IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 10 March 2026, accepted 22 March 2026, date of publication 25 March 2026, date of current version 2 April 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3677675

RESEARCH ARTICLE

Parallel DNNs on Hybrid Heterogeneous Platforms: Design, Implementation, and Optimisation for Performance and Energy

HAMIDREZA KHALEGHZADEH[1], ATEFEH KHAZAEI[1], AND ALEXEY LASTOVETSKY[2], (Member, IEEE)
[1]School of Computing, University of Portsmouth, PO1 3HE Portsmouth, U.K.
[2]School of Computer Science, University College Dublin, Dublin, D04 V1W8 Ireland

Corresponding author: Alexey Lastovetsky (alexey.lastovetsky@ucd.ie)

This work was supported in part by Taighde Éireann–Research Ireland under Grant 20/FFP-P/8683; and in part by the Faculty of Technology, University of Portsmouth, through the Development Projects Funding (DPF) Scheme under Grant 2024/25.

ABSTRACT Deep Neural Network (DNN) packages are designed for homogeneous platforms, limiting their ability to leverage different types of processing resources simultaneously. This article first proposes a hybrid programming model for designing and implementing parallel and portable heterogeneous DNN applications, facilitating DNN parallel training on hybrid platforms. It then explores the performance and energy efficiency of the proposed methodology on hybrid platforms. We implement a fully-connected multilayer perceptron DNN application based on the proposed methodology and study its performance and energy behaviour for different distributions of batches between computing devices. Our findings highlight the outstanding impact of the distribution of batches on the execution time and energy consumption of DNN training, making it a critical decision variable for optimisation. We then mathematically formulate the bi-objective optimisation problem for the proposed DNN methodology, targeting both performance and energy consumption objectives, and obtain Pareto-optimal solutions for these two objectives using two algorithms. The performance improvement and energy saving of the hybrid DNNs are validated on heterogeneous servers. Our results demonstrate that solving the bi-objective optimisation problem yields a wide range 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. Furthermore, adding additional processing resources enhances DNN training performance, but using less energy-efficient devices, while improving performance, leads to increased energy consumption for performance-optimised solutions. We also compare our approach with TensorFlow's workload distribution, and the results show that our batch distribution delivers superior performance and energy efficiency.

INDEX TERMS Bi-objective optimisation, deep neural networks, energy optimisation, hybrid heterogeneous computing, parallel computing, performance optimisation, workload distribution.

I. INTRODUCTION

Due to their effectiveness in handling complex modelling, Deep Neural Networks (DNNs) are increasingly used in a wide range of fields, including image recognition, medical science, textual document processing, autonomous vehicles, etc. Training DNNs is highly computationally intensive because it simulates the super complex human brain processes, making DNN training both time- and energy-consuming. Consequently, modern machine learning frameworks, such as TensorFlow, leverage parallel processing, particularly many-core HPC platforms like GPGPUs (General Purpose Graphics Processing Units) and TPUs (Tensor Processing Units), to meet the escalating demand for faster training speed and improved performance per watt.

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Alawneh.

© 2026 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

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 is straightforward, achieved by dividing the workload *equally* among all devices. This approach guarantees optimal performance and energy consumption for parallel training DNNs executing on *homogeneous* clusters.

In the backdrop of the tight integration of multi-core CPUs and various many-core accelerators, such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), and Intel Xeon Phis, mainstream HPC platforms have become increasingly heterogeneous over the past decade, to maximise key objectives like performance, energy efficiency, cost-effectiveness, and flexibility. However, this heterogeneity introduces unprecedented challenges. Firstly, the diversity of accelerators, each with its own programming models, frameworks, and optimisation techniques, makes implementing DNN training algorithms more challenging and complex. As a result, algorithms designed and developed for a specific device are not portable to other types of devices. This limitation restricts DNN training algorithms to using only a subset of the computational resources available in modern heterogeneous clusters. Additionally, due to being designed for homogeneous executions, state-of-the-art machine learning packages and applications cannot fully exploit the potential performance and energy efficiency benefits of highly heterogeneous HPC platforms via *equal* workload distribution among different processing devices. This is because heterogeneous systems comprise devices with varying execution speeds and energy consumption characteristics, consequently, distributing the workload equally across such devices results in suboptimal computational performance and energy efficiency. For instance, TensorFlow supports 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, it 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 TensorFlow is facilitated with heterogeneous kernels, it only supports homogeneous execution at runtime. This limitation restrains TensorFlow from achieving reasonable performance and energy savings on hybrid heterogeneous platforms by harnessing the processing power of all processing resources simultaneously.

The first objective of this paper is to propose and develop a novel methodology for designing parallel and portable DNN 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 [1], OpenACC [2], [3], and Pthreads [4]. In this model, OpenMP, as an industry-standard API for shared-memory parallel programming, is employed to implement portable, multi-threaded DNN training kernels for CPUs. OpenACC is used to develop portable parallel kernels for offloading and executing parallel DNN training computations onto various accelerators, such as Nvidia GPUs. Pthreads is utilised to integrate the CPU and accelerator kernels into a unified hybrid parallel DNN application, enabling the simultaneous execution of DNN 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 DNN application.

The next goal is to propose an approach to optimise the execution of the DNN 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 multilayer perceptron DNN 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 three key findings: (i) Batches distribution has a significant impact on the performance and energy consumption of our DNN applications, (ii) Employing homogeneous-based strategies leads to suboptimal solutions when applied to heterogeneous systems, and (iii) 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 DNN methodology for performance and energy on heterogeneous platforms. We mathematically formulate the optimisation problem for hybrid DNN 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 DNN 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 DNNs via batches distribution to obtain a comprehensive range of exact trade-off solutions for both performance and energy consumption.

Our parallel and portable DNN 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 time and energy

consumption within Pareto optimal sets obtained for the parallel DNN application. These solutions allow users to configure the DNN application to prioritise either higher performance or greater energy savings, depending on their preferred objectives. The experimental analysis reveals that adding more processing resources for DNN training improves overall performance. However, incorporating less energy-efficient processing units, while improving training performance, results in higher energy consumption for performance-optimised solutions. Furthermore, comparing our results against the execution outcome of the hybrid DNN application under TensorFlow's distribution strategy demonstrates that our proposed distribution method delivers superior performance and energy efficiency.

The main contributions of this paper are summarised as follows:

- A methodology for designing parallel and portable DNN applications that can be executed on hybrid heterogeneous platforms with arbitrary numbers and types of accelerators.
- A hybrid programming model for implementing hybrid parallel DNN applications based on the proposed methodology, utilising three programming standards: OpenMP, OpenACC and Pthreads. Implementation of a fully connected multilayer perceptron DNN training application using the proposed methodology and programming model.
- A methodology for optimising hybrid parallel DNN applications for performance and energy consumption by obtaining Pareto-optimal solutions and enabling trade-offs between these objectives through batches distribution across computing devices.
- Bi-objective optimisation of the hybrid fully connected multilayer perceptron DNN application for performance and energy on different heterogeneous server configurations.

The rest of the paper is organised as follows. We discuss the related work in Section II. In Section III, we propose our methodology for implementing a parallel hybrid DNN application using three programming models: Pthreads, OpenMP and OpenACC. Section IV experimentally demonstrates the importance of bi-objective optimisation for performance and energy consumption in DNN training. The formulation of the bi-objective optimisation problem is presented in Section V. Section VI presents experimental results on different heterogeneous platforms and includes a comparison with TensorFlow's distribution strategy. Finally, we conclude the paper in Section VII.

II. RELATED WORK

In this section, we first review and categorise related research works in the area of DNN parallelisation. Following this, we study the latest research efforts tackling the problem of bi-objective optimisation of parallel applications for performance and energy consumption on HPC platforms.

A. DNN PARALLELISATION

Up to now, numerous research efforts have focused on the parallel execution of DNNs to reduce execution time. However, most of these studies have concentrated on homogeneous environments rather than heterogeneous ones. Additionally, the majority have not treated energy consumption as a separate optimisation objective. Only a few have explored energy reduction as a by-product of parallelisation. Despite these distinctions between previous studies and our work, the examination of these studies has provided valuable insights into the state-of-the-art parallelisation methods and the techniques that could be implemented in DNNs for our research.

DNN parallelisation methods can be broadly categorised into two main categories: (i) Parallelisation in Operators, and (ii) Parallelisation in Networks. Each of these two categories consists of several subcategories, as illustrated in Figure 1.

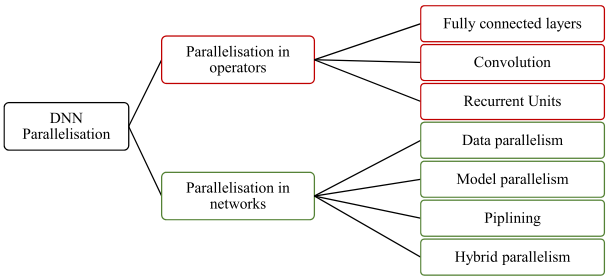


FIGURE 1. Categorisation of DNNs parallelisation research.

In *parallelisation in operators* studies, researchers have explored opportunities to parallelise the execution of layers. For certain types of DNNs, computations can be parallelised directly, while for others, computations must be reshaped to expose parallelism [5]. Below, we address the parallelism of three popular operators:

- **Fully Connected Networks:** A fully connected layer can be expressed and modelled as a matrix-vector multiplication of the weights and the neuron values. To achieve this, efficient linear algebra libraries, such as CUBLAS [6], MKL [7], and ESSL [8], can be employed. Vanhoucke et al. [9] presented various methods to optimise fully connected layers computations on CPUs. Specifically, this research highlights efficient loop construction, vectorisation, blocking, unrolling, and batching. Additionally, the study demonstrated how weights can be quantised to leverage fixed-point arithmetic instead of floating-point computations.
- **Convolutional Neural Networks (CNNs):** The on various platforms. The first algorithmic innovation proposed for CNNs was the transformation of discrete convolution into matrix multiplication using Toeplitz matrices, commonly referred to as the im2col technique [10], [11]. The second method involves leveraging the Fourier domain, where convolution is performed as an element-wise multiplication. In this approach,

the data and the kernels are transformed using FFT, multiplied, and the inverse FFT is applied to the result [12]. The third, and currently the most prevalent, method for CNNs is Winograd's algorithm for minimal filtering [13]. This technique, further advanced by Lavin and Gray [14], modifies the original CNN algorithm to efficiently handle the multiple filters present in convolutions.

- **Recurrent Neural Networks (RNNs):** The gating system within RNN units (e.g., LSTMs) involves multiple operations, such as small matrix multiplications and element-wise operations. These layers are usually implemented as a sequence of high-level operations; however, further acceleration is feasible in these layers. Additionally, RNN units are typically chained together, forming consecutive recurrent layers, which introduces two potential types of parallelism: parallelism within the same layer and parallelism between consecutive layers [5].

The parallelism in neural networks can be leveraged not only to compute individual operators efficiently but also to parallelise the entire neural network across different dimensions. The main strategies for entire network parallelisation include: partitioning input samples (data parallelism), partitioning the network structure (model parallelism), and partitioning layers (pipelining) [5]. These strategies are discussed below:

- **Data Parallelism:** This is a straightforward approach for DNN parallelisation via partitioning batches among multiple computational devices. This approach is supported by the vast majority of DNN frameworks, using a single GPU or multiple identical GPUs [5]. In this method, all processors should have a copy of the DNN parameters. The scaling of data parallelism is naturally defined by the batch size. In the weight update phase, the results of the partitions are averaged to obtain the gradient [15].
- **Model Parallelism:** Model parallelism, also known as network parallelism, divides neurons in each layer among computational devices. In this approach, the batches are replicated across all processors, while different parts of the DNN are computed on separate processors. This method can save memory, as the entire network does not need to be stored in one location, but it introduces additional communication overhead after every layer computation [16], [17].
- **Pipelining:** In neural networks, pipelining can refer to overlapping computations (e.g., between one layer and the next as data becomes available) or partitioning the DNN by depth, assigning specific layers to specific processors. Alternatively, pipelining can be considered as a form of data parallelism, as samples are processed through the network in parallel, similar to model parallelism [18]. The most common and widely adopted form of pipelining involves overlapping feedforward, backpropagation, and weight update computations [19]. This approach improves utilisation by reducing processor idle time.
- **Hybrid Parallelism:** In this approach, two or more parallelism strategies are employed to overcome the limitations of individual approaches and enhance overall performance [20], [21], [22].

B. 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 [23], [24], [25]. 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 minimising execution time are always load-balanced [26], [27].

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 [28], [29], [30], [31], [32]. 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.

C. 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: (i) system-level and (ii) 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, DVFS levels, cache size, and thermal design power. Mathematical, heuristic, or machine learning algorithms are then used to optimally map tasks to the platform [33], [34], [35], [36], [37], [38].

1) APPLICATION-LEVEL OPTIMISATION

: Application-level methods essentially modify applications to achieve optimal performance and energy consumption. This type of approach 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. [39] 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. [40] 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. [41] 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. [42] 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. [43], [44] 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. [45], [46] propose 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. 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 [47].

III. HYBRID DNN IMPLEMENTATION

This section outlines our methodology for designing parallel and portable DNN applications capable of training DNNs 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: (i) kernel-level and (ii) application-level. At the kernel level, parallel training computations are implemented for CPUs and accelerators using multi-threaded programming models. At the application 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 DNN application based on the hybrid parallel methodology.

A. 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 designed for parallel programming on multicore and shared-memory multiprocessor systems. To develop parallel and portable accelerator kernels, we use OpenACC [2], another directive-based programming standard similar to OpenMP, designed for offloading parallel computations onto accelerators.

Now, we discuss in detail the implementation of training kernels for a parallel fully connected multilayer perceptron DNN application using the proposed methodology. We utilise a simple fully connected neural network serial implementation in the C programming language [48] as the foundation for our kernels. This particular implementation is selected due to the following features:

- Supports the construction of DNNs 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 Stochastic Gradient Descent (SGD) for training, a widely used iterative algorithm for optimising DNN 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 the 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 for 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 DNN 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 already be 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 NVC compiler, which does not support 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 that 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 loop` 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.

In heterogeneous servers, accelerators have their own memory spaces, separate from the host CPU's main 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

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 labelled training samples.
- *num_batches*: The number of batches in *batches* for training.
- *batch_size*: The number of samples 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**

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.

Although Algorithm 1 and Algorithm 2 look similar at first glance, there are significant technical and implementation

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 labelled training samples.
     • num_batches: The number of batches in batches for training.
     • batch_size: The number of samples in each batch.
   In-Out :
     • weights: A 1D array of 2D matrices, where the i-th matrix (0 ≤ 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 ≤ 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 loop
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(···);
               // Update weights and biases
12             update_weights_and_biases_ACC(...);
13         end
14     end
15 end
```

differences between them. Algorithm 1 employs OpenMP to parallelise DNN training on multi-core CPUs, which have a limited number of cores with limited potential for parallelism. However, Algorithm 2 implements parallelism on accelerators with a large array of processing cores. Unlike the first algorithm, it is designed and implemented using OpenACC, enabling massive parallelism on many-core accelerators. In addition, explicit data transfers between the host CPU and accelerators are required in Algorithm 2 due to separate memory spaces between them.

B. 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 platform. 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 DNN 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 (`batches`, `num_batches`), consisting of two arrays of size `p`. For a given device $i \in \{0, 1, \cdots, p-1\}$, the pair (`batches`[$i$], `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 DNN is available at [49].

As discussed in the Introduction, one of the challenging steps in Algorithm 3 is to distribute total batches among computational devices (Line 4). In Section IV, 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 DNN algorithm. Then, we propose a method to obtain optimal trade-offs between these two optimisation objectives via workload partitioning.

IV. WORKLOAD DISTRIBUTION IN HYBRID DNN

Although existing machine learning packages support different computing devices, including CPUs and Nvidia

Algorithm 3 Hybrid DNN Training Application

```
1  Function Hybrid_DNN_Training(input_data, batch_size, p, weights,
   biases);
   Input :
     • input_data: A 1D array of labelled training samples.
     • batch_size: The number of samples in each batch.
     • p: The number of devices, including 1 CPU and p-1 accelerators, for parallel run.
   Output:
     • weights: A 1D array of 2D matrices, where the i-th matrix (0 ≤ i < num_layers) contains the
       trained network's weights for the i-th layer.
     • biases: A 2D array, where the i-th column (0 ≤ i < num_layers) contains
       the trained network's biases for the i-th layer.
2  begin
       // Initialise 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
```

GPUs, they are designed for homogeneous execution. These packages cannot be configured for simultaneous execution on hybrid platforms. As a result, they are unable to distribute training workloads effectively across all available resources.

For example, TensorFlow provides several cluster-level strategies for distributing computation across multiple devices using data parallelism. These strategies, such as `MirroredStrategy` for GPUs and `TPUStrategy` for TPUs, assume that the underlying devices have similar computational capabilities. Consequently, the training workload is evenly divided across devices. On heterogeneous platforms, this assumption leads to suboptimal performance, underutilisation of faster devices, and training bottlenecks. In such environments, alternative approaches or custom distribution strategies may be required, allowing users to manually assign workloads according to the computational capabilities of each device.

In this section, we conduct a case study to highlight the impact of workload (batches) distribution on the performance and energy consumption of our implemented parallel and portable DNN 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 PCIe GPU. The technical specifications of the platform are provided in Section VI.

In this case study, the implemented DNN application is trained on the MNIST dataset [50] for handwritten digit recognition. 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 [51]. 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 batches. In this experiment, the parallel DNN 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 Section VI-A.

Figure 2 displays the parallel execution time and energy consumption for the parallel DNN under four different workload distributions, where the right one distributes the batches equally among all the devices. Comparing the first two solutions, we can see that a 20% performance degradation leads 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%. Furthermore, distributing the workload equally among heterogeneous devices leads to higher execution times and energy consumption. 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.
- Employing workload distribution strategies designed for homogeneous platforms leads to degradation in both performance and energy consumption when applied to heterogeneous systems.
- Batches (workload) distribution significantly affects the performance and energy consumption of the Hybrid DNN and can be considered as a key decision variable for optimising the performance and energy efficiency.

These observations lead us to two crucial questions: (i) Is there a methodology to optimise the hybrid DNN execution

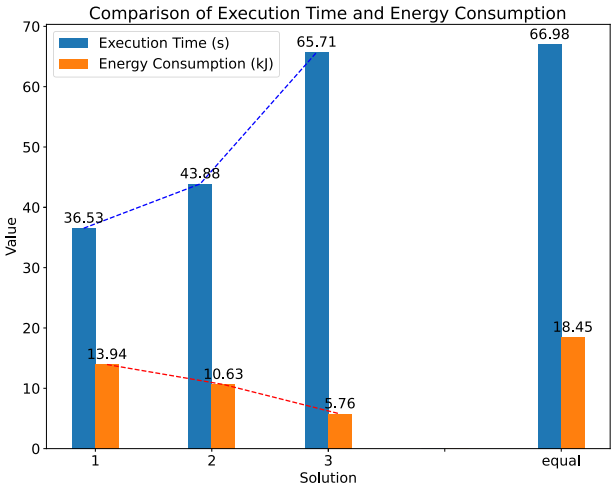


FIGURE 2. A case study comparing the execution time and energy consumption of the hybrid DNN for three different batches distributions.

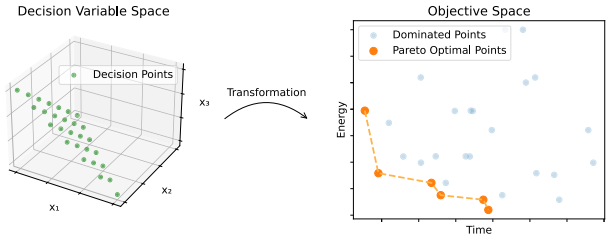


FIGURE 3. An example showing the set of decision variable vectors $\mathcal{F}$, the set of objective vectors $\mathcal{Z}$, and Pareto optimal solutions for $p = 3$.

for the two objectives, execution time and energy consumption, via batches (workload) distribution on heterogeneous platforms? (ii) 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 V, we mathematically formulate the bi-objective optimisation of the hybrid DNN application as a min-max-min-sum problem and then utilise two algorithms, HEPOPTA [45] and LBOPA [47] to solve this optimisation problem.

V. BI-OBJECTIVE OPTIMISATION PROBLEM

As mentioned in Section IV, 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 DNN 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:

$$minimise_X \ \{T(X),\ E(X)\} \tag{1}$$

where $T(X) : \mathbb{N}^p \to \mathbb{R}$ and $E(X) : \mathbb{N}^p \to \mathbb{R}$ 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 batches 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 3, Equation 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 $F(X) = \{f_0(x_0), f_1(x_1), \ldots, f_{p-1}(x_{p-1})\}$ and $G(X) = \{g_0(x_0), g_1(x_1), \ldots, g_{p-1}(x_{p-1})\}$ where $f_i(x) \in \mathbb{R}$ and $g_i(x) \in \mathbb{R}$ 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 1 can be expressed as:

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

where

$$X = \{x_0, x_1, \cdots, x_{p-1}\}$$
$$\&$$
$$B = \sum_{i=0}^{p-1} x_i \tag{2}$$

Equation 2 mathematically formulates the bi-objective optimisation of the hybrid DNN 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 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: (i) heuristic methods, such as Genetic Algorithms, and (ii) exact algorithms, such as LBOPA [47] and HEPOPTA [45]. In section VI, we will use LBOPA and HEPOPTA to obtain Pareto optimal solution sets for the hybrid DNN application. These two algorithms are briefly explained in the following sections.

A. LBOPA

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 paper) and the sum of continuous linear increasing functions (dynamic energy functions in this paper). 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 4 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 [47], the workload distribution for a solution lying between two consecutive breaking points results in an imbalanced allocation of workload across the processors.

B. HEPOPTA

HEPOPTA (Heterogeneous Energy Performance OPtimisation 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. Further details about this optimisation algorithm are available in [45].

VI. EXPERIMENTAL RESULTS

In this section, the performance and energy consumption of the implemented hybrid DNN are studied on several heterogeneous servers with different computational devices. We also compare the effectiveness of our proposed methodology against TensorFlow's workload distribution strategy, which is inherently designed for homogeneous platforms with identical devices.

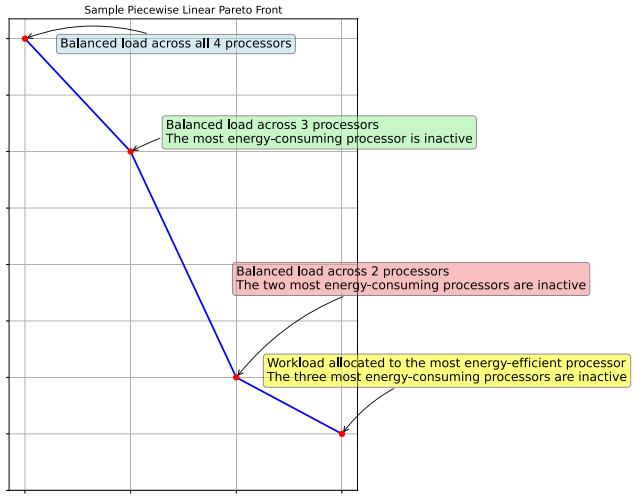


FIGURE 4. A sample continuous piecewise linear Pareto front built by LBOPA for $p = 4$.

First, we explain our methodology for accurately measuring the execution time and energy consumption of the hybrid DNN 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 DNN 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. The results are also compared with those obtained under TensorFlow's distribution strategy.

A. CONSTRUCTING PERFORMANCE AND DYNAMIC ENERGY FUNCTIONS

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

1) PERFORMANCE FUNCTIONS

To create execution time (performance) functions, we accurately measure the execution time of each kernel in the hybrid DNN application when training a specified number of batches for a given batch size. Each measurement captures the elapsed time from the start of a kernel to its completion, ensuring that both computation and communication times are

included. To ensure that our functions are reproducible and to minimise resource sharing between kernels, we employ the concept of *abstract processors* [52]. 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 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 co-scheduling and resource sharing on the times measured. Figure 5 presents the time functions of the application for the MNIST training dataset, 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 per cent confidence interval.

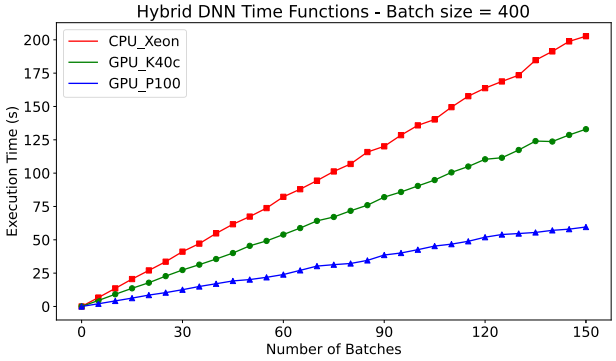


FIGURE 5. Time functions of the hybrid DNN executing on HCLServer01. The application trains a multi-layer perceptron DNN using the MNIST dataset, which consists of 60,000 samples, for a given batch size of 400.

Energy consumption during an application's execution can be measured using three main approaches: (i) on-chip power sensors, (ii) software-based energy predictive models, and (iii) physical measurement using external power meters. According to [53], 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 [45], which utilises system-level power measurements via power meters to accurately model the energy consumption of the *abstract* processors within the hybrid DNN application. According to this methodology, the dynamic energy consumption of the hybrid DNN application is the sum of the energy consumed by each of its kernels when executed individually.

Figure 6 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. In addition, energy measurements were performed from the start to the completion of each kernel to incorporate both computation and communication costs.

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.

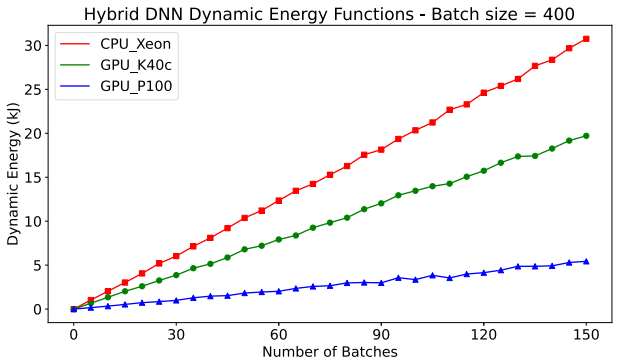


FIGURE 6. Dynamic energy functions of the Hybrid DNN executing on HCLServer01. The application trains a multi-layer perceptron DNN using the MNIST dataset, which consists of 60,000 samples, for a given batch size of 400.

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.

3) DISCUSSION

Using the measurement methodology discussed earlier, we experimentally created the DNN 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 and 6. 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 DNN application for different numbers of batches to construct its F and G functions. Instead, the base execution time and base energy consumption per device can be obtained by measuring the time and 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 7 and 8 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.

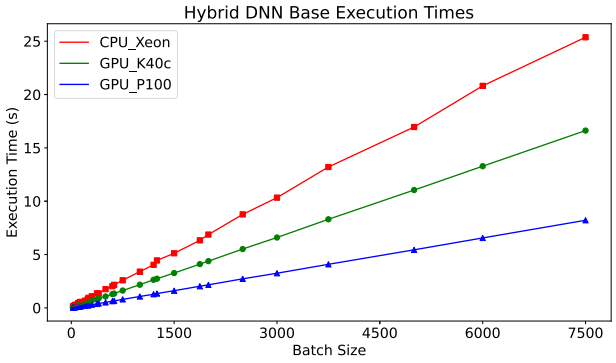


FIGURE 7. Base execution times for training a multi-layer perceptron DNN using a single batch whose sizes range from 25 to 7,500 on HCLServer01. Only batch sizes that evenly divide the total 60,000 samples are considered.

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

B. ANALYZING HYBRID DNN ON A REAL PLATFORM

In this section, we examine the improvements in performance and reductions in dynamic energy consumption of the hybrid DNN 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 1.

As discussed in Section VI-A, 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 DNN 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 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 VI-A.

Consider the training workload of the neural network, implemented in the hybrid DNN application, with a batch size of 400. This workload is executed on the HCLServer01 platform, and the MNIST dataset is used for training. Figure 9 shows two linear segments of the continuous Pareto front, generated using LBOPA and HEPOPTA.

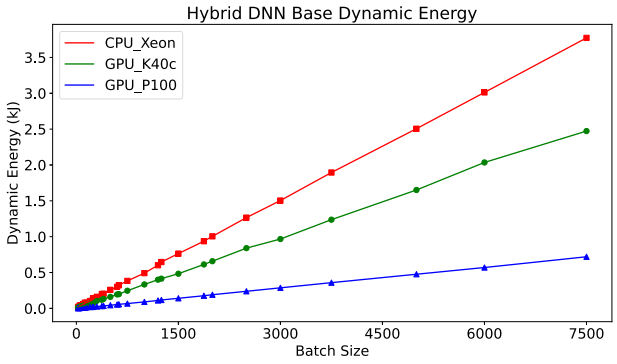


FIGURE 8. Base dynamic energy consumptions for training a multi-layer perceptron DNN using a single batch whose sizes range from 25 to 7,500 on HCLServer01. Only batch sizes that evenly divide the total 60,000 samples are considered.

TABLE 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

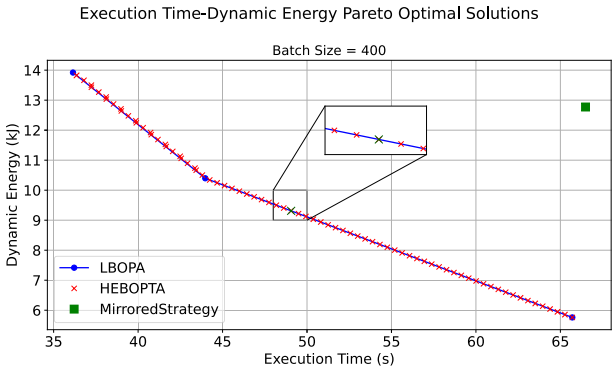


FIGURE 9. Pareto-optimal solutions for the Hybrid DNN application executing on HCLServer01 to train a multi-layer perceptron DNN using the MNIST dataset for a given batch size of 400.

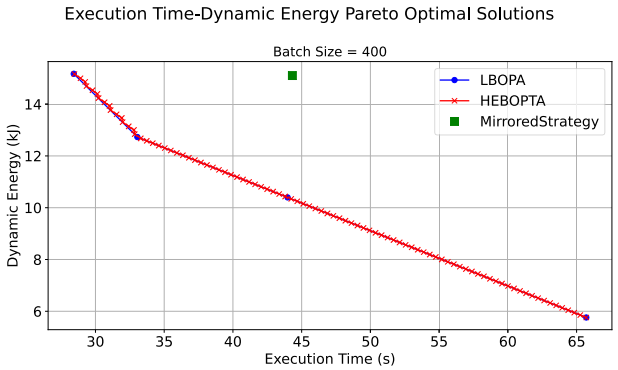


FIGURE 10. Pareto-optimal solutions for the hybrid DNN application executing on a hybrid platform, consisting of four abstract processors, one CPU_Xeon, two GPU_K40cs, and one GPU_P100, to train a multi-layer perceptron DNN using the MNIST dataset for a given batch size of 400.

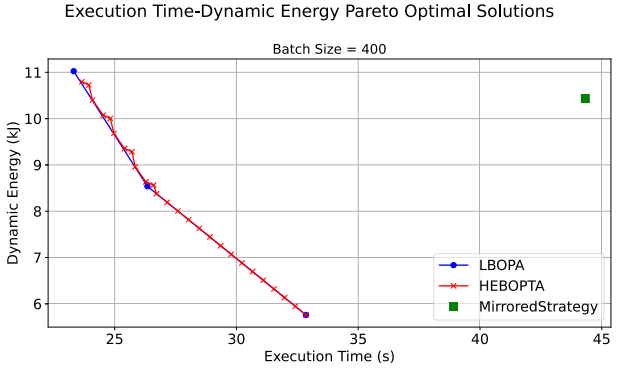


FIGURE 11. Pareto-optimal solutions for the hybrid DNN application executing on a hybrid platform, consisting of four abstract processors, one CPU_Xeon, one GPU_k40c, and two GPU_P100s, to train a multi-layer perceptron DNN using the MNIST dataset for a given batch size of 400.

The figure 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 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 DNN 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 9, which is generated by LBOPA using the continuous linear time and dynamic energy functions of the hybrid DNN. 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 a 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 9, there exists a continuum of theoretical solutions between two practical ones.

Since the application is executed on a hybrid platform comprising three different devices ($p = 3$), the LBOPA Pareto front consists of two ($p - 1$) linear segments (see section V-A). 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

TABLE 2. Comparison of HEOPTA benchmarking results on different hybrid platforms.

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 3. Comparison of HEOPTA vs. TensorFlow benchmarking results on different hybrid platforms.

CPU_ xeon	GPU_ K40c	GPU_ P100	Performance Gap		Energy Gap	
			Average	Max	Average	Max
1	1	1	78.9	83.8	121.3	121.7
1	2	1	47.4	55.7	158.8	162.3
1	3	1	33.3	41.8	179.3	182.6
1	1	2	80.2	89.9	79.4	81.1
1	1	3	81.0	92.7	59.8	60.9
1	3	3	46.6	61.8	115.9	121.7

Pareto-optimal trade-off solutions for execution time and energy consumption.

The green square above the obtained Pareto fronts in Figure 9 represents the execution outcome of the Hybrid DNN application when using TensorFlow's commonly adopted single-node distribution strategy, *MirroredStrategy*. This solution, with an execution time of 66.5 s and a dynamic energy consumption of 12.8 kJ, utilises only the two Nvidia GPUs (K40c and P100) and divides the batches evenly between them. It is approximately 83% slower than the performance-optimal solution and 122% less energy efficient than the energy-optimal Pareto-optimal solution. This experiment demonstrates that executing DNN training on a heterogeneous platform using frameworks designed for homogeneous environments leads to inefficient resource utilisation.

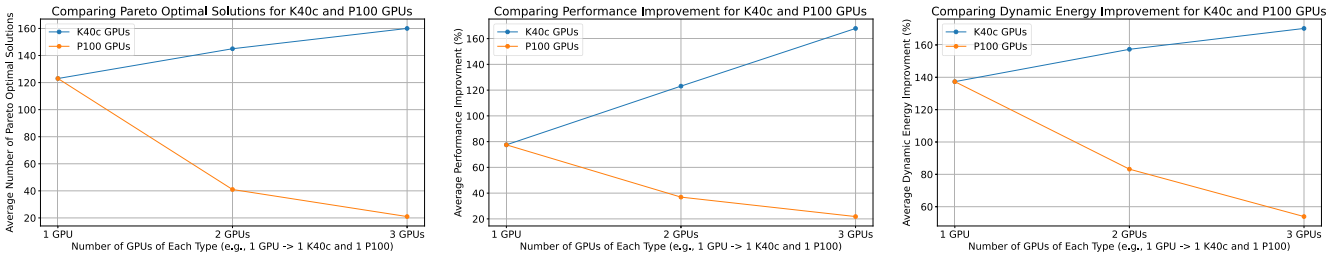
We create an experimental data set which includes all batch sizes ranging between 50 and 7,500 with a step size of 5 and divides the total 60,000 samples. For each batch size, after creating its time and energy functions, we employ HEPOPTA to identify the trade-off solutions for the hybrid DNN application.

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.

We use the experimental dataset to compare the performance and energy efficiency of the proposed method with TensorFlow's MirroredStrategy workload distribution. The performance gap percentage is computed as $\frac{t_{TF} - t_{opt}}{t_{opt}} \times 100$, where t_{opt} denotes the optimal execution time and t_{TF} represents the execution time achieved under MirroredStrategy. Similarly, the dynamic energy gap percentage is calculated as $\frac{e_{TF} - e_{opt}}{e_{opt}} \times 100$, where e_{opt} is the optimal dynamic energy consumption and e_{TF} is the dynamic



(a) Avg number of Pareto optimal solutions (b) Performance improvement percentage (c) Dynamic energy improvement percentage

FIGURE 12. Comparison of trends for different numbers of K40c and P100 GPUs.

energy consumption under MirroredStrategy. The results demonstrate minimum, average, and maximum performance improvements of 57.4, 78.9, and 83.8 per cent, respectively, along with corresponding energy savings of 113.6, 121.3, and 121.7 per cent.

C. ANALYZING HYBRID DNN 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 VI-B.

The first platform in this case study consists of four abstract processors: one CPU_Xeon, two GPU_K40cs, and one GPU_P100. Figure 10 illustrates the discrete, with a cardinality of 91, and the continuous piecewise linear Pareto fronts of the hybrid DNN 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 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 this figure, the green square indicates the application's outcomes when executed using TensorFlow's MirroredStrategy, which distributes the batches evenly across the three GPUs. Further comparison results between HEPOPTA and TensorFlow are presented in Table 3.

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 HEPOTA and LBOPA algorithms for a batch size of 400 are illustrated in Figure 11. 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 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 2. The green square shows the application's outcomes when executed using TensorFlow's MirroredStrategy, distributing batches evenly across the three GPUs. Table 3 contains further comparison results between HEPOPTA and TensorFlow.

In addition to the previous three platforms, we conducted our experiments on three more simulated hybrid machines. Table 2 compares the benchmarking results for all six platforms. The comparison results between HEPOPTA and TensorFlow are summarised in Table 3. The findings demonstrate that HEPOPTA consistently outperforms TensorFlow across all hybrid configurations, achieving better performance and substantially lower dynamic energy consumption.

Moreover, Figures 12a, 12b, and 12c visualise the information presented in Table 2, providing a clearer illustration of how trends are affected as the number of K40c and P100 GPUs changes.

As shown in Figure 12a, 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 consumption achieved by the hybrid DNN application across different heterogeneous servers. Figure 13 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 energy consumption alongside its execution time. Due to the linear nature of the performance and energy functions, the optimal execution time and dynamic energy achievable for the hybrid DNN application on any platform depend on the total workload (i.e., the number of samples

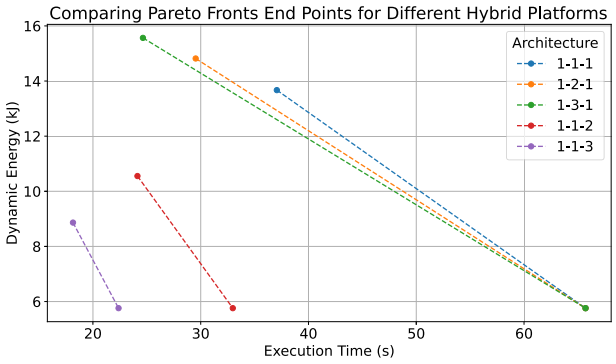


FIGURE 13. Comparison of optimal execution times and dynamic energy consumption achieved for the hybrid DNN application executing on different hybrid platforms.

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 DNN application remain 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 13, 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 12b and 12c.

VII. CONCLUSION

Although there is a wide range of machine learning packages, they are mainly designed for homogeneous platforms. Consequently, they cannot achieve peak performance and energy efficiency of highly hybrid HPC platforms. To address this issue, we first focused on implementing a hybrid DNN 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 DNN application on a heterogeneous server under different distributions of batches, we discovered a significant impact of batches distribution on 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 HEPOPTA, were applied to the linear profiles of the hybrid DNN 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 DNN application and its interplay were experimentally studied on several heterogeneous platforms with various numbers of accelerators. It is demonstrated that there is a wide range of trade-offs between execution time and energy consumption in the application's Pareto-optimal sets. These solutions enable users to configure the DNN 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 compared the training performance when executed using our proposed approach against TensorFlow's workload distribution method. The comparison results demonstrate that the proposed batch distribution consistently outperforms TensorFlow's strategy across all hybrid configurations considered in this paper, achieving superior performance and energy savings.

Although the proposed bi-objective optimisation approach has been evaluated on fully connected DNNs, it is extensible to a wide range of ANN architectures exhibiting comparable data-independence properties, where individual training samples can be processed independently. These include Convolutional Neural Networks (CNNs), Transformers, and Autoencoders and Variational Autoencoders (VAEs).

In future work, we aim to extend our proposed methodology by incorporating training error as an additional optimisation objective.

DECLARATION OF AI-ASSISTED TECHNOLOGIES IN THE WRITING PROCESS

During the preparation of this work, the authors used OpenAI ChatGPT to improve the readability and language of the manuscript. After using this tool, the authors reviewed and edited the content as needed, and they take full responsibility for the content of the published article.

REFERENCES

[1] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Mar. 1998.

[2] S. Wienke, P. Springer, C. Terboven, and D. A. Mey, "Openacc-first experiences with real-world applications," in *Proc. 18th Int. Conf. Euro-Par Euro-Par Parallel Process.*, Rhodes Island, Greece. Berlin, Germany: Springer, 2012, pp. 859–870.

[3] OpenACC. (2024). *OpenACC: Directives for Accelerators*. [Online]. Available: http://www.openacc.org

[4] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*., *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. Sebastopol, CA, USA: O'Reilly Media, 1996.

[5] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surveys*, vol. 52, no. 4, pp. 1–43, Jul. 2020.

[6] NVIDIA. (2022). *CUDA Toolkit Documentation*. [Online]. Available: http://docs.nvidia.com/cuda/cublas

[7] 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™*. Cham, Switzerland: Springer, 2014, pp. 167–188.

[8] IBM. (2019). *Engineering and Scientific Subroutine Library (ESSL). Version 6.2 Guide and Reference*. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSFHY8_6.2/reference/essl_reference_pdf.pdf

[9] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Proc. Deep Learn. Unsupervised Feature Learn. Workshop (NIPS)*, 2011, pp. 1–8.

[10] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Proc. 10th Int. workshop frontiers handwriting Recognit.*, 2006, pp. 1–7.

[11] J. Park, K. Bin, G. Park, S. Ha, and K. Lee, "Aspen: Breaking operator barriers for efficient parallelization of deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 36, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., Red Hook, NY, USA: Curran Associates, 2023, pp. 68625–68638.

[12] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–17.

[13] S. Winograd, *Arithmetic Complexity of Computations* (CBMS-NSF Regional Conference Series in Applied Mathematics), vol. 33. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics (SIAM), 1980, pp. 1–93.

[14] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4013–4021.

[15] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Int. Conf. Mach. Learn. (PMLR)*, 2015, pp. 448–456.

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

[17] H. Lee, C.-J. Hsieh, and J.-S. Lee, "Local critic training for model-parallel learning of deep neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 9, pp. 4424–4436, Sep. 2022.

[18] Y. Lee, J. Park, and C.-O. Lee, "Parareal neural networks emulating a parallel-in-time algorithm," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 35, no. 5, pp. 6353–6364, May 2024.

[19] M. Abadi et al., "Tensorflow: Large-scale machine learning on heterogeneous systems," White Paper, 2015. [Online]. Available: https://www.tensorflow.org/

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

[21] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, and K. Yang, "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 25, 2012, pp. 1232–1240.

[22] 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," 2017, *arXiv:1705.09786*.

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

[24] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *J. Parallel Distrib. Comput.*, vol. 61, no. 4, pp. 520–535, Apr. 2001.

[25] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix multiplication on heterogeneous platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1033–1051, Oct. 2001.

[26] A. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, Apr. 2004, pp. 104–118.

[27] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 1, pp. 76–90, Feb. 2007.

[28] P. K. Smolarkiewicz and W. W. Grabowski, "The multidimensional positive definite advection transport algorithm: Nonoscillatory option," *J. Comput. Phys.*, vol. 86, no. 2, pp. 355–375, Feb. 1990.

[29] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel xeon phi through load imbalancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 787–797, Mar. 2017.

[30] 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 Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1215–1228, Apr. 2017.

[31] 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 Trans. Parallel Distrib. Syst.*, vol. 29, no. 10, pp. 2176–2190, Oct. 2018.

[32] 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, 2020.

[33] 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," *J. Parallel Distrib. Comput.*, vol. 71, no. 11, pp. 1497–1508, Nov. 2011.

[34] H. M. Fard, R. Prodan, J. J. D. Barrionuevo, and T. Fahringer, "A multi-objective approach for workflow scheduling in heterogeneous environments," in *Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2012, pp. 300–309.

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

[36] Y. Kessaci, N. Melab, and E.-G. Talbi, "A Pareto-based metaheuristic for scheduling HPC applications on a geographically distributed cloud federation," *Cluster Comput.*, vol. 16, no. 3, pp. 451–468, Sep. 2013.

[37] J. J. Durillo, V. Nae, and R. Prodan, "Multi-objective energy-efficient workflow scheduling using list-based heuristics," *Future Gener. Comput. Syst.*, vol. 36, pp. 221–236, Jul. 2014.

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

[39] B. Subramaniam and W.-c. Feng, "Statistical power and performance modeling for optimizing the energy efficiency of scientific computing," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun. Int. Conf. Cyber, Phys. Social Comput.*, Dec. 2010, pp. 139–146.

[40] N. Gholkar, F. Mueller, and B. Rountree, "Power tuning HPC jobs on power-constrained systems," in *Proc. Int. Conf. Parallel Architectures Compilation*, Sep. 2016, pp. 179–191.

[41] E. Gabaldon, J. L. Lerida, F. Guirado, and J. Planes, "Black-list muti-objective genetic algorithm for energy saving in heterogeneous environments," *J. Supercomput.*, vol. 73, no. 1, pp. 354–369, Jan. 2017.

[42] A. Chakrabarti, S. Parthasarathy, and C. Stewart, "A Pareto framework for data analytics on heterogeneous systems: Implications for green energy usage and performance," in *Proc. 46th Int. Conf. Parallel Process. (ICPP)*, Aug. 2017, pp. 533–542.

[43] R. R. Manumachu and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Trans. Comput.*, vol. 67, no. 2, pp. 160–177, Feb. 2018.

[44] 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 Computation: Pract. Exper.*, vol. 31, no. 4, p. 4958, Feb. 2019.
[45] H. Khaleghzadeh, M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on heterogeneous HPC platforms for performance and energy through workload distribution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 543–560, Mar. 2021.
[46] 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.
[47] H. Khaleghzadeh, R. R. 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 Computation: Pract. Exper.*, vol. 35, no. 20, p. 7285, Sep. 2023.
[48] S. Becerra. (2019). *Simple Neural Network Implementation in C++*. [Online]. Available: https://towardsdatascience.com/simple-neural-network-implementation-in-c-663f51447547
[49] A. Khazaei and H. Khaleghzadeh. *Parallel_ANN: Implementation of a Parallel Artificial Neural Network*. Accessed: Dec. 4, 2024. [Online]. Available: https://github.com/UoP-Data/Parallel_ANN
[50] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Nov. 2012.
[51] Y. LeCun. (1998). *The Mnist Database of Handwritten Digits*. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[52] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2506–2518, Sep. 2015.
[53] 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, Jun. 2019.



HAMIDREZA KHALEGHZADEH received the B.Sc. and M.Sc. degrees in computer engineering (software), in 2007 and 2011, respectively, and the Ph.D. degree from the School of Computer Science, University College Dublin, Ireland, in 2019. He is currently a Senior Lecturer with the School of Computing, University of Portsmouth, U.K. His research interests include optimizing performance and energy consumption in massively heterogeneous systems, high-performance heterogeneous systems, energy efficiency, and big data applications.



ATEFEH KHAZAEI received the B.Sc. degree (Hons.) in computer engineering from Imam Reza International University, Mashhad, Iran, in 2009, the M.Sc. degree in software engineering from Shiraz University, Shiraz, Iran, in 2011, and the Ph.D. degree from the School of Computer Science, University College Dublin (UCD), in 2025. She is currently with the School of Computing, University of Portsmouth, U.K. Her research interests include data analysis, overlapping clustering, text mining, high-performance computing, and software security.



ALEXEY LASTOVETSKY (Member, IEEE) received the Ph.D. degree from Moscow Aviation Institute, in 1986, and the Doctor of Science degree from the Russian Academy of Sciences, in 1997. He authored the monographs *Parallel Computing on Heterogeneous Networks* (Wiley), in 2003, and *High-Performance Heterogeneous Computing* (Wiley), in 2009. He is an inaugural ScholarGPS Highly Ranked Scholar-Lifetime (ranked #18 worldwide in Parallel Computing). His main research interests include heterogeneous parallel computing and energy-efficient parallel computing.