Parallel ANNs on Hybrid Heterogeneous Platforms: Design, Implementation, and Optimization for Performance and Energy

Atefeh Khazaei

UCD student number: 20202243

The thesis is submitted to University College Dublin
in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science

School of Computer Science and Informatics

Head of School: Dr. Neil Hurley

Research Supervisor: Assoc. Prof. Alexey Lastovetsky

May 2025

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. Alexey Lastovetsky, for giving me the opportunity to pursue my PhD in the Heterogeneous Computing Laboratory (HCL). His invaluable guidance kept me on the right path throughout my studies, and I am deeply indebted to him for his insightful advice and unwavering encouragement.

I would like to express my deepest gratitude to Dr. Hamidreza Khaleghzadeh. Beyond being my husband, he has been an invaluable source of technical support and unwavering encouragement throughout my studies. As a fellow researcher, his guidance and belief in my work have made this achievement even more meaningful. This journey would not have been the same without him by my side.

This research has emanated from research conducted with the financial support of School of Computer Science at University College Dublin (UCD). I would like to thank University College Dublin for their financial support in the form of scholarship awards.

I would like to express my deepest gratitude to my family for their unwavering support and love throughout this journey. To my parents, whose encouragement and sacrifices have shaped the person I am today. To my husband, Hamidreza, for his patience, understanding, and steadfast belief in me. And to my little daughter, Ava, who was born during my PhD and brought immeasurable joy and inspiration to my life.

Last but not least, I am deeply grateful to God for his countless blessings throughout my life.

To Hamidreza and Ava

Abstract

Artificial Neural Networks (ANNs) are widely used in various areas, including image recognition, medical science, textual document processing, and autonomous vehicles. State-of-the-art machine learning packages for ANNs are designed for homogeneous platforms, limiting their ability to simultaneously leverage different types of processing resources during training.

This research first proposes a methodology based on a hybrid programming model for designing and implementing parallel and portable heterogeneous ANN applications. The methodology incorporates OpenMP, OpenACC, and Pthreads, facilitating the parallel training of ANN applications on hybrid HPC platforms with diverse types of devices. It then explores the performance and energy efficiency of the proposed methodology on hybrid High-Performance Computing (HPC) platforms. We implement a fully connected multilayer perceptron ANN training application based on the proposed methodology and study its performance and energy behaviour for varying distributions of batches between computing devices on heterogeneous servers with different configurations.

Our findings highlight the outstanding impact of the distribution of batches on the execution time and energy consumption of ANN training, making it a critical decision variable for optimisation. We then mathematically formulate the bi-objective optimisation problem for the proposed ANN methodology, tar-

geting both performance and energy consumption objectives, and apply two solution methods to obtain Pareto-optimal solutions for these two objectives. The improvements in performance and energy saving of the hybrid ANNs are validated on a heterogeneous server consisting of one multicore CPU and two Nvidia GPUs, as well as on simulated hybrid platforms with different numbers of accelerators.

Our results demonstrate that solving the bi-objective optimisation problem results in superior Pareto fronts containing a broad spectrum of trade-off solutions between execution time and energy consumption, where only one solution achieves load balancing and others achieve partial load balancing across computing devices or even complete imbalancing. In addition, adding additional processing resources enhances overall performance in training the ANN. However, incorporating less energy-efficient processing units, while improving training performance, leads to increased energy consumption for performance-optimised solutions. [1]

[1]This research has financially supported by School of Computer Science at University College Dublin (UCD).

Contents

List of Figures

List of Tables

Statement of Original Authorship

I hereby certify that the submitted work is my own work, was completed while registered as a candidate for the degree stated on the Title Page, and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work.

Chapter 1

Introduction

Due to their effectiveness in handling complex modelling, Artificial Neural Networks (ANNs) are increasingly used in a wide range of fields, including image recognition, medical science, textual document processing, autonomous vehicles, etc. Modeling with ANNs is highly computationally intensive because it simulates the super complex human-brain processes; and it is both time- and energy-consuming [2, 3]. Consequently, modern machine learning frameworks, such as TensorFlow [4], Keras [5] (more examples in 2.2) leverage parallel processing, particularly many-core High Performance Computing (HPC) platforms like General-Purpose Graphics Processing Unit (GPGPU) [6] and Tensor Processing Unit (TPU) [7], to meet the escalating demand for faster modeling speed and less energy consumption.

Over the last few years, the modern HPC platforms have become highly heterogeneous owing to the tight integration of multicore Central Processing Unit (CPU) processors and accelerators (such as Graphics Processing Unit (GPU) [8], Intel Xeon Phi (Xeon Phi) [9], or Field Programmable Gate Array (FPGA) [10]). These heterogeneous platforms can be one of the best options to reduce the concerns about the growth of ANNs applications. Optimal resources usage on heterogeneous platforms can significantly increase performance and reduce energy consumption.

In light of the significance and popularity of ANNs, coupled with the diverse range of available computing resources, the overarching objective of this re-

search is to investigate strategies for optimising the performance and energy efficiency of ANNs on modern hybrid heterogeneous platforms.

Through a brief survey of mainstream solutions and conversations with experts, we identified several key observations:

- **ANNs are getting omnipresent,** and their models are growing increasingly complex and large. As a result, training these models demands more time and computational resources. Traditional serial implementations can take days, months, or even years to complete training.
- **To overcome this bottleneck,** most popular ANNs applications and frameworks leverage parallel computing—naturally aligning with the design of modern processors and servers, which are inherently parallel. This approach significantly speeds up ANNs modeling.
- **However, energy efficiency is largely overlooked.** Despite rising concerns about climate change and growing awareness among Artificial Intelligence (AI) experts, mainstream frameworks typically do not optimise for energy consumption. Energy usage has become a critical optimisation goal, yet remains unaddressed in most current solutions.
- **Modern computing systems are increasingly heterogeneous,** combining general-purpose CPUs with one or several specialized processors known as accelerators (like GPUs)—even on ordinary desktops and laptops. However, existing frameworks can not utilise all available hardware concurrently. They rely either on multicore CPUs or GPUs, but not both in tandem during ANN modeling.

These observations clarify why this project aims to explore how to develop ANNs applications that fully utilise all computing devices within hybrid heterogeneous platforms, and how to optimise their execution for both performance and energy efficiency. Our *original plan* was as follows:

- We aimed to identify and examine existing research parallel implementations of ANNs applications designed for heterogeneous platforms—specifically those that utilise all available computing devices of

platforms and are ideally portable and configurable. The goal was to understand the current state of the art in developing hybrid parallel ANNs applications.

- Once we gained sufficient insight into how to design and implement such applications in a portable and configurable way, the next step was to explore strategies for optimising their execution with respect to both performance and energy efficiency.

However, our extensive investigation revealed a significant gap: there is no existing ANN application specifically implemented for execution on hybrid heterogeneous platforms. This presented our **first major challenge** and prompted a shift in our original plan. Our primary research question now focuses on how to design and implement a parallel ANN application for hybrid heterogeneous platforms in a way that is both portable and configurable. This overarching question breaks down into several sub-questions. The first is to define the scope of our application, as it's not feasible to cover all types of ANNs and all phases of the modeling process within this research. The defined scope of our application is as follows:

- Modeling of ANNs has two main phases: Training and Inference. We have focused on the training phase. Why? Because it is the most resource and energy consuming phase. Also, the use of pre-trained ANN models is becoming increasingly common. They are ANN models that have already been trained on large datasets and can be fine-tuned or directly applied to new tasks. Training or retraining of these pre-trained models can be highly time-consuming due to their complexity and size [11].

- We have focused on studying data parallelism for training ANNs, as it is the only type of parallelism that effectively scales. It is also the most widely used form of parallelism in parallel ANNs, with other types being less common and offering limited potential for acceleration. Their scalability and subsequent performance gains are restricted (more details in

section 2.1). We would like to explore the mainstream parallelism that holds the greatest potential for performance improvements.

- Among the various types of ANNs, we have chosen to focus on Fully Connected Networks, which serve as the foundation of neural networks by modeling complex functions through learned weighted connections between neurons. As discussed earlier, we employ data parallelism approach that is particularly effective when individual training samples can be processed independently. Under this assumption, we expect that our data parallelism strategy applied to fully connected ANNs can be extended to a wide range of other ANN architectures (see Chapter 6 for further discussion).

We addressed the first challenge by proposing a programming model and implementation methodology for data-parallel hybrid heterogeneous ANNs applications that are both portable and configurable. To the best of our knowledge, this is the first solution of its kind.

Having answered the first research question and successfully implemented the ANN application, we then turned to the next question: how to optimise its execution for both performance and energy efficiency. Our approach focuses on using workload distribution across devices as the sole decision variable to determine the optimal configuration—one that minimises both execution time and energy consumption.

1.1 Motivations of This Research

State-of-the-art machine learning packages and applications are designed for homogeneous execution on identical processing devices. Since identical processing devices have similar speeds and power consumption, distributing the training workload across a homogeneous cluster of identical devices can be more straightforward. The current ANNs packages divide the workload *equally* among all devices; but this method does not guarantee optimal performance and energy consumption for parallel training ANNs executing even on homogeneous clusters. Lastovetsky and Reddy [12] proved that distributing the

workload equally across all devices is optimal when the performance profile follows a linear pattern. However, when the performance profile is nonlinear, an equal workload distribution does not guarantee optimality, even in homogeneous clusters.

On the other side, none of available parallel ANNs applications have considered energy consumption as an optimisation objective alongside performance. For these applications, reducing the energy consumption is only a positive side effect of parallelisation.

In today's world, most machines—from Personal Computers (PCs) and laptops to small and large clusters—contain heterogeneous devices, making them widespread. In the backdrop of the tight integration of multi-core CPUs and various many-core accelerators, such as GPUs [8], FPGAs [10], Digital Signal Processor (DSP) [13], and Intel Xeon Phis [9], mainstream HPC platforms have become increasingly heterogeneous over the past decade, to maximise key objectives like performance, energy efficiency, cost-effectiveness, and flexibility. In a nutshell, on real-world platforms, current ANNs applications can not utilise resources efficiently, and their energy consumption is not optimal.

Since state-of-the-art ANNs packages and applications are designed for homogeneous platforms, they can not take full advantage of the performance and energy efficiency benefits offered by hybrid heterogeneous HPC platforms. For instance, TensorFlow [4], Keras [5], and NVIDIA DIGITS [14] (more examples in 2.2), support operations on both CPU and GPU, enabling the execution of training tasks on either a CPU, a GPU, or a cluster of identical CPUs or GPUs. However, they cannot configure the framework to distribute the training workload across different types of devices for the simultaneous execution of training tasks on a hybrid heterogeneous CPU-GPU platform. In other words, although these ANNs applications are facilitated with heterogeneous kernels, they only support homogeneous execution at runtime. This limitation restrains the current parallel ANNs applications from achieving reasonable performance and energy savings on hybrid heterogeneous platforms by harnessing the processing power of all processing resources simultaneously.

Given the suboptimal performance and energy consumption of existing ANNs applications on one hand and the widespread presence of heterogeneous devices on the other, this research aims to introduce the first optimal parallel ANNs application in a portable and configurable form for hybrid heterogeneous platforms. However, achieving this goal comes with the following challenges:

1. Could we find an available parallel implementation in a portable configurable form for hybrid heterogeneous platforms? At the beginning of this research, we hypothesised that the code of one of the available parallel ANNs applications can be used to develop a set of portable and configurable parallel applications on hybrid heterogeneous platforms. If we could find such a parallel ANNs application, we would optimise its performance and energy for hybrid heterogeneous platforms in a portable and configurable form. But, after dedicating significant time to this hypothesis, we realized it was incorrect. None of the available ANNs applications has been parallelised on hybrid heterogeneous platforms (more details 2.2).

2. Upon discovering that no parallel ANNs applications existed for hybrid heterogeneous platforms, we faced a new challenge: How could we design a parallel ANNs application that is both portable and configurable for hybrid heterogeneous platforms? To address this challenge, we explored various network parallelisation methods—data parallelism, model parallelism, and pipelining—to identify the most suitable approach for a parallel, portable, and configurable ANNs application on hybrid heterogeneous platforms (details provided in 2.1).

3. Once this parallel ANNs application is designed, the next challenge is implementing it in a portable and configurable manner for hybrid heterogeneous platforms. The wide variety of accelerators available today, each with distinct programming models, frameworks, and optimisation techniques, significantly increases the complexity of implementing ANNs training algorithms. Consequently, lots of algorithms designed and developed for a specific device are not portable to other

types of devices. To address this challenge, we have chosen to utilise Open Multi-Processing (OpenMP) [15], Open Accelerators (OpenACC) [16, 17], and POSIX threads (POSIX: Portable Operating System Interface) (Pthreads) [18], as detailed in 4.2.

4. How could we optimise our parallel ANNs application for hybrid heterogeneous platforms? In other words, How could we distribute workloads in a manner that optimises both performance and energy consumption? Energy consumption efficiency is an aspect that previous ANNs applications have not addressed. To tackle this issue, we employ bi-objective optimisation algorithms for performance and energy, as detailed in 3.2.

1.2 Objectives of This Research

To the best of our knowledge, no ANNs parallel implementation has yet been proposed that achieves optimal performance and energy efficiency on hybrid heterogeneous platforms (more in 2.2). In this research, for the first time, we introduce and develop a novel methodology for designing parallel, portable and configurable ANNs applications that can be executed on hybrid heterogeneous platforms, leveraging the simultaneous utilisation of different types of computing devices for training. To tackle this challenge, a hybrid programming model is introduced, incorporating three parallel programming standards: OpenMP [15], OpenACC [16, 17], and Pthreads [18]. In this model, OpenMP, as an industry-standard Application Programming Interface (API) for shared-memory parallel programming, is employed to implement portable, multi-threaded ANNs training kernels for CPUs. OpenACC is used to develop portable parallel kernels for offloading and executing parallel ANNs training computations onto various accelerators, such as Nvidia GPUs. Pthreads is utilised to integrate the CPU and accelerator kernels into a unified hybrid parallel ANNs application, enabling the simultaneous execution of ANNs training across hybrid heterogeneous platforms with arbitrary numbers and types of processing resources. We validate our proposed methodology by implementing a fully connected multilayer perceptron ANNs application.

The next goal is to propose an approach to optimise the execution of the ANNs applications developed based on the proposed hybrid methodology for performance and energy on heterogeneous HPC systems via workload distribution. To address this, we first study the performance and energy consumption behaviour of the implemented Fully Connected ANNs application on heterogeneous platforms, consisting of one Intel multi-core CPU integrated with Nvidia GPUs, for varying numbers of batches allocated to each kernel (device). These case studies reveal two key findings: (i) Batches distribution has a significant impact on the performance and energy consumption of our ANNs applications, and (ii) performance and energy consumption are two conflicting objectives for optimisation, where improving one typically results in the degradation of the other.

Motivated by these observations, we consider batches distribution a critical decision variable that can no longer be ignored in the bi-objective optimisation of the proposed ANNs methodology for performance and energy. We mathematically formulate the optimisation problem for hybrid ANNs applications by considering batches distribution as the only decision variable. To solve this, we apply two solution methods to obtain Pareto optimal sets containing a wide range of trade-off solutions between the two conflicting objectives execution times and energy consumption. Each solution in the Pareto front determines a specific batches distribution for ANNs training across the processing devices in a heterogeneous platform. Notably, while the optimal solution for performance balances batches distribution across all processing resources, the other solutions in the Pareto-optimal set exhibit partial balance or even complete imbalance. Remarkably, no prior research has addressed the bi-objective optimisation of hybrid ANNs via batches distribution to obtain a comprehensive range of exact trade-off solutions for both performance and energy consumption.

Our parallel, portable and configurable ANNs application and bi-objective optimisation methods were experimentally validated on several heterogeneous platforms with various numbers of accelerators. The results demonstrated a wide range of trade-off solutions between execution times and energy consumption within Pareto optimal sets obtained for the parallel ANNs application. These solutions allow users to configure the ANNs application to pri-

oritise either higher performance or greater energy savings, depending on their preferred objectives. The experimental results revealed that including additional processing resources for training an ANNs enhances overall performance. However, incorporating less energy-efficient processing units, while improving training performance, resulted in increased energy consumption for performance-optimised solutions.

1.3 Contributions of This Research

Based on our research objectives, our main contributions in this thesis are:

1. A methodology for designing parallel, portable and configurable ANNs applications that can be executed on hybrid heterogeneous platforms with arbitrary numbers and types of accelerators.

2. A hybrid programming model for implementing hybrid parallel ANNs applications based on the proposed methodology, utilising three programming standards: OpenMP, OpenACC and Pthreads. Implementation of a fully connected multilayer perceptron ANNs training application using the proposed methodology and programming model.

3. A methodology for optimising hybrid parallel ANNs applications for performance and energy consumption by obtaining Pareto-optimal solutions and enabling tradeoffs between these objectives through batches distribution across computing devices.

4. Bi-objective optimisation of the hybrid fully connected multilayer perceptron ANNs application for performance and energy on different heterogeneous server configurations.

1.4 Thesis Structure

The structure of this thesis is as follows. In Chapter 2, we present related works on ‘ANN parallelisation’, and compare available ANNs applications. In

Chapter 3, we review the related works in ‘Performance Optimisation’, and ‘Bi-objective Optimisation for Performance and Energy’ on heterogeneous platforms. Chapter 4 will cover the implementation details; this is followed by Chapter 5, which describes how the experiments were designed and what results have been obtained. Finally, Chapter 6 concludes the thesis.

Chapter 2

Background on ANNs Parallelisation

In this chapter, first, the related research works in ANNs parallelisation areas are reviewed and categorised. Following this, the most popular available parallel ANNs implementations will be reviewed and compared.

2.1 ANNs Parallelisation

Up to now, there are lots of researches that have focused on the parallel execution of ANNs to reduce the execution time. Most of these studies have focused on homogeneous environments, not heterogeneous ones (See 2.2). For most of these researches, optimising the energy consumption is not an objective; it is only a positive side effect of these parallelisation [19]. Despite these differences between previous researches and ours, the investigation of these studies has given us a good idea of the state of the art in parallelisation methods and which methods should be implemented in ANNs to be used in our research.

ANNs parallelisation methods can be categorized in 2 main categories: Parallelisation in Operators, and Parallelisation in Networks. Each of these two main categories has several subcategories (Figure 2.1).

In *parallelisation in operators* studies, the researchers have tried to take

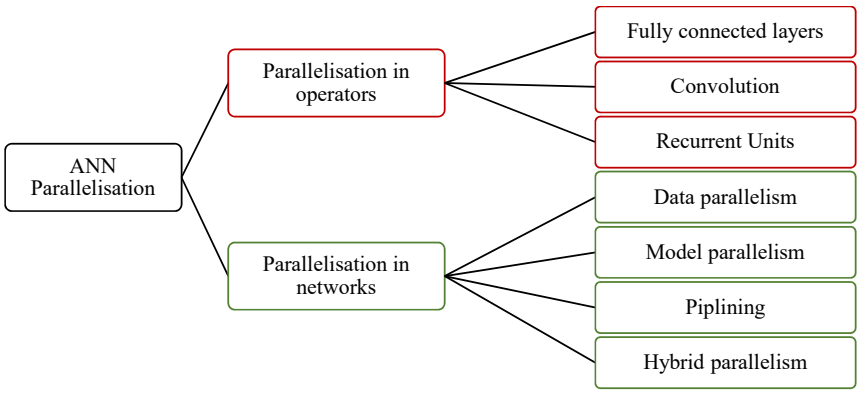

Figure 2.1: Categorisation of ANNs parallelisation researches.

advantage of the opportunities for parallelising layer execution. In some of ANNs' types, computations can be parallelised directly; and in other networks types, computations have to be reshaped to reveal parallelism [1]. In the following the parallelism of three popular operators have been addressed:

- **Fully Connected Networks:** A fully connected layer can be expressed and modeled as a matrix-multiplication of the weights and the neuron values. To this aim, efficient linear algebra libraries, such as CUDA Basic Linear Algebra Subroutines library (cuBLAS) [20], Intel Math Kernel Library (MKL) [21], and IBM Engineering and Scientific Subroutine Library (ESSL) [22], can be used.

  Vanhoucke et al. [23] have presented some different methods to further optimise CPU implementations of fully connected layers. In particular, this research shows efficient loop construction, vectorization, blocking, unrolling, and batching. These researchers also proved how weights can be quantized to use fixed-point math instead of floating-point.

- **Convolutional Neural Network (CNN):** The research community and the industry have put considerable effort into optimizing CNN computation on different platforms. The first algorithmic change proposed for CNNs was the use of the famous technique to transform a discrete convolution into matrix multiplication, using Toeplitz matrices (usually known as im2col) [24, 25].

The second method proposed for CNNs is using of the Fourier domain, in which convolution is defined as an element-wise multiplication. In this method, both the data and the kernels are transformed using Fast Fourier Transform (FFT), multiplied, and the inverse FFT is applied on the result [26].

The third and the prevalent method used today to perform CNNs is Winograd's algorithm for minimal filtering [27]. This method, first, proposed by Lavin and Gray [28], and modifies the original CNN algorithm for multiple filters that there are in convolutions. Due to the wide range of applications for CNNs, researchers continue to explore ways to accelerate these models [29, 30].

- **Recurrent Neural Network (RNN):** The gate systems that situate within RNN units (e.g., Long Short-Term Memory (LSTM)) contain multiple operations, each of which does a small matrix multiplication or an element-wise operation. Due to this reason, these layers were commonly implemented as a series of high-level operations; but the further acceleration of such layers is possible. On the other hand, RNN units are usually chained together (forming consecutive recurrent layers), so two types of parallelism can be considered: within the same layer, and between consecutive layers [1].

It is important to mention that other types of ANNs (operators) have also been the focus of parallelization efforts by some researchers, such as Graph Neural Network (GNN) [31] and Spiking Neural P neurons (SNP) Convolutional Network [32]. However, our focus in this review is on the most popular and widely used ones: Fully Connected Networks, CNNs, and RNNs.

In this research, we focused on Fully Connected Networks, which form the foundation of ANNs by modeling complex functions through learned weighted connections between neurons. The parallel implementation of these networks can be adapted to other neural network architectures, Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), by utilising similar matrix operations and activation functions, which we consider for future work.

(a) Data Parallelism (b) Model Parallelism (c) Layer Pipelining

Figure 2.2: Parallelisation in networks methods [1].

The high average parallelism in neural networks may not only be done to compute individual operators efficiently but also to parallelise the whole network with respect to different dimensions. In the following the main partitioning strategies (parallelisation in networks) have been addressed: partitioning input samples (data parallelism), partitioning the network structure (model parallelism), and partitioning the layer (pipelining) [1] (Figure 2.2).

- **Data Parallelism:** This method is a straightforward approach for ANNs parallelisation; in this method the work of the batch samples is partitioned among multiple computational resources (cores or devices). Today, data parallelism is supported by the vast majority of ANNs frameworks, using a single GPU, multiple GPUs, or a cluster of multi-GPU nodes [1].

  In this method, all ANNs parameters have to be accessible for all processors, which means that they should be replicated. The scaling of data parallelism is naturally defined by the batch size. In the weight update phase, the results of the partitions have to be averaged to obtain the gradient [33, 34].

- **Model Parallelism:** Model parallelism is also known as network parallelism. This strategy divides the work according to the neurons in each layer. In this method, the batch samples are copied to all processors, and different parts of the ANNs are computed on different processors. This method can conserve memory (since the full network is not stored in one place) but cause additional communication after every layer [35, 36].

- **Piplining:** In neural networks, pipelining can either refer to overlapping computations (for example, between one layer and the next one as data becomes ready); or to partitioning the ANNs according to depth, assigning layers to specific processors. In another view, pipelining can be a form of data parallelism, since the samples are processed through the network in parallel as model parallelism [37].

  The first and widely used in practice form of pipelining is overlapping of feedforward, backpropagation, and weight updates [38]. This scheme increases utilisation by mitigating processor idle time.

- **Hybrid Parallelism:** In some researches, researchers try to combine multiple parallelism schemes to overcome the drawbacks of each scheme [39, 40, 41].

Table 2.1 provides a summary of the network parallelisation methods.

Our main research objective is to develop an optimal parallel ANNs application on hybrid heterogeneous platforms. This application aims to reduce execution time and energy consumption by optimally distributing the workload across heterogeneous devices.

Modern machines equipped with various types of accelerators have the capability to execute thousands of tasks simultaneously. As Table 2.1 shows Data Parallelism is the only method that effectively leverages this power to run lots of tasks in parallel. Model Parallelism and Pipelining methods can hardly balance workload across devices. In other words, defining a large number of tasks for these models is a challenge so they are not efficient on hybrid heterogeneous platforms.

Additionally, the Data Parallelism approach is independent of the network topology. Unlike Model Parallelism and certain Pipelining implementations, the number of tasks and workload distribution do not rely on the network topology.

In summary, Data Parallelism is the most scalable, efficient, and suitable parallelisation method for hybrid heterogeneous platforms. Therefore, in this research, we implemented Fully Connected Networks using Data Parallelism, optimising them for hybrid heterogeneous environments.

Table 2.1: Comparing parallelisation in networks methods.

Parallelism Type	Summary	Advantages	Disadvantages
Data Parallelism	The same model is replicated across multiple devices, each processing different subsets of the data. After processing, gradients are aggregated and updated globally.	* Scales well especially with increasing data size. * Easier to implement. * Independent of network topology (flexibility). * No need to modify the model architecture.	- Synchronization overhead. - Batch size constraints may limit scaling efficiency.
Model Parallelism	The model is split across multiple devices, each handling a different part of the model. Suitable for very large models that don't fit in one device's memory.	* Enables training of very large models that exceed a single device's memory. * Reduces memory footprint per device.	- Harder to balance workload across devices, leading to inefficiencies. - Dependent on network topology. - Increased inter-device communication overhead. - More complex implementation compared to data parallelism.
Pipelining	The model is divided into sequential stages, each placed on a different device. Batches are processed in a staggered manner to improve efficiency.	* Helps hide communication overhead by overlapping computation with data transfer. * More efficient utilisation of hardware resources compared to model parallelism.	- Requires careful workload balancing to avoid pipeline stalls. - Higher latency for individual samples. - More complex to implement and debug.

2.2 Available Parallel ANNs Implementations

At the beginning of this research, we hypothesised that the code of one of the available parallel ANNs applications can be used to develop a set of portable and configurable parallel applications on hybrid heterogeneous platforms. If we could find such a parallel ANNs application, we would optimise its performance and energy for hybrid heterogeneous platforms in a portable and configurable form. But, after dedicating significant time to this hypothesis, we realized it was incorrect.

We have investigated lots of available ANNs implementations. Considering the goals of this research, we were looking for implementations that satisfy the following specifications:

- Open Source,
- Parallelised on at least CPUs and GPUs,
- Run on hybrid platforms (Simultaneous execution on different types of processors),
- General usage (No dependency on specific platforms),
- Simple to extend it and apply different optimisation methods

Table 2.2 lists some of the most well-known available ANNs packages [42, 43, 44]. This table compares the applications based on the supported types of ANNs (Fully Connected (FC in the table), CNN, and RNN), their parallelisation status (parallelised on CPUs, GPUs, and Hybrid platforms), Open Source (OS), and Industrial/Research application (Inds/Rsrch).

Table 2.2 highlights our effort to include the most significant industrial and research applications in our review. Among these, Keras [5] and TensorFlow [4] belong to both categories and are currently the most widely used ANNs applications globally. Neural Designer [45] and NeuroSolutions [46] are categorized as industrial applications, while the remaining ones fall under research applications.

Table 2.3 provides a comparison between industrial and research applications. In summary, industrial applications prioritize reliability, scalability, and performance for business use, emphasizing ease of use and efficiency in production environments. In contrast, research applications focus on innovation, experimentation, and testing new ideas, offering greater flexibility for model development but often lacking the stability and optimisation required for industrial deployment.

The most important finding in Table 2.2 is that none of the investigated ANNs applications have 'Y' in the last columns. In other words none of these applications can simultaneously execute on different types of processors. One of the key contributions of our research is the first-ever presentation of a parallel and portable ANNs application capable of running on hybrid heterogeneous platforms with any number and type of accelerators.

Additionally, none of the reviewed applications are sufficiently simple or well-documented to serve as a benchmark for applying our optimisation methods on heterogeneous platforms.

After investigating the available ANNs applications, we decided to present our ANNs applications as one of our important contributions. The presented application have the following specifications:

- *Implemented using C++ programming language*,
- *Open Source*,
- *Support Fully Connected Neural Networks*,
- *Parallelised by Data Parallelism method*,
- *Parallelised on CPUs and GPUs using OpenMP, OpenACC, and Pthreads packages*,
- *Run on heterogeneous hybrid platforms*,
- *Independent from specific platforms*,
- *Simple, configurable, and well-documented*.

Table 2.2: Comparing some well-known available ANNs applications.

App Name	OS	FC	CNN	RNN	CPUs	GPUs	Hybrid	Inds/ Rsrch
Neural Designer [45]	N	Y	N	N	Y	Y	N	Inds
Neuroph [47]	Y	Y	N	N	N	N	N	Rsrch
Darknet [48]	Y	Y	Y	Y	Y	Y	N	Rsrch
Keras [5]	N	Y	Y	Y	Y	Y	N	both
TensorFlow [4, 38]	Y	Y	Y	Y	Y	Y	N	both
TFlearn [49]	N	Y	Y	Y	Y	Y	N	Rsrch
ConvNetJS [50]	Y	Y	Y	N	N	N	N	Rsrch
NeuroSolutions [46]	N	Y	N	N	N	N	N	Inds
NVIDIA DIGITS [14]	Y	Y	Y	Y	Y	Y	N	Rsrch
SNNS [51]	N	Y	Y	Y	Y	Y	N	Rsrch
Torch [52]	Y	Y	Y	Y	Y	Y	N	Rsrch
MLPNeuralNet [53]	Y	Y	N	N	N	N	N	Rsrch
DNNGraph [54]	Y	Y	Y	N	N	N	N	Rsrch
DeepPy [55]	N	Y	Y	N	N	Y	N	Rsrch
Aforge.Neuro [56]	N	Y	N	N	N	N	N	Rsrch
Cuda-convnet2 [57]	Y	Y	Y	N	N	N	N	Rsrch
Dn2A [58]	Y	Y	Y	Y	N	N	N	Rsrch
Knet.jl [59]	Y	Y	Y	Y	Y	Y	N	Rsrch
HNN [60]	Y	Y	N	N	Y	Y	N	Rsrch
Lasagne [61]	Y	Y	Y	Y	N	N	N	Rsrch
Mocha [62]	Y	Y	Y	N	Y	Y	N	Rsrch
LambdaNet [63]	Y	Y	N	N	Y	Y	N	Rsrch
GoBrain [64]	Y	Y	N	Y	N	N	N	Rsrch
NEON [65, 66]	Y	Y	Y	Y	N	Y	N	Rsrch
Deeplearn-rs [67]	Y	Y	Y	Y	N	N	N	Rsrch
RustNN [68]	Y	Y	N	Y	Y	Y	N	Rsrch

Table 2.3: Comparing Industrial and Research applications.

Category	Industrial Applications	Research Applications
Purpose	Focus on solving real-world problems in business and industry.	Focus on experimentation, development, and exploration of new techniques and models.
Users	Developers, data scientists, and engineers in industries such as healthcare, finance, etc.	Researchers, scientists, and developers in academic and experimental settings.
Performance and Stability	Highly optimised for speed, scalability, and stability in production environments.	Focus on flexibility and experimental features, may sacrifice some stability for research.
Complexity	Aimed at providing a complete solution that is easy to integrate and use in the industry.	Allows for customization, experimentation, and rapid prototyping.
Documentation and Support	High-quality documentation and support for ease of use and deployment.	May have less support but offers deep technical documentation for customization.
Real-World Application	Primarily used in production environments, solving industry-specific problems.	Primarily used for theoretical exploration, model testing, and academic research.

C++ is well-suited for parallel programming because it provides low-level memory control, efficient resource management, and supports multithreading through libraries like OpenMP, Message Passing Interface (MPI), and C++ standard threading. Its performance-oriented design enables developers to optimise execution across multiple cores and processors [69].

OpenMP [15], OpenACC [16, 17], and Pthreads [18] are suitable for implementing applications because they provide efficient parallel programming models tailored to different hardware architectures. OpenMP simplifies multithreading for general-purpose multicore CPUs, allowing easy parallelisation of loops and tasks. OpenACC provides performance portability across heterogeneous systems by supporting NVIDIA GPUs, multicore CPUs, and, depending on compiler support, other accelerator architectures such as AMD GPUs. It enables directive-based parallelism without requiring in-depth hardware-specific programming expertise (See Section 4.2 for a detailed discussion of this choice). Pthreads (POSIX Threads) offer fine-grained control over CPU thread management, making them ideal for performance-critical applications requiring low-level concurrency. These packages enhance computational efficiency, scalability, and portability, making them essential for developing high-performance applications across diverse computing environments.

At the conclusion of this chapter, we can assert that, to the best of our knowledge, no ANNs parallel implementation has yet been proposed that achieves optimal performance and energy efficiency on hybrid heterogeneous platforms. In this research, we introduce, for the first time, a parallel implementation of Fully Connected Networks using the Data Parallelism method, achieving optimal performance and energy efficiency on hybrid heterogeneous platforms.

Chapter 3

Background on Performance and Energy Optimisation

This chapter begins with a review of related researches in the field of performance optimisation. Next, it explores relevant studies on bi-objective optimisation for performance and energy. The final two sections provide more details of the Heterogeneous Energy Performance Optimisation Algorithm (HEPOPTA) and Linear Bi-objective OPtimisation Algorithm (LBOPA) algorithms, which we utilise in our research to address bi-objective optimisation for both performance and energy.

3.1 Performance Optimisation

The simplest approach for performance optimisation of data-parallel applications executing on HPC platforms is the Constant Performance Model (CPM). In this technique, the speed of applications is characterised using positive constant numbers such as normalised cycle time, normalised processor speed, or average execution time [70, 71, 72]. Then, the computational workload is distributed among processing devices according to their execution speeds. This approach assumes no dependency between processor performance and workload size, and the optimal solutions balance workloads across processing devices.

The advanced load-balancing algorithms use application-specific models such as the Functional Performance Model (FPM). In FPM, the speed of processors is modelled by continuous functions of problem size, where the shapes of these functions are assumed to be smooth enough. This ensures that the optimal solutions minimizing execution time are always load-balanced [73, 74].

Later, to address the limitations of FPM-based load-balancing algorithms, new model-based optimisation algorithms were proposed that consider the real-life performance profiles of applications [75, 76, 77, 78, 79]. These algorithms take the most general shapes of performance profiles as input and determine optimal workload distributions that minimise execution time. Unlike load-balancing algorithms, the optimal solutions derived from these algorithms may not balance the workload across processors.

3.2 Bi-objective Optimisation for Performance and Energy

In this section, we review state-of-the-art research works focusing on bi-objective optimisation of performance and energy consumption on HPC platforms. The proposed methods are categorised into two main groups:

1. System-level approaches,
2. Application-level approaches.

System-level methods reconfigure system or environment parameters to achieve optimal performance and energy consumption. These approaches employ a diverse range of system-related parameters, such as task computation costs, communication costs, supply voltage, clock frequencies, number of processors, Dynamic Voltage and Frequency Scaling (DVFS) levels, cache size, and thermal design power. Mathematical, heuristic, or machine learning algorithms are then used to optimally map tasks to the platform [80, 81, 82, 83, 84, 85].

3.2.1 Application-level optimisation

Application-level methods essentially modify applications to achieve optimal performance and energy consumption. This type of approaches relies on application-level models for predicting the performance and energy consumption of applications.

Application-level methods primarily modify applications to achieve optimal performance and energy consumption. These approaches rely on application-level models to predict the performance and energy consumption of applications, enabling the optimisation process.

Subramaniam et al. [86] propose a multi-variable regression algorithm to obtain partial trade-off solutions between performance and energy consumption. Their approach is based on four input parameters: problem size, block size, the number of process rows, and the number of processor columns in the process grid.

Gholkar et al. [87] propose a solution method for performance-power optimisation under a limited power budget. The algorithm takes as inputs the maximum number of processors, the machine's power budget, and the number of processors. The decision variables for optimisation are the CPU frequency and the optimal number of processors allocated for a job.

Gabaldon et al. [88] propose a genetic algorithm to compromise between execution time and energy consumption for parallel applications. The decision variable is task scheduling or mapping. Energy consumption is modelled as a mathematical function incorporating idle and computing energy consumption, along with computing and idle execution times.

Chakrabarti et al. [89] propose a bi-objective optimisation algorithm based on workload distribution on heterogeneous platforms. Performance is formulated as a linear function of workload size, while energy consumption is predicted using historical data tables.

Manumachu et al. [90, 91] propose algorithms to address the bi-objective optimisation problem for performance and energy consumption on homogeneous multicore platforms. These algorithms take as inputs real-life execution time and energy consumption functions, employing workload distribution as

the sole decision variable.

Khaleghzadeh et al., after introducing an algorithm for dynamic energy optimization on heterogeneous high-performance computing platforms [92, 93], concentrated on bi-objective optimization problems. They proposed an algorithm to address discrete bi-objective optimisation problems on heterogeneous platforms for performance and dynamic energy, as well as for execution time and total energy. The discrete performance and dynamic energy functions of each processor are provided to the algorithm as input, which then returns a full set of Pareto-optimal solutions. The decision variable is workload distribution [94, 95]. Later, they propose another workload-distribution algorithm to solve the continuous bi-objective optimisation problem on heterogeneous HPC platforms for performance and energy, where the performance functions are continuous and strictly increasing, and the energy functions are linearly increasing [96, 97].

Our research aims to develop an optimally parallelised Fully Connected ANNs application for hybrid heterogeneous platforms. To achieve this, we employ two algorithms to address bi-objective optimisation for both performance and energy:

- **HEPOPTA** (Heterogeneous Energy Performance Optimisation Algorithm) [94, 95],
- **LBOPA** (Linear Bi-objective Optimisation Algorithm) [96, 97].

The following sections provide more details of these algorithms.

3.3 HEPOPTA Algorithm

HEPOPTA (Heterogeneous Energy Performance Optimisation Algorithm) addresses the optimisation of data-parallel applications with a dual focus on performance and energy efficiency through workload distribution. It utilises input performance and dynamic energy profiles of processors, represented as arbitrary discrete functions of workload size, to generate discrete Pareto-optimal

solutions. By leveraging branch-and-bound and dynamic programming techniques, the algorithm efficiently computes these solutions with polynomial time complexity.

Key Components of HEPOPTA algorithm are:

- *Input Profiles:* HEPOPTA requires discrete performance and dynamic energy profiles for each processor. These profiles, which represent execution time and energy consumption as functions of workload size, are constructed through system-level measurements.
- *Optimisation Algorithm:* Utilising the provided profiles, HEPOPTA employs an efficient global optimisation algorithm to identify all Pareto-optimal solutions. Each solution determines a workload distribution between heterogeneous processors. In each Pareto-front, there is only one solution achieves load balancing and others make the load partially or even completely imbalanced across computing devices. These imbalanced solutions are overlooked by traditional load-balancing methods.
- *Output Solutions:* The algorithm outputs a set of Pareto-optimal solutions, each specifying how to partition the workload among processors to achieve optimal performance-energy trade-offs.

Further details about this optimisation algorithm are available in [94, 98].

3.4 LBOPA Algorithm

LBOPA (Linear Bi-objective Optimisation Algorithm) solves the bi-objective optimisation problem by constructing continuous piecewise linear Pareto fronts of the optimal solutions in the objective space. The algorithm minimises the maximum of continuous and strictly increasing functions (time functions in this thesis) and the sum of continuous linear increasing functions (dynamic energy functions in this thesis). We will use a simplified version of LBOPA that takes two groups of `p` linear increasing functions `F` and `G`. The algorithm constructs a continuous Pareto front, consisting of `p - 1` linear segments.

Figure 3.1 displays a sample Pareto front built by LBOPA. In this example, the number of processors is considered 4 (`p = 4`). Initially, LBOPA creates the Pareto front by determining the breaking points. The left-most endpoint in the figure represents the solution with minimal execution time. This solution distributes the computing workload between all processors in proportion to their speeds, achieving a load-balanced configuration. The next breakpoint allocates zero workload to the most energy-consuming processor and distributes the whole workload among the remaining processors in proportion to their speeds. The algorithm continues this process for the remaining processors until reaching the right-most endpoint, determining the solution with minimum energy consumption. Except for the left-most breaking point, which achieves minimal execution time with fully balanced workload distribution among all processors, the other endpoints represent partially balanced solutions, where the workload is distributed among active processors in proportion to their speeds, and the most energy-consuming processors are excluded by assigning them zero workloads.

LBOPA is facilitated with a sub-algorithm named `Partition`, which determines workload distribution for any point on the created Pareto front, including the breaking points and the points between them. As demonstrated in [96, 97], the workload distribution for a solution lying between two consecutive breaking points results in an imbalanced allocation of workload across the processors.

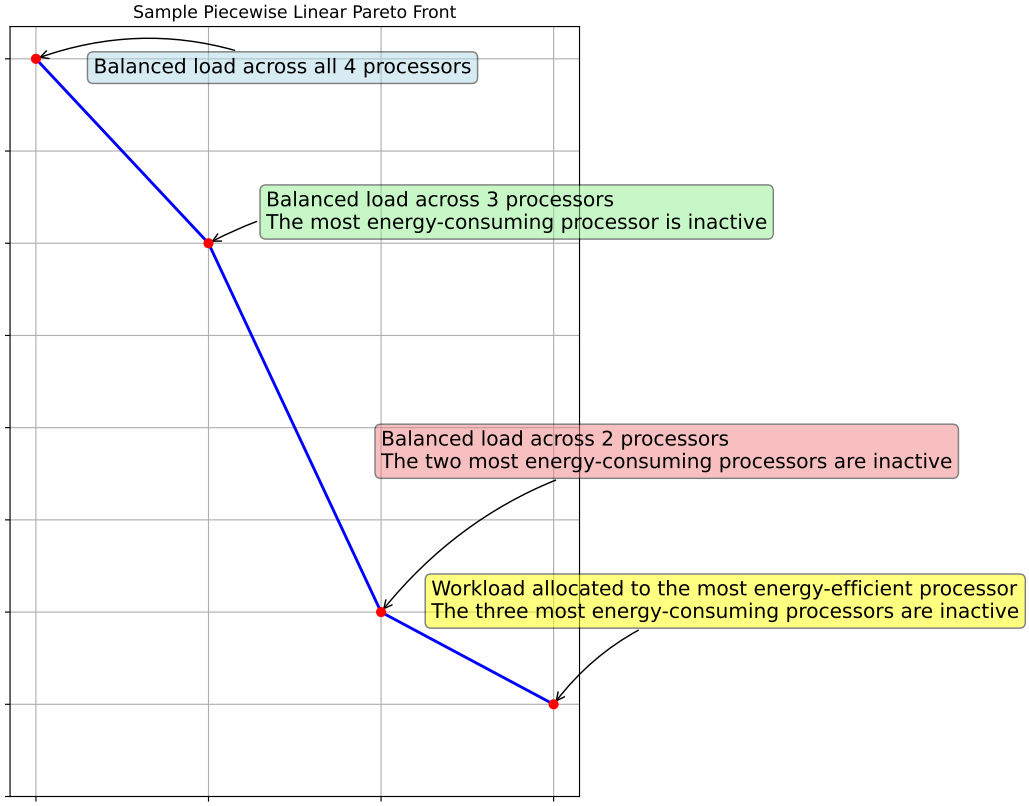

Figure 3.1: A sample continuous piecewise linear Pareto front built by LBOPA for $p = 4$.

Chapter 4

Implementation

In the proposed application, the training process proceeds as follows (Figure 4.1):

- **Data Partitioning:** The input dataset is first partitioned into multiple batches.
- **Batch Distribution:** Based on the performance and energy consumption characteristics of the available resources, these batches are optimally distributed using the EPOPTA and LBOPA algorithms.
- **Local Training on Resources:** Each resource maintains a local copy of the neural network and processes its assigned batches by performing:
    1. Forward propagation
    2. Loss computation
    3. Backward propagation
    4. Local updates of weights and biases
- **Synchronization at the CPU:** After local training, each resource sends its updated weights and biases to the CPU. The CPU synchronizes the results by averaging the received parameters to update the global weights and biases.

- **Parameter Redistribution:** The updated global weights and biases are then shared back with all resources for the next training epoch.
- **Final Output:** After completing all epochs, the final output is a globally trained model.

The following of this chapter will first clarify the challenges of ANNs training on heterogeneous platforms. Next, it will detail the implementation of our application, focusing on kernel and application-level parallelism. Following that, the workload distribution in hybrid ANNs will be discussed. Finally, we will mathematically define our bi-objective optimisation problem and present our proposed solution.

4.1 Challenges in ANNs Training on Heterogeneous Platforms

State-of-the-art machine learning packages are equipped with computational kernels for parallel execution of ANNs training on homogeneous platforms, consisting of an array of identical processing devices. These packages implement parallelism at two levels:

1. The cluster level,
2. The device level.

In *cluster-level* parallelism, the whole training task is evenly distributed across identical devices in the homogeneous cluster. Then, at the *device level*, the workload allocated to each device is further distributed evenly across the identical cores of the device. This approach try to optimise performance and energy consumption, in homogeneous clusters (all processing devices are identical, with similar execution speeds and power consumption characteristics), but this guarantee holds only if the performance profile is linear [12].

Over the past decade, mainstream digital platforms such as HPC servers, data centres, and cloud infrastructures have become increasingly *heterogeneous*. In these platforms, multicore CPUs are integrated with different types

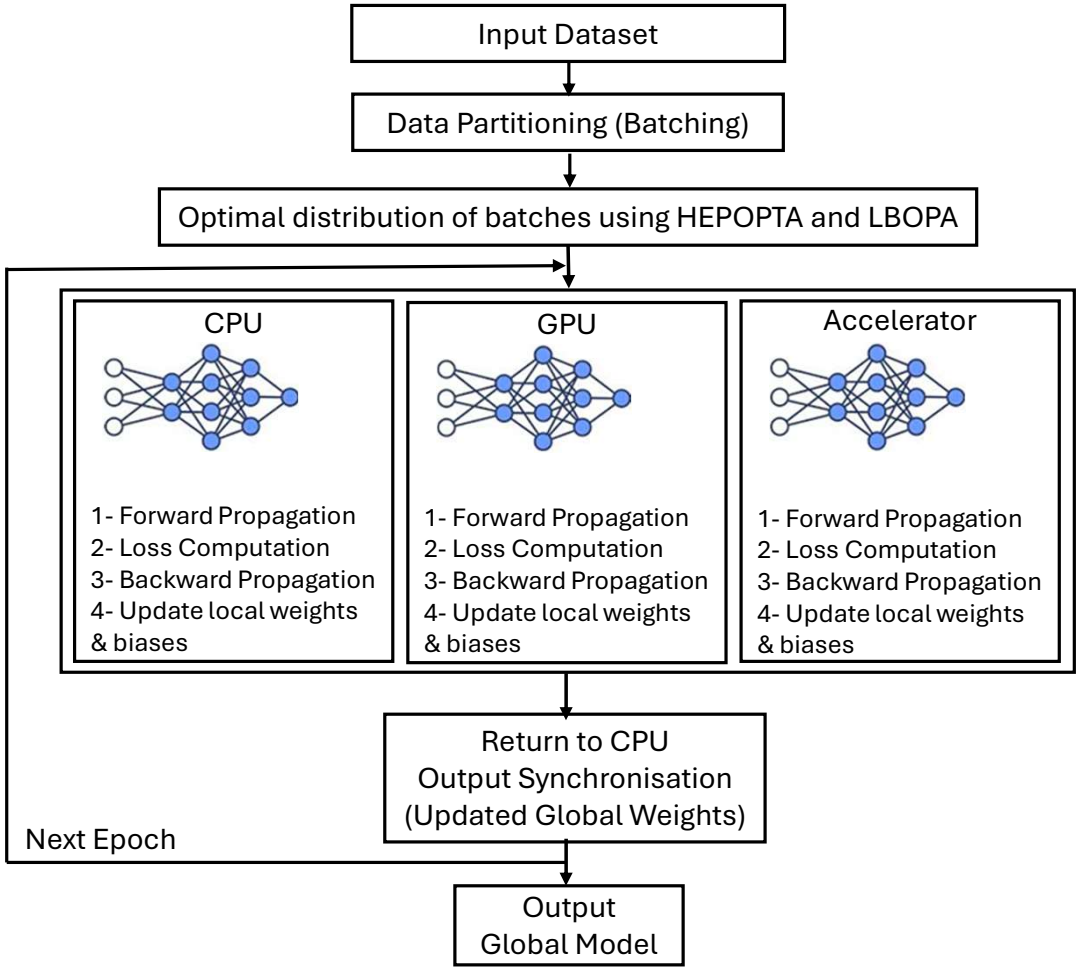

Figure 4.1: Training workflow of the proposed parallel ANN on heterogeneous platforms.

of accelerators, each with distinct performance and energy consumption specifications. This shift from homogeneity to heterogeneity introduces complex challenges, discussed below.

Traditional ANNs packages only support homogeneous execution and cannot perform ANNs training computations simultaneously across different types of devices. This feature limits them to only a subset of the computational resources available in modern heterogeneous clusters. Consequently, these packages underutilise performance and energy-saving potential in modern

platforms, as they are designed for homogeneous environments. This thesis proposes a parallel programming methodology for developing portable hybrid ANNs applications to address this limitation. The proposed methodology enables parallel training of ANNs on hybrid heterogeneous HPC platforms with diverse types of accelerators. The implementation details are explained in Section 4.2.

Furthermore, traditional ANNs packages rely on a straightforward equal workload partitioning approach, which is effective for homogeneous clusters. However, this approach no longer ensures optimal execution time and energy consumption on heterogeneous platforms. This is because heterogeneous platforms comprise diverse computational devices with varying execution speeds and energy consumption rates. To address this challenge, we introduce a workload distribution strategy to minimise execution time and energy consumption within the proposed programming framework on heterogeneous platforms. We formulate this optimisation problem mathematically and apply two solution methods to obtain optimal workload distributions for performance and energy consumption. Section 4.4 covers the proposed optimisation approach.

4.2 Hybrid ANNs Implementation

This section outlines our methodology for designing parallel and portable ANNs applications capable of training ANNs on hybrid heterogeneous HPC platforms. The methodology leverages various types of processing devices to enable the simultaneous utilisation of multiple computing resources for training. It employs a hybrid programming model with two levels of parallelism:

1. Kernel-level,

2. Application-level.

At the *kernel level*, parallel training computations are implemented for CPUs and accelerators using multi-threaded programming models. At the *ap-*

plication level, host CPU threads are created to enable the simultaneous execution of parallel kernels across multiple devices on a heterogeneous server.

Additionally, in this section, we implement a hybrid parallel fully connected multilayer perceptron ANNs application based on the hybrid parallel methodology.

4.2.1 Kernel-level Parallelism

The proposed methodology utilises two main types of parallel multithreaded kernels at the base level of parallelism: one for multicore CPUs and another for accelerators. Parallel CPU kernels are implemented using OpenMP, a directive-based standard API for parallel programming on multicores and shared-memory multi-processors. To develop parallel and portable accelerator kernels, we use OpenACC [16], another directive-based programming standard similar to OpenMP, designed for offloading parallel computations onto accelerators. OpenACC is not restricted to a specific device and supports a wide range of accelerators, including GPUs, FPGAs, DSPs, and Intel Xeon Phi.

It is worth mentioning that in many previous studies, researchers have used cuBLAS, a highly optimised NVIDIA math library for dense linear algebra on GPUs, to parallelise Fully Connected ANNs with computational efficiency [1, 23]. In this research, however, OpenACC was chosen over cuBLAS because the goal was to develop a portable, parallel application that could be extended to other types of ANNs and run on different platforms. For networks such as CNNs, the performance bottlenecks are dominated by custom loop-based computations rather than dense linear algebra kernels. While cuBLAS provides highly optimised GPU implementations of BLAS routines, it is limited in scope and requires significant code refactoring and explicit GPU memory management. In contrast, OpenACC provides performance portability across heterogeneous systems by supporting NVIDIA GPUs, multicore CPUs, and, depending on compiler support, other accelerator architectures such as AMD GPUs. OpenACC enables directive-based acceleration of large portions of the existing code with minimal restructuring, improving development productivity,

maintainability, and portability across heterogeneous systems[99, 100, 101]. This approach delivers broader application-level speedups, rather than optimising isolated computational kernels, making it a more effective and sustainable solution for this project.

Now, we discuss in detail the implementation of training kernels for a parallel fully connected multilayer perceptron ANNs application using the proposed methodology. We utilise a simple fully connected neural network serial implementation in C programming language [102] as the foundation for our kernels. This particular implementation is selected due to the following features:

- Supports the construction of ANNs with a configurable number of hidden layers,
- Does not use any vector or matrix libraries, offering better opportunities for code understanding and optimisation of parallelisation,
- Allows batch training for an arbitrary number of epochs,
- Utilises Gradient Descent (GD) for training, a widely used iterative algorithm for optimising ANNs weights,
- Includes various activation and error calculation functions within the implementation.

Algorithm 1 presents the parallel kernel for training the neural network's weights and biases, employing OpenMP programming standard for parallel execution on multicore CPUs. The kernel starts as a single thread executing the outer `for` loop, which iterates over the batches of the input. At each iteration of this loop, the `#pragma omp parallel for` directive in line 4 creates a team of CPU threads with the original thread as a master of the team and distributes iterations of the inner `for` loop between these threads, this way parallelising the forward pass and backpropagation computations for samples in the batch. The forward pass step updates all activations across the input, hidden, and output layers per each batch. It is followed by a backward pass computation to calculate the errors and incremental changes in the network

weights. Once all samples in the batch are processed, the inner `for` loop is finished, the team of threads is destroyed, and the execution is continued by the original single thread, which calls functions `sum_values_and_deltas_CPU()` and `update_weights_and_biases_CPU()` to update the weights and biases.

Within the helper functions (`compute_forward_pass_CPU()` and others), computations iterating over nodes of the ANN layers are further parallelised using `#pragma omp parallel for` directives. If the batch size is sufficiently large (e.g., greater than the number of CPU cores), this additional parallelism is not necessary in functions `compute_forward_pass_CPU()` and `calculate_errors_and_deltas_CPU()`, as all CPU cores will be already utilised by the team of threads processing samples of the batch in parallel. However, it will help accelerate the execution of functions `sum_values_and_deltas_CPU()` and `update_weights_and_biases_CPU()`.

In our experiments, we use the NVIDIA C compiler (NVC) compiler, which does not support the OpenMP nested parallelism. Therefore, it will serialise the execution of each function call within the parallel `for` loop and parallelise the execution of the function calls updating weights and biases. Given we use large batch sizes in our experiments, this compilation mode will fully utilise all available CPU cores.

The two input-output arrays `weights` and `biases` are passed into the kernel by reference. Consequently, they hold the caller's weights and biases when the training kernel is invoked, and any modifications made to the arrays within the kernel will directly affect the local weights array in the caller's scope.

Algorithm 2 presents the parallel multi-threaded kernel for training the neural network on accelerators. This kernel utilises the OpenACC programming standard to offload parallel execution onto accelerators. Similar to the CPU kernel, the forward pass and backpropagation computations are parallelised over samples of the batch, but in this kernel the `#pragma acc parallel for` directive (Line 6) is used to implement multithreading. Unlike the CPU kernel, the additional parallelism in all helper functions is necessary. It helps utilise the large number of accelerator cores (thousands for modern GPUs). In our experiments, we use the NVC compiler, which supports the nested OpenACC parallelism.

Algorithm 1: Implementation of Training Kernel for CPUs

1 **Function** `Training_Kernel_CPU(batches, num_batches, batch_size, weights, biases);`

Input :

- *batches*: A 1D array of vectors where each vector contains a training sample and its corresponding true answer.
- *num_batches*: The number of batches existing in *batches* for training.
- *batch_size*: The number of samples contained in each batch.

In-Out :

- *weights*: A 1D array of 2D matrices, where the i-th matrix ($0 \leq i < num\_layers$) contains a local copy of the network's weights for the i-th layer. This array is passed into the kernel by reference. Consequently, it holds the caller's weights when the training kernel is invoked, and any modifications made to this array within the kernel will directly affect the local weights array in the caller's scope.
- *biases*: A 2D array, where the i-th column ($0 \leq i < num\_layers$) contains a local copy of the network's biases for the i-th layer. This array is passed into the kernel by reference. Consequently, it holds the caller's biases when the training kernel is invoked, and any modifications made to this array within the kernel will directly affect the local bias array in the caller's scope.

2 **begin**
3 **for** $b = 0;\ b < num\_batches;\ b++$ **do**
4 #pragma omp parallel for
5 **for** $s = 0;\ s < batch\_size;\ s++$ **do**
`// Forward pass across input, hidden, and output layers to compute all activations`
6 compute_forward_pass_CPU(batches[b * batch_size + s], weights, biases);
`// Backpropagation and error calculation`
7 calculate_errors_and_deltas_CPU(batches[b * batch_size + s], weights, biases);
8 **end**
`// Sum hidden layer values and deltas`
9 sum_values_and_deltas_CPU(···);
`// Update weights and biases`
10 update_weights_and_biases_CPU(...);
11 **end**
12 **end**

In heterogeneous servers, accelerators have their own memory spaces, separate from the host CPU's main memory (Random Access Memory (RAM)). Unlike the CPU kernel, data required for the accelerator kernel must be explicitly transferred from the host's main memory to the accelerator's local memory before any computation begins. In addition, results must be transferred back to the host after the computation is complete.

To manage these data transfers, we employ the `#pragma acc data` directive with two clauses `copyin` and `copy`, as shown in line 3 of Algorithm 2. The `copyin` clause allocates memory on the accelerator and transfers data from the caller's host memory to the accelerator upon entering its block. We use this clause to transfer the `batches`, `num_batches`, and `batch_size` input variables because these variables are not modified within the kernel. Consequently, they do not need to be copied back to the caller's host memory upon exiting the block. However, since the two input-output arrays `weights` and `biases` are updated within the kernel, we use the `copy` clause to transfer them. This clause allocates memory on the accelerator, copies data from the caller's host memory to the accelerator upon entering the block, and then returns the updated data back to the caller's host memory upon exiting the block. Consequently, the arrays `weights` and `biases` contain the caller's initial weights and biases when the training kernel is offloaded, and the updated values are copied back to the caller's scope upon kernel completion. This approach ensures data consistency between the computations within the kernel and the caller's data.

4.2.2 Application-level Parallelism

At the second level of parallelism in the proposed methodology, we employ Pthreads, a standard programming model that enables the simultaneous execution of multiple tasks, to integrate the parallel CPU kernel(s) with the parallel and portable accelerator kernel(s) into a unified hybrid application. This approach spawns a host thread for each device in a heterogeneous server and invokes the training kernels (threads) on these devices in parallel, facilitating the simultaneous execution of parallel kernels across the heterogeneous plat-

Algorithm 2: Implementation of Training Kernel for Accelerators

1 **Function** `Training_Kernel_ACC(batches, num_batches, batch_size, weights, biases);`

Input :

- *batches*: A 1D array of vectors where each vector contains a training sample and its corresponding true answer.
- *num_batches*: The number of batches existing in *batches* for training.
- *batch_size*: The number of samples contained in each batch.

In-Out :

- *weights*: A 1D array of 2D matrices, where the i-th matrix ($0 \leq i < num\_layers$) contains a local copy of the network's weights for the i-th layer. This array holds the caller's weights when the training kernel is invoked, and upon kernel completion, the updated weights are copied back to the caller's local weights array.
- *biases*: A 2D array, where the i-th column ($0 \leq i < num\_layers$) contains a local copy of the network's biases for the i-th layer. This array holds the caller's biases when the training kernel is invoked, and upon kernel completion, the updated biases are copied back to the caller's local bias array.

2 **begin**

`// Data Copy for Accelerators`

3 #pragma acc data copyin (batches, num_batches, batch_size) copy (weights, biases);

4 **begin**

5 **for** $b = 0;\ b < num\_batches;\ b++$ **do**

6 #pragma acc parallel for;

7 **for** $s = 0;\ s < batch\_size;\ s++$ **do**

`// Forward pass across input, hidden, and output layers to compute all activations`

8 compute_forward_pass_ACC(batches[b * batch_size + s], weights, biases);

`// Backpropagation and error calculation`

9 calculate_errors_and_deltas_ACC(batches[b * batch_size + s], weights, biases);

10 **end**

`// Sum hidden layer values and deltas`

11 sum_values_and_deltas_ACC($\cdots$);

`// Update weights and biases`

12 update_weights_and_biases_ACC($\ldots$);

13 **end**

14 **end**

15 **end**

form. At this level of parallelism, Pthreads is used instead of OpenMP because of limitations in mainstream compilers. Some compilers, such as NVC, do not support OpenMP nested parallelism, which will result in serial execution of CPU kernels. On the other hand, the GCC compiler does not compile projects targeting both OpenMP and OpenACC. These limitations make Pthreads the only practical option for portable integration of OpenMP and OpenACC kernels.

Algorithm 3 displays the pseudocode of the parallel ANN application for training a neural network in parallel on a heterogeneous platform consisting of `p` devices, one multicore CPU and `p-1` accelerators. The Pthreads library is employed to invoke the training kernels on processing devices simultaneously.

After initialising the weights and biases matrices (Line 3), the function `distribute_batches` is called to create total batches from the input dataset (`input_data`) and distribute them among the `p` devices (Line 4). The function returns a pair $(\texttt{batches}, \texttt{num\_batches})$, consisting of two arrays of size `p`. For a given device $i \in \{0, 1, \cdots, p-1\}$, the pair $(\texttt{batches}[i], \texttt{num\_batches}[i])$ represents the subset of the input dataset allocated to the i-th device, where `batches`[i] points to the first sample of the first batch allocated to the device, and `num_batches`[i] is the number of batches assigned to the device.

Following the batches distribution, the main training loop iterates over epochs, and for each epoch, the CPU and accelerators host threads are spawned using the non-blocking `pthread_create` function to train their local weights and biases with the Stochastic Gradient Descent algorithm (Lines 7 and 10). Since Pthreads passes parameters to the kernel by reference, any modifications made to the training kernels' `weights` and `biases` parameters will also affect the passed arguments `weights_local` and `biases_local`.

Once all threads terminate (Line 13), the local weights and biases computed by the `p` kernels are averaged to obtain global weights and biases matrices (Lines 15 and 16). These global matrices are used in the next epoch, ensuring the training process benefits from the up-to-date network parameters computed by the `p` devices.

The complete source code of the hybrid ANNs is available at [103].

As discussed in Section 4.1, one of the challenging steps in Algorithm 3 is

Algorithm 3: Hybrid ANNs Training Application

1 **Function** `Hybrid_ANN_Training(input_data, batch_size, p, weights, biases);`

Input :

- *input_data*: A 1D array of vectors where each vector contains a training sample and its corresponding true answer.
- *batch_size*: The number of samples in each batch.
- p: The number of devices, including 1 CPU and p-1 accelerators, for parallel execution.

Output:

- *weights*: A 1D array of 2D matrices, where the i-th matrix ($0 \leq i < num\_layers$) contains the trained network's weights for the i-th layer.
- *biases*: A 2D array, where the i-th column ($0 \leq i < num\_layers$) contains the trained network's biases for the i-th layer.

2 **begin**

`// Initialize weights and biases matrices`

3 init_weights_biases(weights, biases);

`// Create batches and distribute them between CPU and accelerators`

4 (batches, num_batches) = distribute_batches (input_data, batch_size, p);

`// Loop for epochs`

5 **for** $e = 0;\ e < num\_epochs;\ e++$ **do**

`// Create local copies of global weights and biases arrays`

6 weights_local[0] = weights; biases_local[0] = biases;

`// Execute CPU kernel on a multicore CPU`

7 pthread_create(thread_id[0], Training_Kernel_CPU(batches[0], num_batches[0], batch_size, weights_local[0], biases_local[0]));

`// Execute accelerator kernels on` $p-1$ `devices`

8 **for** $d = 1;\ d < p;\ d++$ **do**

`// Create local copies of global weights and biases arrays`

9 weights_local[d] = weights; biases_local[d] = biases;

10 pthread_create(thread_id[d], Training_Kernel_ACC(batches[d], num_batches[d], batch_size, weights_local[d], biases_local[d]));

11 **end**

`// Wait for all` p `threads to finish`

12 **for** $d = 0;\ d < p;\ d++$ **do**

13 pthread_join(thread_id[d]);

14 **end**

`// Average all local weights and biases to obtain global weights and biases`

15 weights = average_local_weights(weights_local);

16 biases = average_local_biases(biases_local);

`// Calculate error using averaged weights and biases`

17 calculate_error(...);

18 **end**

19 **return** $(weights, biases)$;

20 **end**

to distribute total batches among computational devices (Line 4). In Section 4.3, we explore how workload distribution, specifically dividing total batches among computational devices, affects execution time (performance) and energy (electricity) consumption during the training phase of the Hybrid ANNs algorithm. Then, we propose a method to obtain optimal trade-offs between these two optimisation objectives via workload partitioning.

4.3 Workload distribution in Hybrid ANNs

Although existing machine learning packages, such as TensorFlow, support different computing devices, including CPUs and Nvidia GPUs, they are designed for homogeneous execution. These packages cannot be configured for simultaneous execution on hybrid platforms to utilise the processing power of all available resources. Consequently, these packages fail to distribute training workloads effectively between processing devices, hindering optimal performance and energy efficiency on heterogeneous systems.

In this section, we study the impact of workload (batches) distribution on the performance and energy consumption of our implemented parallel and portable ANNs application. This case study is conducted on HCLServer01, a heterogeneous platform which includes three different computing devices: a multi-core CPU, an Nvidia K40c GPU, and an Nvidia P100 Peripheral Component Interconnect Express (PCIe) GPU. The technical specifications of the platform are provided in the next chapter, Section 5.2.

In this case study, the implemented ANNs application is trained on the Modified National Institute of Standards and Technology database (MNIST) dataset [104] for handwritten digit recognition (Figure 4.2). The network consists of an input layer with 784 neurons (representing pixel values), connected to five fully connected hidden layers of sizes $\{2500, 2000, 1500, 1000, 500\}$, and a single neuron output layer. This is one of the best-performing topologies for MNIST handwritten digit recognition [104]. The MNIST dataset, containing 60,000 samples, is used for training, and the training batch size and epoch number are respectively set to 600 and 1, resulting in a total workload of 100

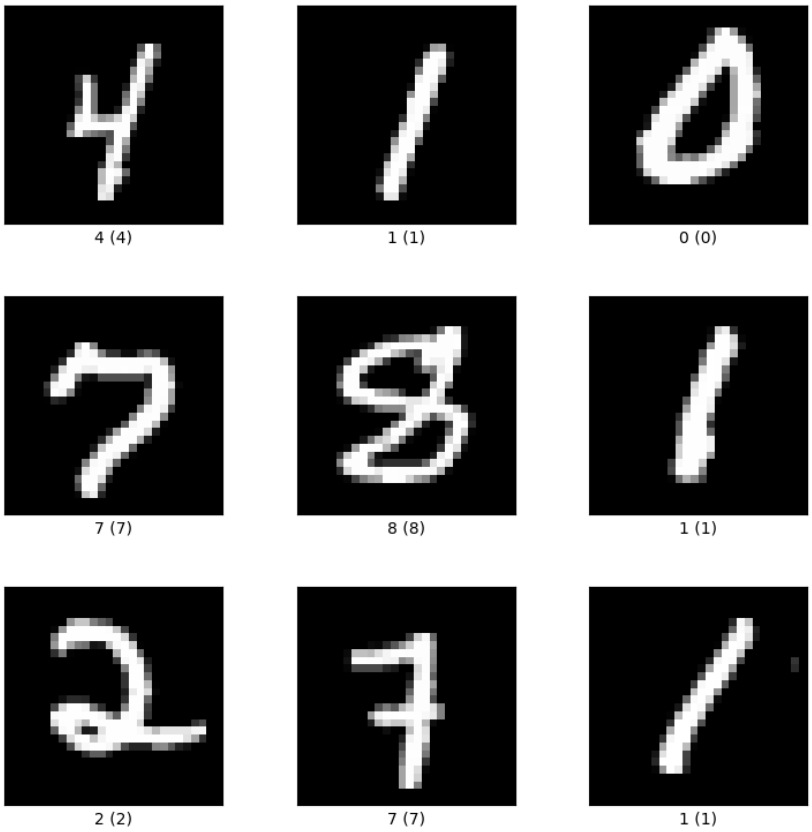

Figure 4.2: Some samples of MNIST dataset.

batches. In this experiment, the parallel ANNs application is executed with three different distributions of batches across the computational resources, and the training time and dynamic energy consumption of each parallel execution are accurately measured. Our methodology for precisely measuring execution time and energy consumption of parallel applications on HPC platforms is detailed in the next chapter, Section 5.1.

Figure 4.3 displays the parallel execution time and energy consumption for the parallel ANNs under three different workload distributions. Comparing the first two solutions, we can see that a 20% performance degradation leads

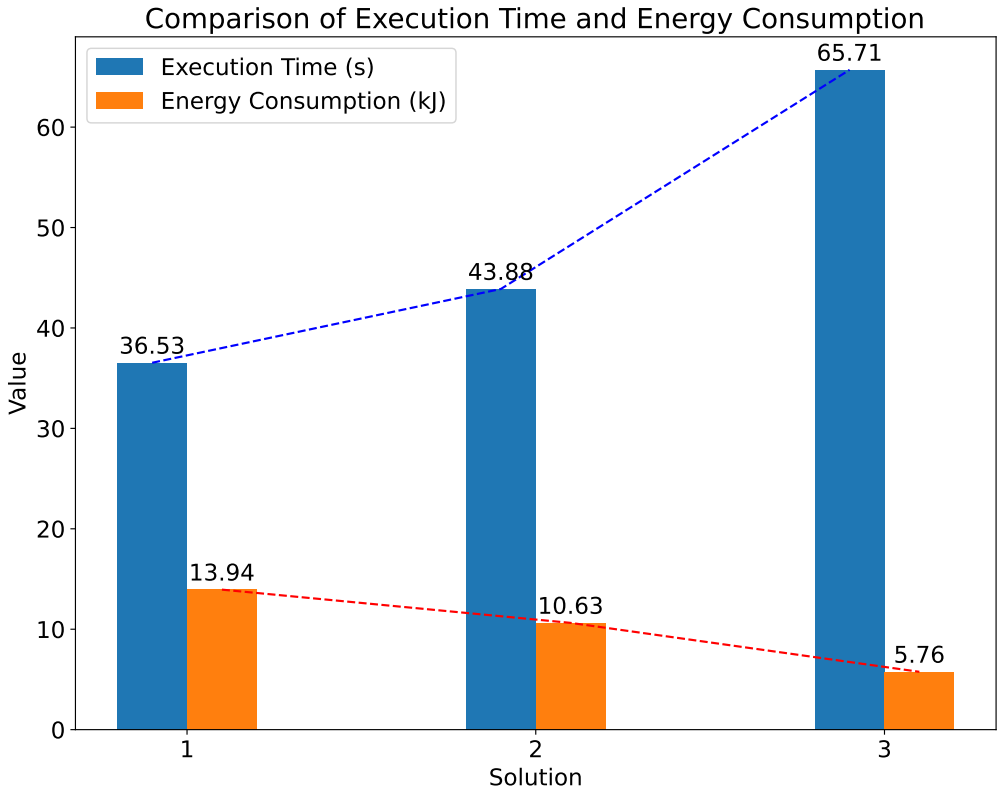

Figure 4.3: A case study comparing the execution time and energy consumption of the hybrid ANNs for three different batches distributions.

to around 31% energy savings. In addition, switching from the first solution to the third one results in a performance drop of approximately 80% and an energy efficiency improvement of 142%. Based on the obtained results, it can be concluded that:

- Execution time and energy consumption are two conflicting objectives for optimisation, meaning that improving performance typically degrades energy efficiency and vice versa.
- Batches (workload) distribution significantly affects the performance and energy consumption of the Hybrid ANNs and can be considered as a key decision variable for optimising the performance and energy efficiency.

These valuable observations form the basis of our method for optimizing

the execution of hybrid ANNs with respect to two key objectives: execution time and energy consumption. However, since our experimental evaluation is limited, we conducted an extensive review of related work to ensure that these observations are consistent with and supported by findings reported by other researchers.

- **Trade-off between execution time and energy consumption:** Khaleghzadeh et al. [96, 97, 94, 93] demonstrated in their studies that execution time and energy consumption are inherently conflicting objectives. Improving one often leads to the degradation of the other. To address this trade-off, they proposed a bi-objective optimization algorithm that jointly considers both metrics, enabling a balanced solution depending on system constraints and application requirements. Their findings confirm the necessity of treating execution time and energy consumption as coupled objectives rather than optimising them independently [96, 97, 94, 93].
- **Impact of batch distribution on performance and energy efficiency:** The effect of batch distribution on both system performance and energy consumption has also been widely observed in prior research. Several studies have shown that the way computational workloads and data batches are distributed across resources significantly influences overall execution time [105, 106, 107]. Inefficient batch allocation may lead to resource underutilisation, increased synchronization overhead, and unnecessary energy expenditure, whereas optimised distribution strategies can substantially improve both performance and energy efficiency.

These observations lead us to two crucial questions:

1. Is there a methodology to optimise the hybrid ANNs execution for the two objectives, execution time and energy consumption, via batches (workload) distribution?
2. Given the conflicting nature of performance and energy consumption in this application, is there a methodology to obtain trade-off solutions between these two optimisation objectives?

To address these questions, in Section 4.4, we mathematically formulate the bi-objective optimisation of the hybrid ANNs application as a min-max-min-sum problem and then utilise two algorithms, HEPOPTA [94] and LBOPA [96] to solve this optimisation problem.

4.4 Bi-objective Optimisation Problem

As mentioned in Section 4.3, the two objectives – performance and dynamic energy, are conflicting, i.e., optimising one objective negatively impacts the other. Therefore, the performance and dynamic energy consumption optimisation should be moved from a single objective to a bi-objective optimisation problem.

Consider a dataset of size $S \in \mathbb{N}^+$ and a batch size of $b \in \mathbb{N}^+$, resulting in a total number of batches $B = \frac{S}{b}$ for a one-epoch ANNs training task. The training process is executed on a platform with $p \in \mathbb{N}^+$ heterogeneous processors. We formulate the problem for a single-epoch training. It does not result in losing generality as the computational process remains consistent across all epochs of a training task.

The bi-objective optimisation problem for the training task can be formulated as follows:

$$minimise_X \{T(X),\ E(X)\} \tag{4.1}$$

where $T(X) : \mathbb{N}^p \rightarrow \mathbb{R}$ and $E(X) : \mathbb{N}^p \rightarrow \mathbb{R}$ represent the two objectives, execution time and dynamic energy consumption, for processing all B batches, respectively. The decision vector $X = \{x_0, x_1, \cdots, x_{p-1}\}$ determines the distribution of B batches among p processors such that $B = \sum_{i=0}^{p-1} x_i$, and $x_i \in \mathbb{N}$ denotes the number of samples assigned to processor P_i. In this context, $X \in \mathcal{F}$, where the set $\mathcal{F} \subset \mathbb{N}^p$ represents all feasible distributions of B among the p processors. As illustrated in Figure 4.4, Equation 4.1 maps the feasible decision region $\mathcal{F}$ to a feasible objective region $T(X) \times E(X) \subset \mathbb{R}^2$.

Given two sets of functions $F(X) = \{f_0(x_0), f_1(x_1), \cdots, f_{p-1}(x_{p-1})\}$ and $G(X) = \{g_0(x_0), g_1(x_1), \cdots, g_{p-1}(x_{p-1})\}$ where $f_i(x) \in \mathbb{R}$ and $g_i(x) \in \mathbb{R}$

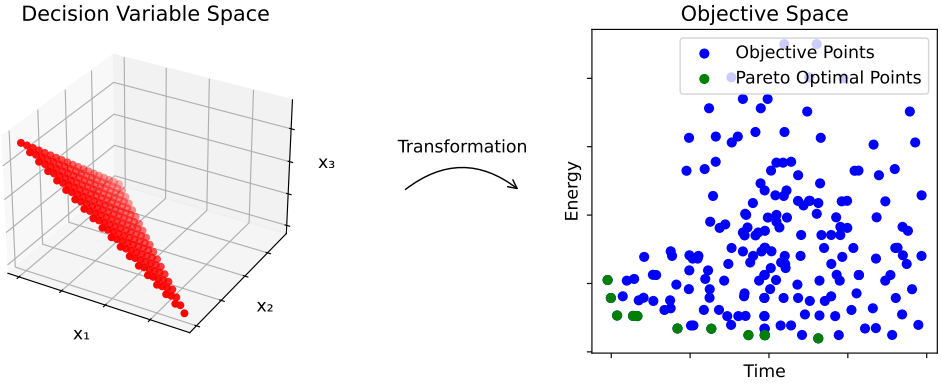

Figure 4.4: An example showing the set of decision variable vectors $\mathcal{F}$, the set of objective vectors $\mathcal{Z}$, and Pareto optimal solutions for $p = 3$.

determine the execution time (time function) and dynamic energy consumption (energy function) of processing x batches of size b on the i-th processor (P_i) respectively, the parallel execution time of the training task will be $T(X) = \max_{i=0}^{p-1} f_i(x_i)$, and its dynamic energy consumption will be $E(X) = \sum_{i=0}^{p-1} g_i(x_i)$. Therefore, Equation 4.1 can be expressed as:

$$minimise_X \ \{\max_{i=0}^{p-1} f_i(x_i),\ \sum_{i=0}^{p-1} g_i(x_i)\}$$

$$\begin{gathered} \text{where} \\ X = \{x_0, x_1, \cdots, x_{p-1}\} \\ \& \\ B = \sum_{i=0}^{p-1} x_i \end{gathered} \tag{4.2}$$

Equation 4.2 mathematically formulates the bi-objective optimisation of the hybrid ANNs application as a min-max-min-sum problem. This formulation aims to minimise both objective functions, `T(X)` and `E(X)`, simultaneously, via workload (batches) distribution. However, since these two objectives are conflicting, it is not feasible to find a single solution that is optimal for both objectives simultaneously. Therefore, to solve Equation 4.2, we need to obtain Pareto-optimal solutions, which include a set of trade-off solutions between the two objectives. There are two main categories of algorithms for obtaining Pareto-optimal solutions:

1. Heuristic methods, such as Genetic Algorithms,

2. Exact algorithms, such as LBOPA [96] and HEPOPTA [94].

In the next chapter, sections 5.2 and 5.3, we will use LBOPA and HEPOPTA to obtain Pareto optimal solution sets for the hybrid ANNs application. These two algorithms were explained in the previous chapter, 3.3 and 3.4 sections. In summary, these algorithms work as follows:

- **LBOPA**: This algorithm addresses the bi-objective optimisation problem by constructing continuous piecewise linear Pareto fronts of optimal solutions in the objective space. It minimizes both the maximum of continuous, strictly increasing functions (time functions in this thesis) and the sum of continuous, linearly increasing functions (dynamic energy functions in this thesis). A simplified version of LBOPA will be used, which considers two groups of `p` linear increasing functions, `F` and `G`. The algorithm constructs a continuous Pareto front composed of `p - 1` linear segments [96, 97].

- **HEPOPTA**: This algorithm solves optimisation problems of data-parallel applications for two objectives performance and energy through workload distribution. It takes input performance and dynamic energy profiles of processors, represented as arbitrary discrete functions of workload size, to generate discrete Pareto-optimal solutions. The algorithm obtains these solutions in polynomial time complexity by employing branch-and-bound and dynamic programming methods [94].

Chapter 5

Experiments and Results

In this chapter, the performance and energy consumption of the implemented hybrid ANNs are studied on several heterogeneous servers with different computational devices.

First, we explain our methodology for accurately measuring the execution time and energy consumption of the hybrid ANNs application on heterogeneous servers. This approach is essential for constructing the application's execution time (F) and energy consumption (G) functions. Then, to study the optimality of our hybrid ANNs application for performance and energy, the constructed functions are used by LBOPA and HEPOPTA to obtain the Pareto-optimal solutions for the application on HCLServer01, a heterogeneous server consisting of three different computing devices. Finally, we conduct simulated experiments on hypothetical heterogeneous platforms with varying numbers of accelerators.

5.1 Constructing Performance and Dynamic Energy Functions

This section describes our methodology for accurately measuring the execution time and energy consumption of the hybrid ANNs application on heterogeneous servers. These measurements are used to construct the hybrid ANNs application's execution time (F) and energy consumption (G) functions that vary

with the number of batches.

5.1.1 Performance Functions

To create execution time (performance) functions, we need to accurately measure the execution time of each kernel in the hybrid ANNs application when training a specified number of batches for a given batch size. To ensure that our functions are reproducible and to minimise resource sharing between kernels, we employ the concept of *abstract processors* [108]. An abstract processor represents computing resources involved in the execution of an individual kernel of a hybrid application. For example, consider HCLServer01, a heterogeneous server consisting of three computing devices: a 22-core Intel Xeon Gold CPU, an Nvidia K40c GPU, and an Nvidia P100 PCIe GPU. The server comprises three abstract processors as follows:

1. **CPU_Xeon**: 20 out of 22 physical cores of the Intel Xeon multicore CPU.
2. **GPU_K40c**: 1 core out of the remaining 2 CPU cores, the Nvidia K40c computational cores, and the Peripheral Component Interconnect (PCI) communication link between them.
3. **GPU_P100**: 1 remaining CPU core, the Nvidia P100 computational cores, and the PCI communication link connecting them.

We simultaneously measure the execution time of all loosely coupled abstract processors on the platform to account for the impact of thread coscheduling and resource sharing on the times measured. Figure 5.1 presents the time functions of the application for the MNIST training dataset [104], which consists of 60,000 samples. The time functions show the training time for different numbers of batches of size 400 on HCLServer01 with a step size of 5. For each data point in these functions, the measurements are repeated until the sample means of all three kernels running on the abstract processors fall within a 95 percent confidence interval.

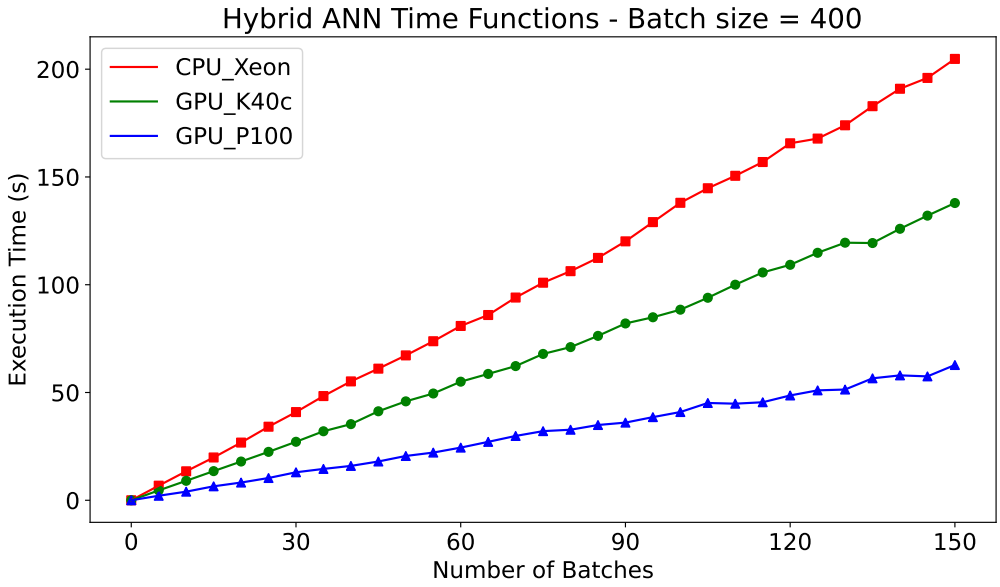

Figure 5.1: Time functions of the hybrid ANNs executing on HCLServer01. The application trains a multi-layer perceptron ANNs using the MNIST dataset, which consists of 60,000 samples, for a given batch size of 400.

5.1.2 Dynamic Energy Functions

Energy consumption in computing devices is classified into dynamic energy and static energy. From an application perspective, static energy consumption refers to the energy consumed by the platform when the application is not executing. The dynamic energy consumed during application execution is calculated by subtracting the static energy from the platform's total energy consumption. In other words, dynamic energy represents the energy consumed solely to run the application.

Energy consumption during an application's execution can be measured using three main approaches:

1. On-chip power sensors,
2. Software-based energy predictive models,
3. Physical measurement using external power meters.

According to [109], the first two approaches do not offer the same level of accuracy as physical measurement with external power meters. Consequently, we employ the additivity methodology [94], which utilises system-level power measurements via power meters to accurately model the energy consumption of the *abstract* processors within the hybrid ANNs application. According to this methodology, the dynamic energy consumption of the hybrid ANNs application is the sum of the energy consumed by each of its kernels when executed individually.

Figure 5.2 illustrates the dynamic energy profiles of the application as a function of the number of batches. Consistent with the performance functions, we used the MNIST training dataset and measured the dynamic energy consumption of the application for a batch size of 400. The fans for computational devices are set to full speed before launching the experiments to eliminate their contribution to dynamic energy consumption. By doing so, the energy consumed by the fans is accounted for as part of the static energy, rather than being included in the dynamic energy consumption.

5.1.3 Discussion

Using the measurement methodology discussed earlier, we experimentally created the ANNs application's execution time and dynamic energy functions for different batch sizes. All the obtained functions exhibited a linear increasing trend, similar to those observed in Figures 5.1 and 5.2. This is because the computations for processing each batch are similar, resulting in nearly identical time and energy consumption per batch. Consequently, the execution time and dynamic energy consumption for processing k batches of a given size are approximately k times those required for processing a single batch of the same size.

Considering the linear relationship between execution time and energy consumption with the number of batches, for a given batch size, it is unnecessary to execute the ANNs application for different numbers of batches to construct its $\mathtt{F}$ and $\mathtt{G}$ functions. Instead, the base execution time and base energy consumption per device can be obtained by measuring the time and

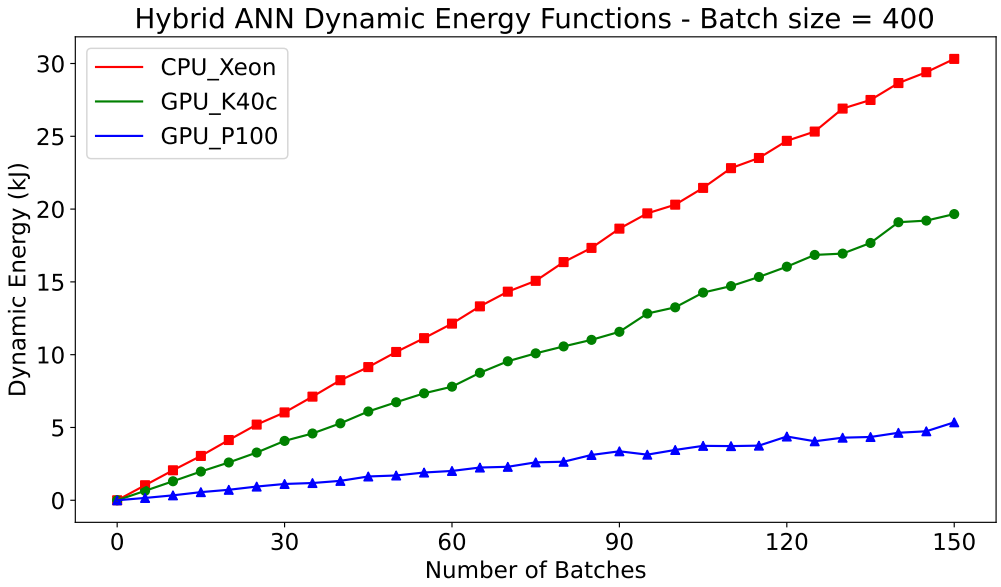

Figure 5.2: Dynamic energy functions of the Hybrid ANNs executing on HCLServer01. The application trains a multi-layer perceptron ANNs using the MNIST dataset, which consists of 60,000 samples, for a given batch size of 400.

energy consumption for processing a single batch of the given batch size. Then, these base measurements can be utilised to approximate the execution time and energy consumption of the application for larger numbers of batches with high accuracy.

Figures 5.3 and 5.4 present the base execution time and base energy consumption of the application for batch sizes ranging from 25 to 7,500, with a step size of 5. Only batch sizes that evenly divide the total samples of 60,000 are considered, and we use the MNIST training dataset for training. Each data point in these time (energy) functions represents the execution time (energy consumption) required to process a single batch of a given size.

By utilising these base functions, and considering their linear trend, the execution time and dynamic energy functions for the hybrid ANNs application can be generated for any batch size.

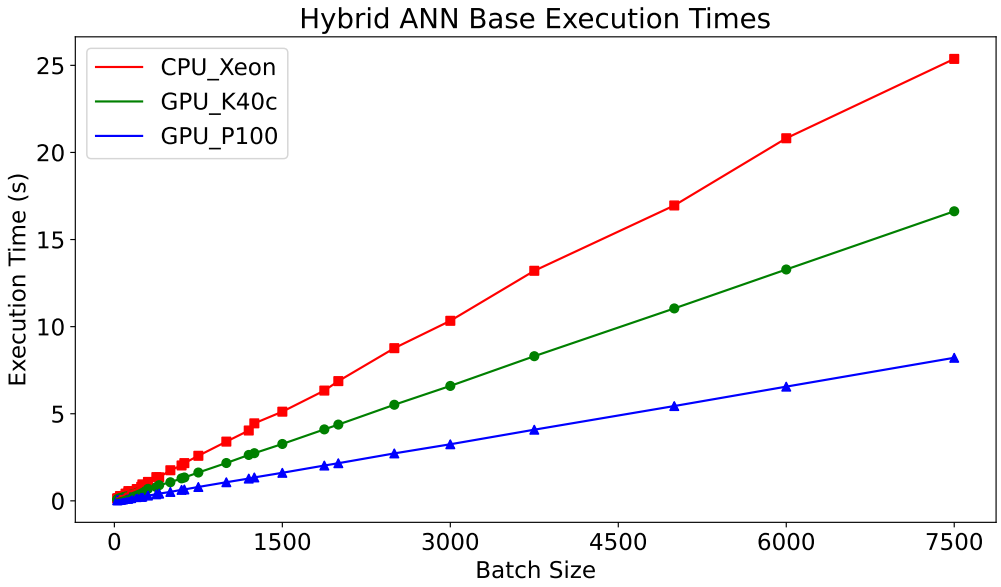

Figure 5.3: Base execution times for training a multi-layer perceptron ANNs using a single batch whose sizes range from 25 to 7,500 on HCLServer01. Only batch sizes that evenly divide the total 60,000 samples are considered.

5.2 Analysing Hybrid ANNs on a Real Platform

In this section, we examine the improvements in performance and reductions in dynamic energy consumption of the hybrid ANNs application achieved through workload (batches) distribution. These experiments are conducted on HCLServer01, comprising three devices ($p = 3$), a CPU_Xeon, a GPU_K40c and a GPU_P100, whose specifications are detailed in Table 5.1.

As discussed in Section 5.1, we conclude that the execution time and energy consumption of the application exhibit linear behaviour with the number of batches. As a result, the performance and energy consumption of the ANNs application can be accurately approximated using continuous linear increasing functions of the number of batches. This observation supports the application of the LBOPA algorithm for batches partitioning to obtain continuous Pareto front solutions for the two optimisation objectives of execution time and dynamic energy. We also generate discrete linear performance and energy functions with a step size of one. These discretised functions are then input to

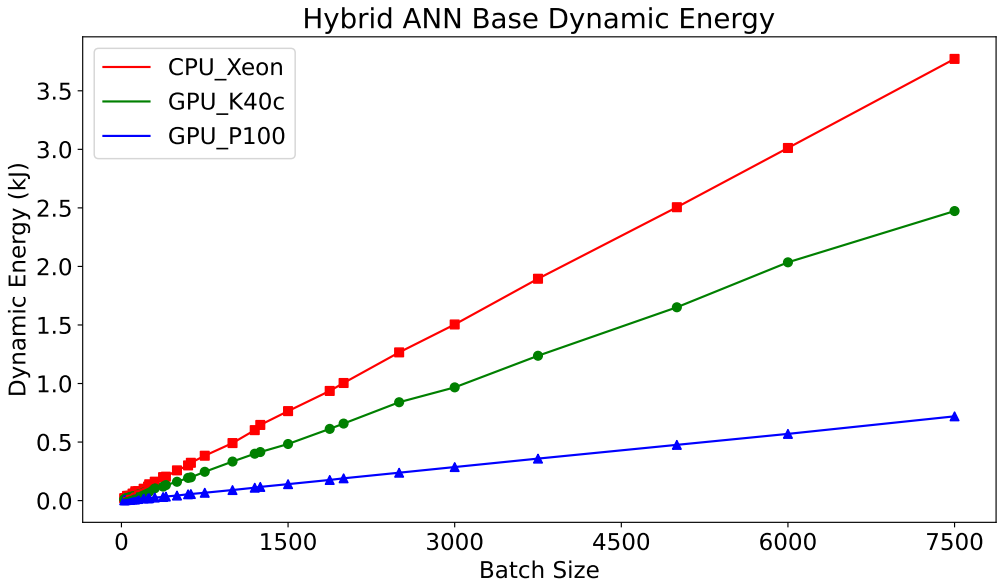

Figure 5.4: Base dynamic energy consumptions for training a multi-layer perceptron ANNs using a single batch whose sizes range from 25 to 7,500 on HCLServer01. Only batch sizes that evenly divide the total 60,000 samples are considered.

the other optimisation algorithm, HEPOPTA, to obtain discrete sets of Pareto optimal solutions for the two optimisation objectives. It should be highlighted that to construct the functions for a given batch size, we use its base execution time and dynamic energy consumption to generate both continuous and discrete functions as discussed in Section 5.1.

Consider the training workload of the neural network, implemented in the hybrid ANNs application, with a batch size of 400. This workload is executed on the HCLServer01 platform, and the MNIST dataset is used for training. Figure 5.5 shows two linear segments of the continuous Pareto front, generated using LBOPA and HEPOPTA.

Figure 5.5 displays a discrete set of 76 Pareto optimal solutions obtained by HEPOPTA based on the discrete time and dynamic energy functions. These Pareto optimal solutions are highlighted in red crosses, where each point represents a trade-off solution between execution time and energy consumption for a given distribution of 150 batches of size 400 between the three abstract

Table 5.1: HCLServer01: Specifications of the Intel Xeon Gold 6152 multicore CPU, Nvidia K40c and Nvidia P100 PCIe.

Intel Xeon Gold 6152	
Socket(s)	1
Cores per socket	22
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30976 KB
Main memory	96 GB
NVIDIA K40c	
No. of processor cores	2880
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec
NVIDIA P100 PCIe	
No. of processor cores	3584
Total board memory	12 GB CoWoS HBM2
Memory bandwidth	549 GB/sec

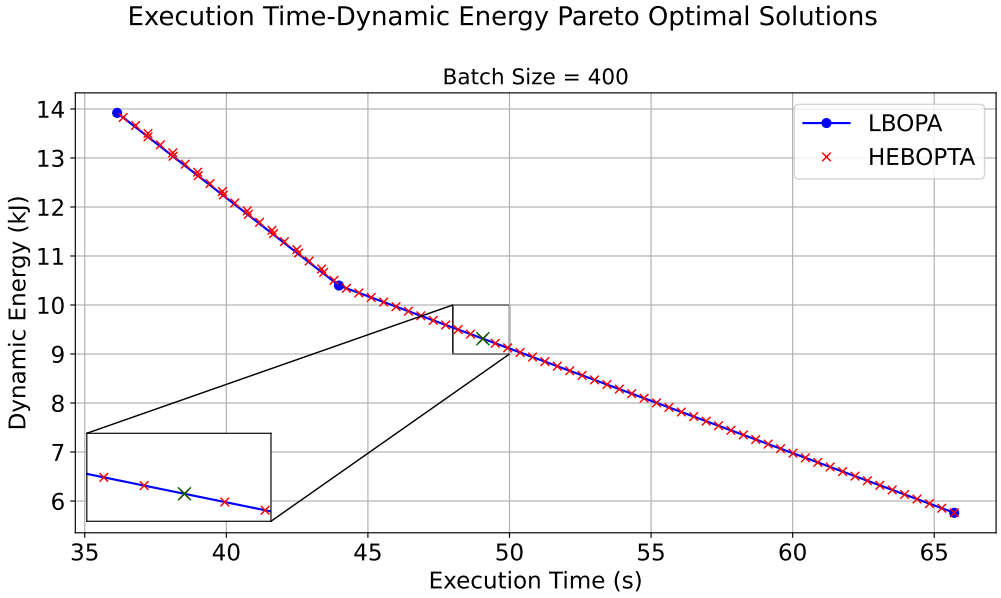

Figure 5.5: Pareto-optimal solutions for the Hybrid ANNs application executing on HCLServer01 to train a multi-layer perceptron ANNs using the MNIST dataset for a given batch size of 400.

processors. For instance, the solution highlighted in green on the Pareto front distributes the total 150 batches among the CPU_Xeon, GPU_K40c, and GPU_P100 as 0, 38 and 112, respectively. Executing the hybrid ANNs application with this partitioning configuration results in an execution time of 49 seconds (s) and a dynamic energy consumption of 9.3 kilojoules (kJ). According to this Pareto front, the workload distribution maximising the performance has an execution time of 36.4 seconds and dynamic energy consumption of 13.8 kJ. The workload distribution with the optimised dynamic energy consumption of 5.76 kJ has an execution time of 65.7 seconds. It implies that optimising for performance results in an energy consumption increase of up to 139%, while reducing energy consumption leads to a maximum of 80% degradation in performance.

There also exists another Pareto front in Figure 5.5, which is generated by LBOPA using the continuous linear time and dynamic energy functions of the hybrid ANNs. This piecewise continuous linear Pareto front (the blue linear segments) not only includes the solutions obtained by HEOPTA but also determines an infinite number of theoretical solutions. We call a solution theoretical when it assigns a non-integer number of batches that cannot be directly implemented in real-world scenarios. These solutions represent abstract distributions of total batches that are mathematically valid but infeasible for actual execution due to the requirement for whole, discrete batches. In contrast, the solutions returned by HEPOPTA are practical because they do not divide a batch. For instance, the left-most solution in the LBOPA Pareto front determines the theoretical minimum execution time for training the neural network for the batch size of 400. This theoretical solution has an execution time of 36.1 seconds and dynamic energy consumption of 13.9 kJ, and the workload distribution between CPU_Xeon, GPU_K40c and GPU_P100 for this solution is $\{26.7, 40.8, 82.5\}$. Due to the non-integer distribution of batches in LBOPA, there are some discrepancies between LBOPA and HEPOPTA Pareto fronts, however, they both follow the same trend in execution time and dynamic energy consumption. In addition, as shown in the zoomed area in Figure 5.5, there exists a continuum of theoretical solutions between two practical ones.

Since the application is executed on a hybrid platform comprising three dif-

ferent devices (`p = 3`), the LBOPA Pareto front consists of two (`p - 1`) linear segments (see section 3.4). The left-most solution is performance optimal, balancing the load of the three processors. The second breakpoint in the LBOPA Pareto front allocates zero workload to the CPU, and the 150 batches are distributed between GPU_K40 and GPU_P100 in proportion to their speeds. Finally, the right-most energy-optimal solution assigns the whole workload to GPU_P100, leaving the other devices unused (inactive). Therefore, except for the left-most solution, the other Pareto-optimal solutions are imbalanced. It is important to note that there is an infinite number of Pareto-optimal trade-off solutions for execution time and energy consumption.

We create an experimental data set which includes all batch sizes ranging between 50 and 7,500 with a step size of 5 and divides the total 60,000 samples. For each batch size, after creating its time and energy functions, we employ HEPOPTA to identify the trade-off solutions for the hybrid ANNs application.

The minimum, average, and maximum cardinality of Pareto fronts determined by HEPOPTA are 4, 123, and 611, respectively. There is a greater number of Pareto-optimal solutions for smaller batch sizes compared with the larger ones.

We study the impact of performance optimisation on dynamic energy consumption and vice versa using the experimental data set. The percentage performance improvement is obtained using $\frac{t_{e_{opt}} - t_{opt}}{t_{opt}} \times 100$, where t_{opt} is optimal execution time and $t_{e_{opt}}$ determines the execution time of the solution with optimal dynamic energy consumption. The percentage dynamic energy improvement is calculated using $\frac{e_{t_{opt}} - e_{opt}}{e_{opt}} \times 100$, where e_{opt} is optimal dynamic energy consumption, and $e_{t_{opt}}$ indicates the amount of dynamic energy consumption for the solution with the minimum execution time. The results indicate minimum, average and maximum performance improvements of 60.0, 77.5 and 81.7 per cent, respectively. Additionally, the minimum, average and maximum energy savings are 113.9, 137.3, and 142.6 per cent. We repeat this study to obtain the results from LBOPA. The performance improvement and energy savings are 81.8 per cent and 141.7 per cent, respectively. Because of the integer distribution in HEPOPTA, there are some discrepancies

between the minimum, average, and maximum improvements in the results obtained from HEPOPTA Pareto fronts. In contrast, these differences are not observed for the results obtained by LBOPA.

We also find to what extent execution time can be improved when the dynamic energy consumption is increased by up to 10 per cent over the optimal solution and to what extent dynamic energy can be reduced with a 10 per cent degradation in performance over the optimal solution. The percentage performance improvement is obtained using $\frac{t_{e_{opt}} - t_{e_{opt \times 1.1}}}{t_{e_{opt}}} \times 100$, where $t_{e_{opt \times 1.1}}$ is the execution time of solution with ten per cent greater dynamic energy consumption over the optimal dynamic energy consumption. Similarly, the percentage dynamic energy saving is obtained using $\frac{e_{t_{opt}} - e_{t_{opt \times 1.1}}}{e_{t_{opt}}} \times 100$ where $e_{t_{opt \times 1.1}}$ determines the dynamic energy consumption of a solution with ten per cent increase in execution time over optimal execution time. Based on the results obtained by HEPOPTA, the average and maximum performance improvements are 3.0 and 4.1 per cent. In addition, the average and maximum dynamic energy savings are 10.1 and 15.7 per cent. It is important to note that the slope of the line segments on the Pareto fronts represents the trade-off between execution time and dynamic energy consumption. A steeper slope indicates a significant reduction in dynamic energy for a minor increase in execution time, whereas a gentler slope results in a minor change between the two objectives.

5.3 Analysing Hybrid ANNs on Simulated Platforms

In this section, we replicate our experiments with HEPOPTA on several simulated hybrid machines, each comprising varying numbers of GPU_K40c and GPU_P100 accelerators. In this experiment, all parameters, including the MNIST training dataset and the experimental data set of batch sizes, remain consistent with those used in Section 5.2.

The first platform in this case study consists of four abstract processors: one CPU_Xeon, two GPU_K40cs, and one GPU_P100. Figure 5.6 illustrates the discrete, with a cardinality of 91, and the continuous piecewise linear

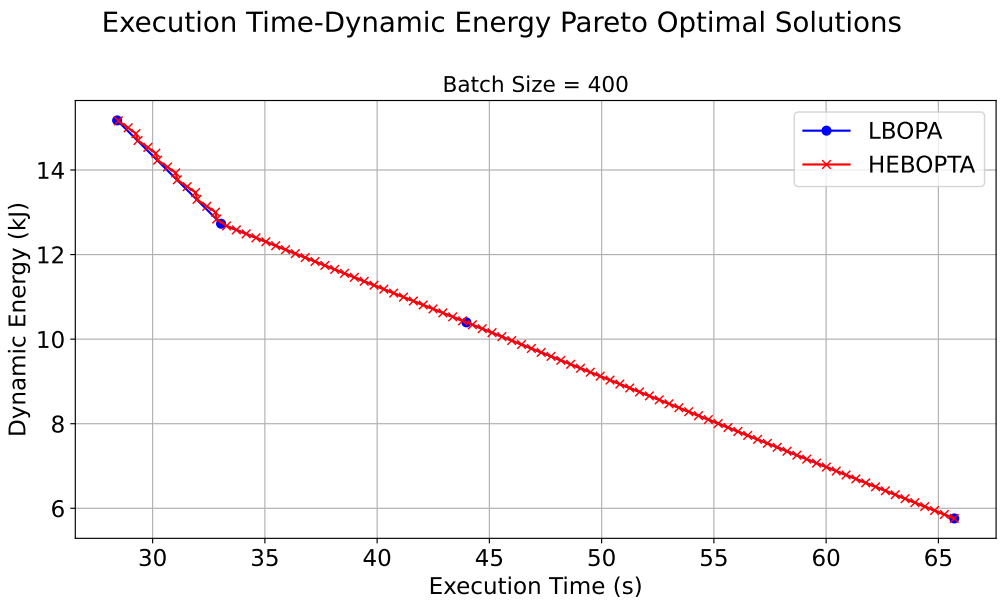

Figure 5.6: Pareto-optimal solutions for the hybrid ANNs application executing on a hybrid platform, consisting of four abstract processors, one CPU_Xeon, two GPU_K40cs, and one GPU_P100, to train a multi-layer perceptron ANNs using the MNIST dataset for a given batch size of 400.

Pareto fronts of the hybrid ANNs application for a batch size of 400. The minimal executing time is 28.5 seconds and its dynamic energy consumption is 15.16 kJ. The optimised dynamic energy consumption is 5.76 kJ with an execution time of 65.7 seconds. Table 5.2 summarises the average and maximum performance improvements and energy savings achieved on this platform for the batch sizes existing in the experimental data set.

In our next case study, we examine another platform consisting of four abstract processors: one CPU, one GPU_K40c, and two GPU_P100s. The Pareto fronts generated with HEPOPTA and LBOPA algorithms for a batch size of 400 is illustrated in Figure 5.7. Although a Pareto front of three segments was expected, the obtained Pareto front consists of only two segments, where the right-most solution equally divides the workload of 150 batches between the two identical GPU_P100 units, where each receives 75 batches. This results in a load-balanced distribution between these two abstract processors. Since these abstract processors are identical and the most energy-efficient

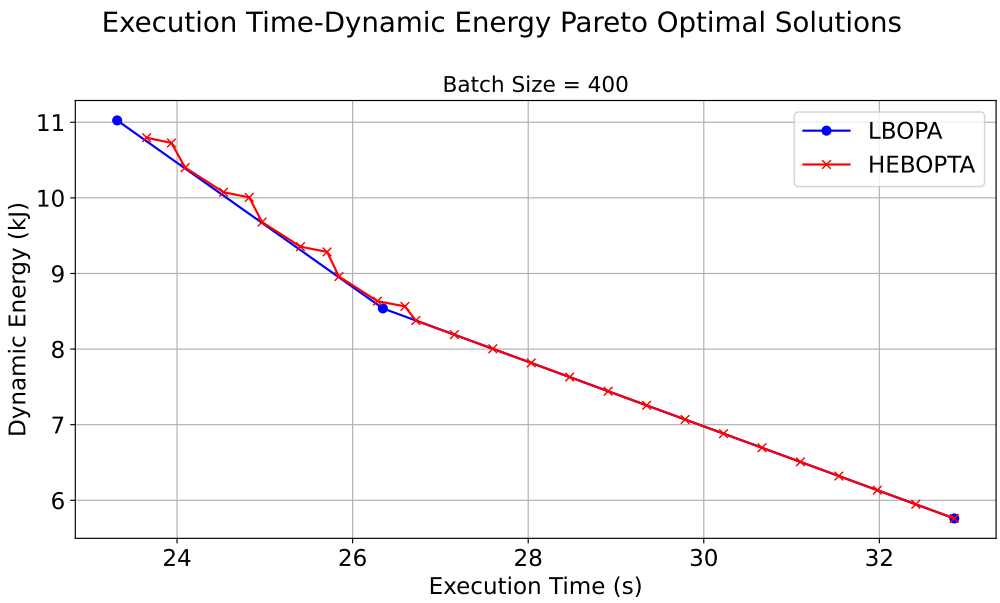

Figure 5.7: Pareto-optimal solutions for the hybrid ANNs application executing on a hybrid platform, consisting of four abstract processors, one CPU_Xeon, one GPU_k40c, and two GPU_P100s, to train a multi-layer perceptron ANNs using the MNIST dataset for a given batch size of 400.

ones available on this platform, no other solution offers better energy consumption as execution time increases. The experimental results obtained on this platform are available in Table 5.2.

In addition to the previous three platforms, we conducted our experiments on three more simulated hybrid machines. Table 5.2 compares the benchmarking results for all six platforms. Moreover, Figures 5.8, 5.9, and 5.10 visualise the information presented in the table, providing a clearer illustration of how trends are affected as the number of K40c and P100 GPUs changes.

As shown in Figure 5.8, the number of Pareto optimal solutions increases as more K40c GPUs are added, however, this trend does not hold for the P100 GPUs. This is because the GPU is the most energy-efficient processing unit on our platforms. As discussed earlier, adding two or more energy-efficient GPUs, such as the P100, does not introduce new line segments to the Pareto front.

We also compare the optimal execution times and dynamic energy con-

CPU xeon	GPU K40c	GPU P100	Pareto Front Cardinality		Performance Improvement		Energy Improvement		Performance Gain		Energy Gain	
			Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
1	1	1	123	611	77.5	81.7	137.3	142.6	3.0	4.1	10.1	15.7
1	2	1	145	723	123.1	131.2	157.3	163.5	3.0	4.1	8.5	15.0
1	3	1	160	800	167.9	180.3	170.2	177.9	3.0	4.1	7.4	11.9
1	1	2	41	203	36.9	40.8	83.2	91.7	2.7	7.7	13.8	21.7
1	1	3	21	100	21.8	33.3	53.9	67.7	3.1	11.1	13.6	20.8
1	3	3	32	160	53.2	60.2	89.3	103.6	3.1	11.1	8.8	13.6

Table 5.2: Comparison of HEOPTA benchmarking results on different hybrid platforms.

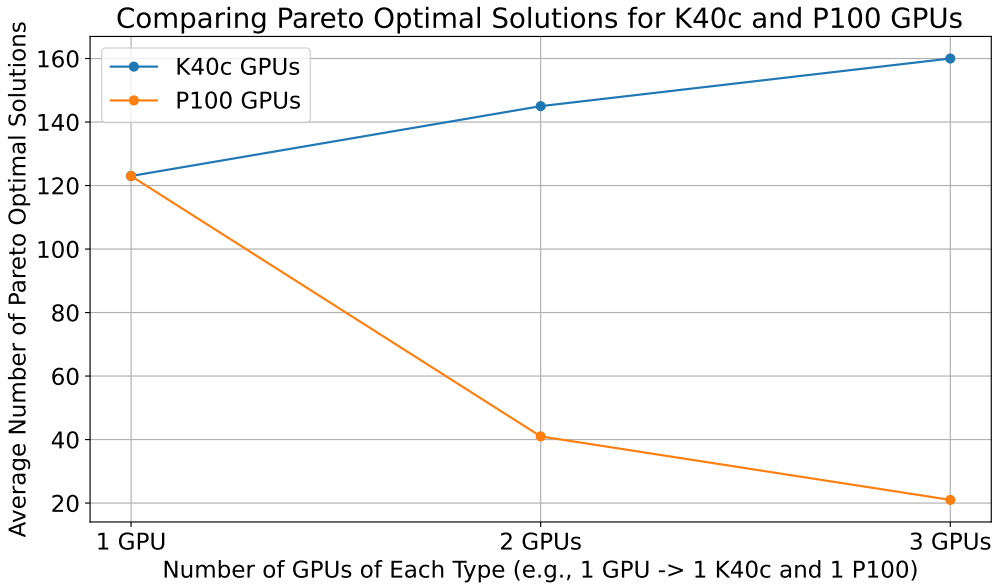

Figure 5.8: Average number of Pareto optimal solutions.

sumption achieved by the hybrid ANNs application across different heterogeneous servers. Figure 5.11 presents the simplified Pareto fronts for five different platforms. Each platform is encoded using the format x-y-z, where x, y, and z denote the number of CPU_Xeon, GPU_K40c, and GPU_P100 units in the respective architecture. The left-most point on each line represents the optimal execution time and its corresponding dynamic energy consumption for a given device, while the right-most point illustrates the minimal dynamic en-

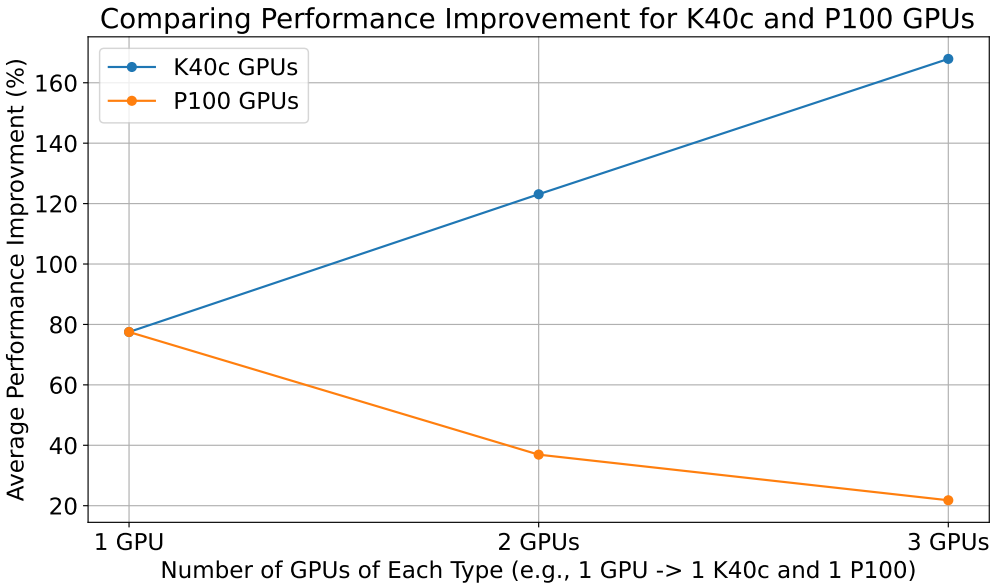

Figure 5.9: Performance improvement percentage.

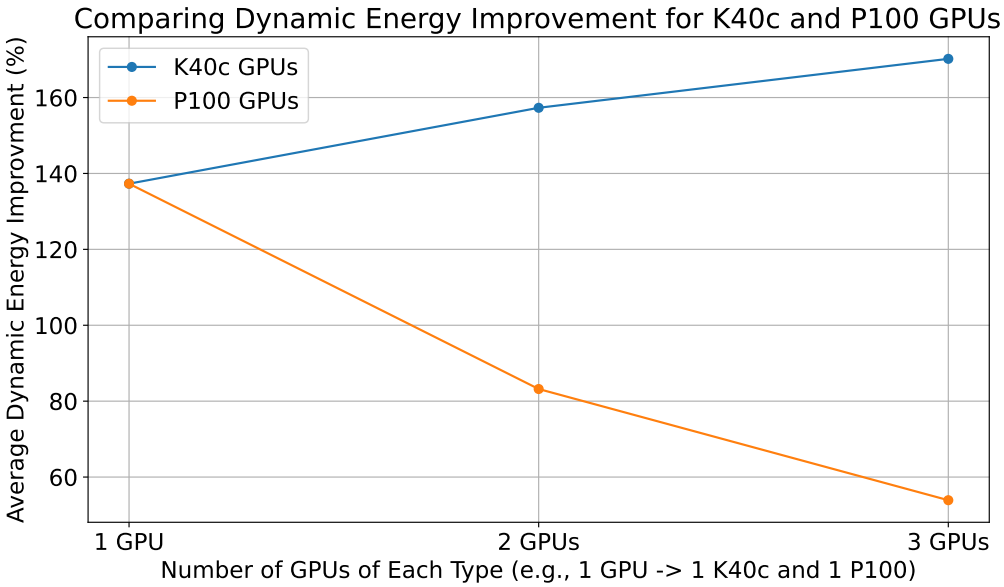

Figure 5.10: Dynamic energy improvement percentage.

ergy consumption alongside its execution time. Due to the linear nature of the performance and energy functions, the optimal execution time and dynamic

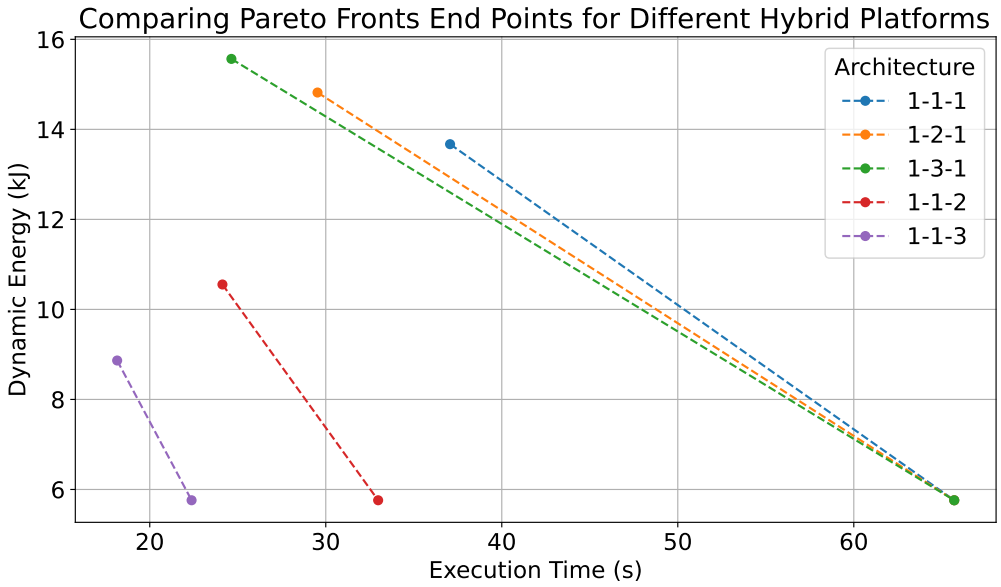

Figure 5.11: Comparison of optimal execution times and dynamic energy consumption achieved for the hybrid ANNs application executing on different hybrid platforms.

energy achievable for the hybrid ANNs application on any platform depend on the total workload (i.e., the number of samples in a dataset), rather than the batch size, and consequently, the number of batches. Since the number of samples is consistent across all experiments, the optimal performance and energy consumption for the hybrid ANNs application remains the same on a hybrid platform, regardless of batch size. Although minor differences may arise in the optimal values obtained from HEPOPTA due to the integer-based distribution of batches, the theoretical optimal values for a platform are identical.

Because performance-optimal solutions balance the load across all abstract processors, the optimal execution time decreases when more processing resources (such as GPUs) are added to run the application. On the other hand, all the energy-optimal distributions have identical dynamic energy consumption. This is because the optimal solution for dynamic energy allocates the whole workload (all batches) to GPU_P100, the most energy-efficient abstract processor available on our experimental platforms. Including additional less-energy-efficient processing units improves the training performance but

results in higher energy consumption for the performance-optimal solution. Furthermore, as shown in Figure 5.11, using less-energy-efficient processing units during training results in a wider difference between the two endpoints of Pareto fronts. This increased disparity leads to a higher percentage improvement for both training performance and energy savings, as illustrated in Figures 5.9 and 5.10.

Chapter 6

Conclusion

While numerous machine learning packages exist, most are tailored for homogeneous platforms. Those that claim compatibility with heterogeneous systems fail to fully utilise all available resources concurrently. Consequently, they cannot achieve peak performance and energy efficiency of highly hybrid HPC platforms. To address this issue, in this thesis, we first focused on implementing a hybrid ANNs application using a programming model integrating Pthreads, OpenMP and OpenACC and enabling the simultaneous execution of training tasks on different processing resources of the hybrid heterogeneous HPC platform.

After executing the hybrid ANNs application on a heterogeneous hybrid server for different distributions of batches, we discovered a significant impact of batches distribution on the execution time and dynamic energy consumption. In addition, we observed a conflicting correlation between performance and energy, indicating no single solution can optimise both objectives simultaneously.

These observations motivated us to consider the distribution of batches between processing devices as an important decision variable for the performance and energy bi-objective optimisation problem. It aims to optimise the parallel execution of a given training workload of n batches by a set of p heterogeneous processors, with batches distribution as the sole decision variable.

To solve this problem, two global optimisation algorithms, LBOPA and HEP-

OPTA, were applied to the linear profiles of the hybrid ANNs application to obtain its Pareto optimal sets. These sets contain trade-off solutions between execution times and energy consumption.

The performance and energy consumption of our hybrid ANNs application and their interplay were experimentally studied on several heterogeneous platforms with various numbers of accelerators. It is demonstrated that there is a wide range of trade-off solutions between execution times and energy consumption in the Pareto optimal sets obtained for the hybrid ANNs application. These solutions enable users to configure the ANNs application to either achieve higher performance or save more energy, depending on their preferred optimisation objective.

In addition, our experiments revealed that each Pareto front includes a load-balanced solution that optimises performance. However, the rest of the solutions in the Pareto optimal set are either partially balanced or fully imbalanced. We also observed that including additional processing resources for training an ANNs leads to enhanced performance. However, adding less-energy-efficient processing units, while improving training performance, leads to higher energy consumption for the performance-optimal solution.

The potential future work, which could be relevant in the extension of this thesis, includes:

1. **Extending the optimised hybrid application for other types of ANNs:** In this research, Fully Connected networks serve as the foundational architecture for ANNs, and data parallelism is employed as the model parallelisation strategy. Data parallelism is one of the most widely used and effective parallelisation methods in available ANN applications [4, 105]. In data parallelism approach, the same model is replicated across multiple devices; the input data are partitioned among them; each device computes gradients on its local data subset; and the gradients are subsequently synchronized (e.g., via averaging).

   Based on prior studies [105, 110], data parallelism is particularly effective when individual training samples can be processed independently. Under this assumption, we hypothesise that the data parallelism ap-

proach used for fully connected ANNs is extendable to many other ANN architectures, particularly those with similar data-independence characteristics, including:

- *Convolutional Neural Networks (CNNs):* CNNs are widely used in image processing, computer vision, and pattern recognition tasks due to their ability to automatically extract spatial features. Images constitute independent training samples; consequently, data parallelism has been widely adopted in practice for training CNNs. Extending the optimised hybrid approach to CNNs could lead to significant improvements in training speed and scalability, particularly for large-scale datasets. Efficient parallelization of convolutional layers, pooling operations, and backpropagation would be key areas of focus.
- *Transformers:* Transformers are widely used in Natural Language Processing (NLP), computer vision, and multimodal learning due to their ability to model long-range dependencies through self-attention mechanisms. Training data such as sentences, image patches, or token sequences typically constitute independent samples, making data parallelism a natural and effective parallelisation strategy for Transformer models. Extending the optimised hybrid approach to Transformers could significantly improve training throughput and scalability, particularly for large-scale datasets and deep architectures.
- *Autoencoders and Variational Autoencoders (VAEs):* Autoencoders and VAEs are commonly employed for representation learning, dimensionality reduction, and generative modeling. These models are typically trained on datasets where each input sample is processed independently to learn latent representations, which aligns well with the assumptions of data parallelism. Consequently, data parallelism has been widely applied in practice to accelerate the training of autoencoders and VAEs. Extending the optimised hybrid approach to these architectures could enhance training effi-

ciency and scalability, particularly for high-dimensional data and large datasets. Key areas of focus would include the parallelisation of encoder–decoder networks, latent space sampling (in the case of VAEs), and backpropagation across reconstruction and regularization losses.

However, it is important to note that data parallelism has inherent limitations and cannot be universally applied to all ANN types. In particular, the proposed approach is not directly extendable to the following cases:

(a) **Extremely large models that cannot fit on a single device.** When a single model replica exceeds GPU memory capacity, data parallelism alone becomes infeasible, and alternative strategies such as model or pipeline parallelism are required.

(b) **ANN architectures with non-independent samples, such as Graph Neural Networks (GNNs).** In these models, nodes may share neighbors across batches, leading to strong inter-sample dependencies and significant communication overhead, which can substantially reduce the effectiveness of data parallelism.

2. **Incorporating out-of-sample error as an additional optimisation objective:** While this thesis focuses on computational efficiency and scalability, and optimising the energy consumptions, another crucial aspect is improving training accuracy and out-of-sample error convergence speed.

   Data parallelism in neural network training potentially reduces the wall-clock time required to reach a desired out-of-sample error. However, the impact on the total number of training steps needed to achieve this error threshold can vary depending on factors such as batch size, learning rate, and model architecture. While larger batch sizes enabled by data parallelism can lead to faster per-epoch training times, they may also necessitate careful tuning of hyperparameters to maintain convergence efficiency [105, 111]. The previous related studies (like [105] and [111])

suggest that while data parallelism can reduce the time to reach a specified error threshold, achieving optimal results depends on appropriate adjustments to the training process.

In the future work, we can explore integrating training error minimisation as an explicit optimisation objective within the parallelisation framework. This could involve dynamic load balancing strategies that adapt based on error gradients, optimising hyper-parameters in real-time, or incorporating adaptive learning rate techniques to enhance convergence. Such an approach would not only improve the accuracy of the trained models but also contribute to better generalization performance across different neural network architectures.

3. **New hybrid approach combining multiple parallelisation methods:** In 2.1, a categorization of Artificial Neural Networks (ANNs) parallelisation methods was presented, dividing them into two main categories: Parallelisation in operators and Parallelization in networks. In this research, we specifically focused on data parallelism, which is a subcategory of parallelisation in networks and the most commonly used approach. Data parallelism allows for distributing training data across multiple computing units, enabling simultaneous processing and accelerating model training.

   However, while data parallelism offers significant benefits in scalability and performance, it does not fully exploit all potential optimizations. Our hypothesis is that a hybrid approach combining multiple parallelisation methods — such as model parallelism, pipeline parallelism, and operator-level parallelization — could further enhance computational efficiency and reduce energy consumption. By strategically leveraging different parallelization strategies based on the specific characteristics of neural networks and hardware architectures, future research can explore optimized hybrid implementations that balance workload distribution, minimize memory overhead, and improve overall system efficiency.

4. **Comparison of our proposed hybrid heterogeneous application**

with existing real-world applications: In this research, an optimized hybrid heterogeneous ANN application has been presented for the first time. However, the proposed approach does not yet fully reflect real-world deployment scenarios. In practice, parallel homogeneous frameworks such as TensorFlow are widely adopted. As future work, the practical relevance of the proposed application can be further demonstrated through a comprehensive performance comparison with popular, real-world ANN frameworks. Such an evaluation would provide deeper insight into the strengths, limitations, and applicability of the proposed hybrid heterogeneous approach in realistic computing environments.

Bibliography

[1] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[2] K.-L. Du and M. N. Swamy, *Neural networks and statistical learning*. Springer Science & Business Media, 2013.

[3] W. Knight, "Ai can do great things-if it doesn't burn the planet," *Wired Magazine*, 2020.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." `https://www.tensorflow.org/`, 2015. Software available from tensorflow.org.

[5] N. Ketkar and N. Ketkar, "Introduction to keras," *Deep learning with python: a hands-on introduction*, pp. 97–111, 2017.

[6] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[7] N. P. Jouppi, C. Young, N. Patil, D. Patterson, *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[8] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 3rd ed., 2016.

[9] K. L. Edition, "Intel Xeon Phi Processor High Performance Programming," *Developments in Environmental Modelling*, vol. 27, pp. 247–269, 2015.

[10] S. D. Brown and Z. G. Vranesic, *Fundamentals of digital logic with VHDL design*. McGraw Hill, 2022.

[11] X. Han, Z. Zhang, N. Ding, Y. Gu, X. Liu, Y. Huo, J. Qiu, Y. Yao, A. Zhang, L. Zhang, *et al.*, "Pre-trained models: Past, present and future," *AI Open*, vol. 2, pp. 225–250, 2021.

[12] A. Lastovetsky and R. Reddy, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 1119–1133, 04/2017 2017.

[13] R. G. Lyons, *Understanding Digital Signal Processing, 3/E*. Pearson Education India, 1997.

[14] N. Corporation, "NVIDIA DIGITS." `https://developer.nvidia.com/digits`, 2025. Accessed: 2025-02-24.

[15] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[16] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openaccfirst experiences with real-world applications," in *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island,*

Greece, August 27-31, 2012. Proceedings 18, pp. 859–870, Springer, 2012.

[17] OpenACC, "OpenACC: Directives for Accelerators." `http://www.openacc.org`, 2024. [Online].

[18] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[19] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[20] NVIDIA, "CUDA Toolkit Documentation." `http://docs.nvidia.com/cuda/cublas`, 2022.

[21] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi*, pp. 167–188, Springer, 2014.

[22] IBM, "Engineering and Scientific Subroutine Library (ESSL). version 6.2 guide and reference.." `https://www.ibm.com/support/knowledgecenter/SSFHY8_6.2/reference/essl_reference_pdf.pdf`, 2019.

[23] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," *In Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop (NIPS11)*, 2011.

[24] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth international workshop on frontiers in handwriting recognition*, Suvisoft, 2006.

[25] J. Park, K. Bin, G. Park, S. Ha, and K. Lee, "Aspen: Breaking operator barriers for efficient parallelization of deep neural networks," in

Advances in Neural Information Processing Systems (A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, eds.), vol. 36, pp. 68625–68638, Curran Associates, Inc., 2023.

[26] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," *In Proceedings of the International Conference on Learning Representations (ICLR15)*, 2015.

[27] S. Winograd, "Arithmetic complexity of computations. society for industrial and applied mathematics," 1980.

[28] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4013–4021, 2016.

[29] W. Yang, P. Wang, J. Fang, D. Dong, Z. Pang, R. He, P. Zhang, T. Tang, C. Huang, Y. Che, and J. Ren, "ndirect2: A high-performance library for direct convolutions on multi-core cpus," *IEEE Transactions on Computers*, pp. 1–14, 2025.

[30] S. Li, F. Yu, S. Zhang, H. Yin, and H. Lin, "Optimization of direct convolution algorithms on arm processors for deep learning inference," *Mathematics*, vol. 13, no. 5, 2025.

[31] K. Ma, R. Liu, X. Yan, Z. Cai, X. Song, M. Wang, Y. Li, and J. Cheng, "Adaptive parallel training for graph neural networks," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '25, (New York, NY, USA), p. 29–42, Association for Computing Machinery, 2025.

[32] C. Zhou, L. Ye, H. Peng, Z. Liu, J. Wang, and A. Ramírez-De-Arellano, "A parallel convolutional network based on spiking neural systems," *International Journal of Neural Systems*, vol. 34, no. 05, p. 2450022, 2024. PMID: 38487872.

[33] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456, PMLR, 2015.

[34] C. Nagaraju, Y. Ramesh, and C. K. Mohan, "A data parallel approach for distributed neural networks to achieve faster convergence," in *Sixteenth International Conference on Machine Vision (ICMV 2023)*, vol. 13072, pp. 380–389, SPIE, 2024.

[35] B. Forrest, D. Roweth, N. Stroud, D. Wallace, and G. Wilson, "Implementing neural network models on parallel computers," *The Computer Journal*, vol. 30, no. 5, pp. 413–419, 1987.

[36] H. Lee, C.-J. Hsieh, and J.-S. Lee, "Local critic training for model-parallel learning of deep neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 9, pp. 4424–4436, 2022.

[37] Y. Lee, J. Park, and C.-O. Lee, "Parareal neural networks emulating a parallel-in-time algorithm," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 5, pp. 6353–6364, 2024.

[38] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous systems," 2015.

[39] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[40] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.

[41] A. L. Gaunt, M. A. Johnson, M. Riechert, D. Tarlow, R. Tomioka, D. Vytiniotis, and S. Webster, "AMPNet: Asynchronous model-parallel training for dynamic neural networks," *arXiv preprint arXiv:1705.09786*, 2017.

[42] A. Beatrice, "Top 10 must-know artificial neural network software." https://www.analyticsinsight.net/top-10-must-know-artificial-neural-network-software/, September 2020.

[43] "Top 27 Artificial Neural Network Software." https://www.predictiveanalyticstoday.com/top-artificial-neural-network-software/, March 2022.

[44] H. M. Pandey and D. Windridge, "A comprehensive classification of deep learning libraries," in *Third International Congress on Information and Communication Technology* (X.-S. Yang, S. Sherratt, N. Dey, and A. Joshi, eds.), (Singapore), pp. 427–435, Springer Singapore, 2019.

[45] Neural Designer, "Neural Designer - Machine Learning Software." https://www.neuraldesigner.com, 2025. Accessed: 2025-02-24.

[46] NeuroDimension, Inc., "NeuroSolutions." https://neurosolutions.com, 2025. Accessed: 2025-02-24.

[47] Neuroph Project, "Neuroph - Java Neural Network Framework." http://neuroph.sourceforge.net, 2025. Accessed: 2025-02-24.

[48] J. Redmon, "Darknet: Open Source Neural Networks in C." https://pjreddie.com/darknet/, 2013–2016.

[49] Y. Tang, "TF. Learn: TensorFlow's high-level module for distributed machine learning," *arXiv preprint arXiv:1612.04251*, 2016.

[50] A. Karpathy, "ConvNetJS: Deep Learning in your browser." https://cs.stanford.edu/people/karpathy/convnetjs/, 2014. Accessed: 2025-02-24.

[51] U. of Stuttgart, "Stuttgart Neural Network Simulator (SNNS)." https://www.ra.cs.uni-tuebingen.de/SNNS/, 2025. Accessed: 2025-02-24.

[52] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: A Modular Machine Learning Software Library." `http://torch.ch/`, 2002. Accessed: 2025-02-24.

[53] N. Pavlov, "MLPNeuralNet: Fast Multilayer Perceptron Neural Network Library for iOS and Mac OS X." `https://github.com/nikolaypavlov/MLPNeuralNet`, 2025. Accessed: 2025-02-24.

[54] A. J. Tulloch, "DNNGraph: A Deep Neural Network Model Generation DSL in Haskell." `https://github.com/ajtulloch/dnngraph`, 2025. Accessed: 2025-02-24.

[55] A. B. Larsen, "DeepPy: Pythonic Deep Learning." `https://andersbll.github.io/deeppy-website/index.html`, 2025. Accessed: 2025-02-24.

[56] A. Kirillov, "AForge.Neuro: Artificial Neural Networks Library." `https://www.aforgenet.com/framework/`, 2013. Accessed: 2025-02-24.

[57] A. Krizhevsky, "CUDA-convnet2: Fast c++/cuda implementation of convolutional neural networks." `https://github.com/akrizhevsky/cuda-convnet2`, 2012. Accessed: 2025-02-24.

[58] A. D. Luca, "DN2A: Dynamic neural networks architect." `https://github.com/antoniodeluca/dn2a`, 2025. Accessed: 2025-02-24.

[59] D. Yuret, "Knet.jl: Koç university deep learning framework." `https://github.com/denizyuret/Knet.jl`, 2016. Accessed: 2025-02-24.

[60] R. Fontenot, J. Lazarus, P. Rudick, and A. Sgambellone, "Hierarchical neural networks (HNN): Using tensorflow to build hnn," *SMU Data Science Review*, vol. 6, no. 2, p. 4, 2022.

[61] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, *et al.*, "Lasagne." `https://github.com/Lasagne/Lasagne`, 2015. Accessed: 2025-02-24.

[62] C. Zhang, "Mocha.jl: Deep Learning Framework for Julia." `https://github.com/pluskid/Mocha.jl`, 2015. Accessed: 2025-02-24.

[63] J. Barrow and H. Ju, "LambdaNet: A Functional Neural Network Library in Haskell." `https://github.com/jbarrow/LambdaNet`, 2015. Accessed: 2025-02-24.

[64] Goml, "GoBrain: Neural Networks in Go." `https://github.com/goml/gobrain`, 2025. Accessed: 2025-02-24.

[65] A. Manglik, M. Patel, H. Mao, B. Salami, J. Park, L. Orosa, and O. Mutlu, "NEON: Enabling Efficient Support for Nonlinear Operations in Resistive RAM-based Neural Network Accelerators," *arXiv preprint arXiv:2211.05730*, 2022.

[66] N. Systems, "NEON: Deep Learning Framework." `https://github.com/NervanaSystems/neon`, 2025. Accessed: 2025-02-24.

[67] Tedsta, "Deeplearn-rs: Neural Networks in Rust." `https://github.com/tedsta/deeplearn-rs`, 2024. Accessed: 2025-02-24.

[68] RustNN Contributors, "RustNN: Neural Network Library in Rust." `https://github.com/rustnn/rustnn`, 2025. Accessed: 2025-02-24.

[69] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[70] M. Cierniak, M. J. Zaki, and W. Li, "Compile-time scheduling algorithms for a heterogeneous network of workstations," *The Computer Journal*, vol. 40, no. 6, pp. 356–372, 1997.

[71] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *Journal of Parallel and Distributed Computing*, vol. 61, no. 4, pp. 520–535, 2001.

[72] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix multiplication on heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1033–1051, 2001.

[73] A. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, p. 104, IEEE, 2004.

[74] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *The International Journal of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007.

[75] P. K. Smolarkiewicz and W. W. Grabowski, "The multidimensional positive definite advection transport algorithm: Nonoscillatory option," *Journal of Computational Physics*, vol. 86, no. 2, pp. 355–375, 1990.

[76] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on intel xeon phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 787–797, 2016.

[77] W. Zhang, X. Ji, B. Song, S. Yu, H. Chen, T. Li, P.-C. Yew, and W. Zhao, "Varcatcher: A framework for tackling performance variability of parallel workloads on multi-core," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1215–1228, 2016.

[78] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2176–2190, 2018.

[79] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A hierarchical data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous multi-accelerator NUMA nodes," *IEEE Access*, vol. 8, pp. 7861–7876, 2019.

[80] M. Mezmaz, N. Melab, Y. Kessaci, Y. C. Lee, E.-G. Talbi, A. Y. Zomaya, and D. Tuyttens, "A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 11, pp. 1497–1508, 2011.

[81] H. M. Fard, R. Prodan, J. J. D. Barrionuevo, and T. Fahringer, "A multi-objective approach for workflow scheduling in heterogeneous environments," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 300–309, IEEE, 2012.

[82] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future generation computer systems*, vol. 28, no. 5, pp. 755–768, 2012.

[83] Y. Kessaci, N. Melab, and E.-G. Talbi, "A pareto-based metaheuristic for scheduling hpc applications on a geographically distributed cloud federation," *Cluster Computing*, vol. 16, pp. 451–468, 2013.

[84] J. J. Durillo, V. Nae, and R. Prodan, "Multi-objective energy-efficient workflow scheduling using list-based heuristics," *Future Generation Computer Systems*, vol. 36, pp. 221–236, 2014.

[85] J. Kołodziej, S. U. Khan, L. Wang, and A. Y. Zomaya, "Energy efficient genetic-based schedulers in computational grids," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 809–829, 2015.

[86] B. Subramaniam and W.-c. Feng, "Statistical power and performance modeling for optimizing the energy efficiency of scientific computing," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pp. 139–146, IEEE, 2010.

[87] N. Gholkar, F. Mueller, and B. Rountree, "Power tuning hpc jobs on power-constrained systems," in *Proceedings of the 2016 International*

Conference on Parallel Architectures and Compilation, pp. 179–191, 2016.

[88] E. Gabaldon, J. L. Lerida, F. Guirado, and J. Planes, "Blacklist muti-objective genetic algorithm for energy saving in heterogeneous environments," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 354–369, 2017.

[89] A. Chakrabarti, S. Parthasarathy, and C. Stewart, "A pareto framework for data analytics on heterogeneous systems: Implications for green energy usage and performance," in *2017 46th International Conference on Parallel Processing (ICPP)*, pp. 533–542, IEEE, 2017.

[90] R. R. Manumachu and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 160–177, 2017.

[91] R. Reddy Manumachu and A. L. Lastovetsky, "Design of self-adaptable data parallel applications on multicore clusters automatically optimized for performance and energy through load distribution," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 4, p. e4958, 2019.

[92] H. Khaleghzadeh, M. Fahad, R. Reddy Manumachu, and A. Lastovetsky, "A novel data partitioning algorithm for dynamic energy optimization on heterogeneous high-performance computing platforms," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 21, p. e5928, 2020.

[93] H. Khaleghzadeh, M. Fahad, R. R. Manumachu, and A. Lastovetsky, "Optimization of data-parallel applications on heterogeneous hpc platforms for dynamic energy through workload distribution," in *European Conference on Parallel Processing*, pp. 320–332, Springer, 2019.

[94] H. Khaleghzadeh, M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on het-

erogeneous hpc platforms for performance and energy through workload distribution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 543–560, 2020.

[95] R. R. Manumachu, H. Khaleghzadeh, and A. Lastovetsky, "Acceleration of bi-objective optimization of data-parallel applications for performance and energy on heterogeneous hybrid platforms," *IEEE Access*, vol. 11, pp. 27226–27245, 2023.

[96] H. Khaleghzadeh, R. Reddy Manumachu, and A. Lastovetsky, "Efficient exact algorithms for continuous bi-objective performance-energy optimization of applications with linear energy and monotonically increasing performance profiles on heterogeneous high performance computing platforms," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 20, p. e7285, 2023.

[97] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel algorithm for bi-objective performance-energy optimization of applications with continuous performance and linear energy profiles on heterogeneous hpc platforms," in *European Conference on Parallel Processing*, pp. 166–178, Springer, 2021.

[98] H. Khaleghzadeh, *Novel Data-Partitioning Algorithms for Performance and Energy Optimization of Data-Parallel Applications on Modern Heterogeneous HPC Platforms*. PhD thesis, University College Dublin, 2019.

[99] X. Li and P.-C. Shih, "An early performance comparison of cuda and openacc," in *MATEC Web of Conferences*, vol. 208, p. 05002, EDP Sciences, 2018.

[100] J. Đukić and M. Mišić, "An evaluation of directive-based parallelization on the gpu using a parboil benchmark," *Electronics*, vol. 12, no. 22, p. 4555, 2023.

[101] S. Pophale, S. Boehm, and V. G. Vergara Larrea, "Comparing high performance computing accelerator programming models," in *International*

Conference on High Performance Computing, pp. 155–168, Springer, 2019.

[102] S. Becerra, "Simple neural network implementation in c++." `https://towardsdatascience.com/simple-neural-network-implementation-in-c-663f51447547`, 2019.

[103] A. Khazaei and H. Khaleghzadeh, "Parallel_ANN: Implementation of a Parallel Artificial Neural Network." `https://github.com/UoP-Data/Parallel_ANN`, 2024. Accessed: 2024-12-04.

[104] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[105] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," *Journal of Machine Learning Research*, vol. 20, no. 112, pp. 1–49, 2019.

[106] L. Chen, H. Wang, J. Zhao, D. Papailiopoulos, and P. Koutris, "The effect of network width on the performance of large-batch training," *Advances in neural information processing systems*, vol. 31, 2018.

[107] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou, "Deep learning scaling is predictable, empirically," *arXiv preprint arXiv:1712.00409*, 2017.

[108] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-gpu platforms using functional performance models," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2506–2518, 2014.

[109] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "A comparative study of methods for measurement of energy of computing," *Energies*, vol. 12, no. 11, p. 2204, 2019.

[110] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[111] N. Lee, T. Ajanthan, P. H. Torr, and M. Jaggi, "Understanding the effects of data parallelism and sparsity on neural network training," *arXiv preprint arXiv:2003.11316*, 2020.

[112] HCL, "HCLWattsUp: API for power and energy measurements using WattsUp Pro Meter." `http://git.ucd.ie/hcl/hclwattsup`, 2016.

Appendix A

Experimental Methodology

To make sure the experimental results are reliable, we follow the methodology described below:

- The server is fully reserved and dedicated to the experiments during execution. We also ensure that there are no drastic fluctuations in the load due to abnormal events in the server by monitoring its load continuously for a week using the tool *sar*. Insignificant variation in the load was observed during this monitoring period suggesting normal and clean behaviour of the server.
- A given hybrid application is executed simultaneously on all abstract processors to obtain a data point in its speed or dynamic energy functions. The application is repeatedly executed until the sample mean of measurements lies in a user-defined confidence interval and a user-defined precision is achieved. We set the confidence interval as 95% and the precision as 0.1 (10%) for our experiments. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions by plotting the distributions of observations.
- We set *OMP_PLACES* and *OMP_PROC_BIND* environment variables to bind all the threads of a hybrid application to CPU cores.

A.0.1 Methodology to Measure Execution Time and Energy Consumption

Suppose there exists a hybrid application, which is named *app*, consisting of two sample kernels, *Kernel_cpu* and *Kernel_gpu*, which run in parallel. The goal is to measure the execution time and the dynamic energy consumption of kernels in the application. To do this, we instrument the sample application as shown in Algorithm 4. This instrumented application returns the execution time of each kernel and the energy consumption of the two kernels.

Algorithm 4: Instrumentation of a sample application (*app*) consisting of two kernels, running on CPU and GPU simultaneously.

```
1:  HCL_WATTSUP_START()
2:  #pragma parallel
3:  Begin
4:      te_cpu1 ← gettimeofday()
5:      KERNEL_CPU()
6:      te_cpu2 ← gettimeofday()
7:  End
8:  Begin
9:      te_gpu1 ← gettimeofday()
10:     KERNEL_GPU()
11:     te_gpu2 ← gettimeofday()
12: End
13: energy_app ← HCL_WATTSUP_STOP()
14: te_cpu ← te_cpu2 − te_cpu1
15: te_gpu ← te_gpu2 − te_gpu1
16: return (te_cpu, te_gpu, energy_app)
```

Methodology to Measure Execution Time

We instrument each kernel in the hybrid application (*app*) by using the member function *gettimeofday()* of the Linux library *sys/time.h* to measure its execution time separately. As shown in Algorithm 4, the execution times are stored in variables te_{cpu} and te_{gpu} and are returned at the end of the application execution.

Methodology to Measure the Energy Consumption

We have two heterogeneous hybrid nodes. Each node is facilitated with one WattsUp Pro power meter that sits between the wall A/C outlets and the input power sockets of the node. These power meters capture the total power consumption of the node. The power meters have data cables connected to one USB port of the node. One Perl script collects the data from the power meter using the serial USB interface. The execution of these scripts is non-intrusive and consumes insignificant power.

The power meters are periodically calibrated using an ANSI C12.20 revenue-grade power meter, Yokogawa WT210. The maximum sampling speed of the power meters is one sample every second. The accuracy specified in the data-sheets is $\pm 3\%$. The minimum measurable power is 0.5 watts, and the accuracy at 0.5 watts is ± 0.3 watts.

We use *HCLWattsUp* API, which gathers the readings from the power meters to determine the average power and energy consumption during the execution of an application for the whole node. *HCLWattsUp* API [112] also provides two macros: *HCL_WATTSUP_START* and *HCL_WATTSUP_STOP*. The *HCL_WATTSUP_START* macro starts gathering power readings from the power meter using the aforementioned Perl script, whereas the *HCL_WATTSUP_STOP* stops gathering and return the total energy as a sum of these power readings.

To measure the amount of dynamic energy consumed by the application, we invoke *HCL_WATTSUP_START* (Line 1) and *HCL_WATTSUP_STOP* (Line 13) macros as shown in Algorithm 4. The consumed energy is stored in the variable $energy_{app}$ and is returned at the end of the application execution.

A.0.2 Methodology to Ensure Reliability of Experimental Results

As explained in Section A.0.1, each application is instrumented for measuring its performance and energy consumption. The measured execution times and consumed energy in each run of the application are stored in the variables

te_{cpu}, te_{gpu}, and $energy_{app}$ respectively, which are returned when the application execution finishes (Sample algorithm 4).

We keep running the application until the sample means of the measured execution times and energy consumption of the application lie within a given confidence interval, and a given precision is achieved. For this, we employ a script, which is named $MeanUsingTest()$. Algorithm 5 presents the pseudocode of this script. It executes the application *app* repeatedly until one of the following three conditions is satisfied:

1. The maximum number of repetitions ($maxReps$) has been exceeded (Line 4).

2. The sample means of all devices (kernel execution times and the application energy consumption) fall in the confidence interval cl, and the precision of measurement eps has been achieved (Lines 10-13).

3. The elapsed time of the repetitions of application execution has exceeded the maximum time allowed ($maxT$ in seconds) (Lines 14-15).

$MeanUsingTest()$ returns the sample means of the execution times for each abstract processor (i.e. $time_{cpu}$, $time_{gpu}$) and the energy consumption of all kernels (i.e. $energy$). The input parameters are minimum and maximum number of repetitions, $minReps$ and $maxReps$. For our experiments, these values are set to 3 and 10. The values of $maxT$, cl, and eps are respectively set to 3600, 0.95, and 0.1. If the precision of measurement is not achieved before the maximum number of repeats have been completed, we increase the number of repetitions and also the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in our experiments.

Algorithm 5: Script determining the mean of an experimental run using student's t-test.

Input:

app: The application to execute,

$minReps$: The minimum number of repetitions, $minReps \in \mathbb{Z}_{>0}$,

$maxReps$: The maximum number of repetitions, $maxReps \in \mathbb{Z}_{>0}$,

$maxT$: The maximum time allowed for the application to run, $maxT \in \mathbb{R}_{>0}$,

cl: The required confidence level, $cl \in \mathbb{R}_{>0}$,

eps: The required accuracy, $eps \in \mathbb{R}_{>0}$,

Output:

$reps\#$: The number of experimental runs actually made, $reps\# \in \mathbb{Z}_{>0}$,

$elapsedTime$: The elapsed time, $elapsedTime \in \mathbb{R}_{>0}$,

$time_{cpu}$: The mean CPU execution times, $time_{cpu} \in \mathbb{R}_{\geq 0}$,

$time_{gpu}$: The mean GPU execution times, $time_{gpu} \in \mathbb{R}_{\geq 0}$,

$energy$: The mean consumed energy, $energy \in \mathbb{R}_{>0}$

1 `MeanUsingTest(`$app, minReps, maxReps, maxT, cl, eps$`)`

2 $reps \leftarrow 0$; $stop \leftarrow 0$; $etime \leftarrow 0$;

3 $sum_{cpu} \leftarrow 0$; $sum_{gpu} \leftarrow 0$; $sum_{eng} \leftarrow 0$;

4 **while** *(*$reps < maxReps$*)* ***and*** *(*$!stop$*)* **do**

5 $(t_{cpu}[reps], t_{gpu}[reps], eng[reps]) \leftarrow Execute(app)$;

6 $sum_{cpu} += t_{cpu}[reps]$;

7 $sum_{gpu} += t_{gpu}[reps]$;

8 $sum_{eng} += eng[reps]$;

9 **if** *(*$reps > minReps$*)* **then**

10 $stop_{cpu} \leftarrow CalAccuracy(cl, reps + 1, t_{cpu}, eps)$;

11 $stop_{gpu} \leftarrow CalAccuracy(cl, reps + 1, t_{gpu}, eps)$;

12 $stop_{eng} \leftarrow CalAccuracy(cl, reps + 1, t_{eng}, eps)$;

13 $stop \leftarrow stop_{cpu} \wedge stop_{gpu} \wedge stop_{eng}$;

14 **if** $\max\{sum_{cpu}, sum_{gpu}\} > maxT$ **then**

15 $stop \leftarrow 1$;

16 $reps \leftarrow reps + 1$;

17 $reps\# \leftarrow reps$;

18 $elapsedTime \leftarrow \max\{sum_{cpu}, sum_{gpu}\}$;

19 $time_{cpu} \leftarrow \frac{sum_{cpu}}{reps}$;

20 $time_{gpu} \leftarrow \frac{sum_{gpu}}{reps}$;

21 $energy \leftarrow \frac{sum_{eng}}{reps}$;

22 **return** $(reps\#, elapsedTime, time_{cpu}, time_{gpu}, energy)$;

Algorithm 6 shows the pseudocode of the helper functions $CalAccuracy()$, which is used by $MeanUsingTest()$. It returns 1 if the sample mean of a given reading lies in the 95% confidence interval (cl) and a precision of 0.1 ($eps = 10\%$) has been achieved. Otherwise, it returns 0.

Algorithm 6: Algorithm Calculating Achieved Accuracy

Input: $cl, reps, Array, eps$

Output: Accuracy (0 or 1)

```
1 CalAccuracy(cl, reps, Array, eps)
2 clOut ← fabs(gsl_cdf_tdist_Pinv(cl, reps − 1)) × gsl_stats_sd(Array, 1, reps)/√reps ;
3 if clOut × reps / (Σ_{i=0}^{reps−1} Array[i]) < eps then
4     return 1;
5 return 0;
```

If the precision of measurement is not achieved before the maximum number of repeats have been completed, we increase the number of repetitions and also the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in our experiments.

Appendix B

List of Abbreviations

B.1 Acronyms

The following describes the significance of various acronyms and terms used throughout this thesis. The page on which each one is defined or used is also given.

Acronyms

AI Artificial Intelligence. 2

ANNs Artificial Neural Networks. 1–9, 11–15, 17, 18, 21, 25, 30–34, 39–49, 51–54, 56, 57, 59, 61, 63, 65–67, 69

API Application Programming Interface. 7, 33

CNN Convolutional Neural Network. 12, 13, 17, 33, 67

CPM Constant Performance Model. 22

CPU Central Processing Unit. 1, 2, 5, 7, 8, 12, 17, 18, 24, 30, 32–35, 37, 39–41, 49, 53, 56–59, 61

cuBLAS CUDA Basic Linear Algebra Subroutines library. 12, 33

DSP Digital Signal Processor. 5, 33

DVFS Dynamic Voltage and Frequency Scaling. 23

ESSL IBM Engineering and Scientific Subroutine Library. 12

FFT Fast Fourier Transform. 13

FPGA Field Programmable Gate Array. 1, 5, 33

FPM Functional Performance Model. 23

GNN Graph Neural Network. 13

GPGPU General-Purpose Graphics Processing Unit. 1

GPU Graphics Processing Unit. 1, 2, 5, 7, 8, 14, 17, 18, 33, 35, 41, 49, 53, 56–61, 63

HEPOPTA Heterogeneous Energy Performance Optimisation Algorithm. 22, 25, 26, 45, 47, 48, 54, 56–59, 63, 65

HPC High Performance Computing. 1, 5, 8, 22, 23, 25, 30, 32, 42, 65

LBOPA Linear Bi-objective OPtimisation Algorithm. 22, 25–27, 45, 47, 48, 53, 54, 56–59, 65

LSTM Long Short-Term Memory. 13

MKL Intel Math Kernel Library. 12

MNIST Modified National Institute of Standards and Technology database. 41, 49, 51, 52, 54, 58

MPI Message Passing Interface. 21

NLP Natural Language Processing. 67

NVC NVIDIA C compiler. 35, 39

OpenACC Open Accelerators. 7, 9, 18, 21, 33, 35, 39, 65

OpenMP Open Multi-Processing. 7, 9, 18, 21, 33–35, 39, 65

PCI Peripheral Component Interconnect. 49

PCIe Peripheral Component Interconnect Express. 41, 49, 55

Pthreads POSIX threads (POSIX: Portable Operating System Interface). 7, 9, 18, 21, 37, 39, 65

RAM Random Access Memory. 37

RNN Recurrent Neural Network. 13, 17

SNP Spiking Neural P neurons. 13

TPU Tensor Processing Unit. 1

VAEs Variational Autoencoders. 67, 68

Xeon Phi Intel Xeon Phi. 1, 5, 33