the Mini MOGA interface also allows the researcher to estimate the time necessary to complete the problem execution. These two components should facilitate the work performed by future researchers as it would allow them to better manage their time. The creation of diagrams, the documentation of code and the creation of internal documents should, on the other hand, facilitate the work of future developers that would continue the development of this platform by providing them with a base to start from.

Overall this thesis produced some interesting results. Even though it accomplished the objectives it set out to, the results aren't as good as they were initially expected to be. The main reason for this would be inability to create a faster activation function than the one that already exists in 'C'. A part of it could also be attributed to the way the GPU is being used. It has been previously mentioned that the recommended way to utilize a GPU for processing is applying it to one large problem and ideally use one process per GPU.

There is still room for improvement in this platform. It is worth marking as future work: a GPU compatible activation function that outperforms the current 'C' implementation; the restructuring of some of the platform's architecture; a better way to deploy the platform, instead of using bash scripts; restructuring some parts of the GUI.

The results presented consisting of 6 problems executed in both the legacy platform and the GPU platform were condensed in 6.2. In all instances the GPU implementation is consistently faster by a speedup factor of at least 2.64 times but never exceeding 4.84 times in the problems tested. In the individual problem plot comparisons the GPU implementation also reflects this behavior. The GPU plots show consistent slow starts, although still faster than the CPU version in every case. Something unexpected that appeared in some plots are particularly high times to train certain models, these outliers could exist due to an oversight in programming as they seem only present in the GPU version. In a general manner, the GPU implementation could be considered a success.

Bibliography

[1] P. L. Ferreira and A. E. Ruano, *Evolutionary Multiobjective Neural Network Models Identification: Evolving Task-Optimised Models*. Springer Nature, Jan 2011, p. 21–53. [Online]. Available: https://doi.org/10.1007/978-3-642-11739-8_2

[2] S. E. Fahlman and C. Lebiere, *The Cascade-Correlation Learning Architecture*, Jan 1989, vol. 2.

[3] O. R. N. Laboratory. (2022) Frontier supercomputer debuts as the world's fastest, breaking exascale barrier. [Online]. Available: https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier

[4] ——. (2022) Frontier supercomputer specifications. [Online]. Available: https://www.olcf.ornl.gov/frontier/

[5] E. Larsen and D. K. McAllister, *Fast matrix multiplies using graphics hardware*, Nov 2001. [Online]. Available: https://doi.org/10.1145/582034.582089

[6] P. Du, R. Weber, P. Luszczek, S. Tomov, G. M. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, p. 391–407, Aug 2012. [Online]. Available: https://doi.org/10.1016/j.parco.2011.10.002

[7] NVIDIA. Cuda programming guide 1.1. [Online]. Available: http://developer.nvidia.com/object/cuda.html

[8] J. A. Owens, M. Houston, D. Luebke, S. F. Green, J. E. Stone, and J. L. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, p. 879–899, May 2008. [Online]. Available: https://doi.org/10.1109/jproc.2008.917757

[9] J. A. Anderson, C. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, p. 5342–5359, May 2008. [Online]. Available: https://doi.org/10.1016/j.jcp.2008.01.047

[10] C. Chinrungrueng and C. H. Séquin, “Optimal adaptive k-means algorithm with dynamic adjustment of learning rate,” *IEEE Transactions on Neural Networks*, vol. 6, no. 1, p. 157–169, Jan 1995. [Online]. Available: https://doi.org/10.1109/72.363440

[11] A. E. Ruano, P. J. Fleming, and D. Jones, “Connectionist approach to pid autotuning,” *IEE Proceedings*, vol. 139, no. 3, p. 279, Jan 1992. [Online]. Available: https://doi.org/10.1049/ip-d.1992.0037

[12] A. E. d. B. Ruano, *Artificial Neural Networks*, ch. Alternatives to the error back-propagation algorithm, p. 86–87.

[13] K. Bot, S. Santos, I. H. Laouali, A. E. Ruano, and M. Da Graça Ruano, “Design of ensemble forecasting models for home energy management systems,” *Energies*, vol. 14, no. 22, p. 7664, Nov 2021. [Online]. Available: https://doi.org/10.3390/en14227664

[14] E. Hajimani, M. Da Graça Ruano, and A. E. Ruano, “An intelligent support system for automatic detection of cerebral vascular accidents from brain ct images,” *Computer Methods and Programs in Biomedicine*, vol. 146, p. 109–123, Jul 2017. [Online]. Available: https://doi.org/10.1016/j.cmpb.2017.05.005

[15] A. E. Ruano, S. Pesteh, S. Silva, H. B. Duarte, G. Mestre, P. L. Ferreira, H. Khosravani, and R. Horta, “The imbpc hvac system: A complete mbpc solution for existing hvac systems,” *Energy and Buildings*, vol. 120, p. 145–158, May 2016. [Online]. Available: https://doi.org/10.1016/j.enbuild.2016.03.043

[16] H. Khosravani, M. Del Mar Castilla, M. Berenguel, A. E. Ruano, and P. L. Ferreira, “A comparison of energy consumption prediction models based on neural networks of a bioclimatic building,” *Energies*, vol. 9, no. 1, p. 57, Jan 2016. [Online]. Available: https://doi.org/10.3390/en9010057

[17] E. Weyer, “The asmod algorithm some new theoretical and experimental results,” 1995.

[18] T.-Y. Kwok and D.-Y. Yeung, “Constructive algorithms for structure learning in feedforward neural networks for regression problems.” *IEEE Transactions on Neural Networks*, vol. 8, no. 3, p. 630–45, Jan 1997. [Online]. Available: https://doi.org/10.1109/72.572102

[19] J. L. Subirats, L. Franco, and J. M. Jerez, “C-mantec: A novel constructive neural network algorithm incorporating competition between neurons,” *Neural Networks*, vol. 26, p. 130–140, Feb 2012. [Online]. Available: https://doi.org/10.1016/j.neunet.2011.10.003

[20] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain.” *Psychological Review*, vol. 65, no. 6, p. 386–408, Jan 1958. [Online]. Available: https://doi.org/10.1037/h0042519

[21] J. L. Subirats, L. Franco, I. J. Molina, and J. M. Jerez, *Active Learning Using a Constructive Neural Network Algorithm.* Springer Nature, Jan 2009, p. 193–206. [Online]. Available: https://doi.org/10.1007/978-3-642-04512-7_10

[22] M. S. Islam, X. Yao, and K. Murase, "A constructive algorithm for training cooperative neural network ensembles," *IEEE Transactions on Neural Networks*, vol. 14, no. 4, p. 820–834, Jul 2003. [Online]. Available: https://doi.org/10.1109/tnn.2003.813832

[23] Z. Hong, *A preliminary study on artificial neural network*, Aug 2011. [Online]. Available: https://doi.org/10.1109/itaic.2011.6030344

[24] A. Behnood and E. M. Golafshani, "Predicting the compressive strength of silica fume concrete using hybrid artificial neural network with multi-objective grey wolves," *Journal of Cleaner Production*, vol. 202, p. 54–64, Nov 2018. [Online]. Available: https://doi.org/10.1016/j.jclepro.2018.08.065

[25] Y.-H. Lin and Y.-C. Hu, "Electrical energy management based on a hybrid artificial neural network-particle swarm optimization-integrated two-stage non-intrusive load monitoring process in smart homes," *Processes*, vol. 6, no. 12, p. 236, Nov 2018. [Online]. Available: https://doi.org/10.3390/pr6120236

[26] B. Vaidya, *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA: Effective techniques for processing complex image data in real time using GPUs.* Packt Publishing Ltd, Sep 2018.

[27] R. Nishino and S. H. C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," *31st conference on neural information processing systems*, vol. 151, no. 7, 2017.

[28] T. Ridnik, H. Lawen, A. Noy, and I. Friedman, "Tresnet: High performance gpu-dedicated architecture," *2021 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1399–1408, 2020.

[29] J. H. Park, G. Yun, C. Yi, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y. ri Choi, "Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism," *ArXiv*, vol. abs/2005.14038, 2020.

[30] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," *Advances in neural information processing systems*, vol. 2013, pp. 1223–1231, 2013.

[31] H. Khosravani, A. E. Ruano, and P. L. Ferreira, "A convex hull-based data selection method for data driven models," *Applied Soft Computing*, vol. 47, p. 515–533, Oct 2016. [Online]. Available: https://doi.org/10.1016/j.asoc.2016.06.014

[32] T. Soyata, *GPU Parallel Program Development Using CUDA*. CRC Press, Jan 2018.

[33] B. Cannon. Porting from 2 to 3 (python docs). [Online]. Available: https://docs.python.org/3/howto/pyporting.html

[34] (2016) Python conservative conversion guide. [Online]. Available: https://portingguide.readthedocs.io/en/latest/strings.html

[35] (2019) Lsbinitscripts (daemon). [Online]. Available: https://wiki.debian.org/LSBInitScripts

[36] NVIDIA, "Multi-process service," Oct 2022, https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[37] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, p. 80–113, 2007.

[38] E. T. Barron and R. M. Glorioso, "A micro controlled peripheral processor," *Conference record of the 6th annual workshop on Microprogramming - MICRO 6*, 1973.

[39] K. Levine, "Core standard graphic package for the vgi 3400," *ACM SIGGRAPH Computer Graphics*, vol. 12, no. 3, p. 298–300, 1978.

[40] Nvidia, "Nvidia a100 tensor core gpu architecture whitepaper." [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[41] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, *Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers*, Nov 2018. [Online]. Available: https://doi.org/10.1109/sc.2018.00050

[42] M. Pandey, M. Fernández, F. Gentile, O. Isayev, A. Tropsha, A. C. Stern, and A. Cherkasov, "The transformational role of gpu computing and deep learning in drug discovery," *Nature Machine Intelligence*, vol. 4, no. 3, p. 211–221, Mar 2022. [Online]. Available: https://doi.org/10.1038/s42256-022-00463-x

[43] A. Haidar, H. H. Bayraktar, S. Tomov, J. J. Dongarra, and N. J. Higham, "Mixed-precision solution of linear systems using accelerator-based computing," 2020.

[44] NVIDIA. (2016) Web archive nvidia gpu. [Online]. Available: https://web.archive.org/web/20160408122443/http://www.nvidia.com/object/gpu.html

[45] ORNL. (2018) Ornl launches summit supercomputer. [Online]. Available: https://www.ornl.gov/news/ornl-launches-summit-supercomputer

[46] Nvidia, "Summit and sierra supercomputers: An inside look at the u.s. department of energy new pre exascale systems," 2014. [Online]. Available: https://info.nvidianews.com/rs/nvidia/images/Coral%20White%20Paper%20Final-3-2.pdf

[47] N. García-Pedrajas, C. Hervás-Martínez, and J. Muñoz-Pérez, "Multi-objective cooperative coevolution of artificial neural networks (multi-objective cooperative networks)," *Neural Networks*, vol. 15, no. 10, p. 1259–1278, Dec 2002. [Online]. Available: https://doi.org/10.1016/s0893-6080(02)00095-3

[48] E. Weyer and T. Kavli, "Theoretical properties of the asmod algorithm for empirical modelling," *International Journal of Control*, vol. 67, no. 5, pp. 767–790, 1997. [Online]. Available: https://doi.org/10.1080/002071797223992

[49] K. Tanaka, *A Unified Fuzzy Model-Based Framework for Modeling and Control of Complex Systems: From Flying Vehicle Control to Brain-Machine Cooperative Control*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 185–208. [Online]. Available: https://doi.org/10.1007/978-3-642-30687-7_10

[50] Python 2to3 library (pyhton docs). [Online]. Available: https://docs.python.org/3/library/2to3.html

[51] E. Schofield. (2013) Python future compatible idioms. [Online]. Available: https://python-future.org/compatible_idioms.html

[52] NVIDIA. Multi-process service. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[53] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, p. 431–441, Jun 1963. [Online]. Available: https://doi.org/10.1137/0111030