Universiteit Leiden

Master Computer Science

Hybrid CNN Partitioning On Edge Devices And Its Effects On Energy, Memory and Inference Latency

Name:	Andreas Savva
Student ID:	s3316391
Date:	17/07/2023
Specialisation:	Advanced Computing and Systems
1st supervisor:	Todor Stefanov
2nd supervisor:	Andy D. Pimentel

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Convolutional Neural Networks (CNNs) have gained widespread adoption in various domains due to their excellent performance in image and voice recognition tasks. However, executing CNNs on resource-constrained devices poses several challenges stemming from their high complexity and size. Traditional approaches such as network compression or offloading to the cloud have serious limitations that hinder their effectiveness. This research focuses on a different approach, namely CNN partitioning and execution on multiple edge devices. The study explores the combination of Vertical and Horizontal Partitioning strategies in a hybrid approach to leverage the advantages of both when executing a CNN in a distributed fashion on resource-constrained edge devices. To this end, we introduce a Design Space Exploration (DSE) process utilizing a Genetic Algorithm (GA) and two distinct chromosomes, along with a specially developed analytical model as the fitness function. Through the experimental results, the effectiveness of the hybrid approach is proven along with the advantages of an advanced chromosome which reduces the search space. Furthermore, the research showcases the usefulness of the analytical model and explores the potential performance improvements achieved by leveraging both CPUs and GPUs in hybrid partitioning mappings. Overall, this work contributes to the advancement of CNN execution on resource-constrained devices, offering a promising solution for achieving efficient distributed CNN inference on edge devices.

Contents

1 Introduction

Convolutional Neural Networks (CNNs) have revolutionized various fields of artificial intelligence, delivering exceptional performance in tasks like image and voice recognition and thus have become ubiquitous. The execution of CNNs can be divided into two parts: training, where parameters are learned from a training dataset, and inference, where input data is processed through the network to produce an output based on the learned parameters. Due to the demanding nature of training, it is performed on powerful machines or clusters. Even though the inference process is much less demanding, the increasing complexity of CNN architectures poses significant challenges when it comes to their execution on resource-constrained devices. In real-world scenarios, there are numerous use cases where executing CNNs on resource-constrained devices becomes crucial. For instance, IoT devices often have to perform inference tasks, such as image recognition, where a resource-constrained controller needs to determine the presence of a person in a security camera feed. Similarly, in automotive applications, CNN inference sessions for navigation and object detection necessitate executing CNNs on resource-constrained devices within a vehicle.

Several solutions have been investigated to address the challenges associated with CNN execution on resource-constrained devices. One approach is to perform model compression which aims to reduce the complexity of the network through techniques such as network pruning or knowledge distillation. However, the downside of such techniques is that they come at the cost of accuracy loss and there is a tradeoff between the compression ratio and the accuracy loss. Alternatively, a solution is to offload inference tasks to the cloud. Such an approach does not suffer in terms of accuracy loss but comes at a cost of latency for transferring the data to and from the cloud. In cases such as automotive applications, latency can become a prohibitive factor. Furthermore, this approach also suffers from privacy issues and data restrictions, where there are situations in which the data cannot be moved to a different geographical location.

Another common approach is the partitioning and execution of the CNN onto multiple devices at the edge, thereby performing distributed CNN inference. This technique involves distributing the network among several devices on the edge and performing inference by sending data to each device. Since the edge devices are in close proximity to the data when compared to the cloud, there are no prohibitive latency or privacy issues. As a result, CNN partitioning at the edge offers a promising approach for executing inference on resource-constrained devices. However, achieving optimal partitioning requires to consider multiple goals, including energy consumption, memory usage, and inference latency. Different partitioning strategies exist, each one with its own advantages and disadvantages. Each strategy's objective is to optimise some or all of the goals, with varying success.

In this research project, the different strategies for CNN partitioning are introduced and studied while focusing on the Vertical and Horizontal Partitioning strategies. Their advantages and disadvantages are explored and the choice of combining the Vertical and Horizontal Partitioning strategies into a Hybrid Partitioning is motivated. Such an approach aims to strengthen the combined strategies while minimizing their limitations. To facilitate the evaluation and selection of the generated CNN mappings based on this Hybrid Partitioning, an analytical model has been developed to grade each mapping based on energy consumption per device, peak main memory usage per device, and inference latency. A Design Space Exploration (DSE) process has been implemented using the NSGA-II [1] genetic algorithm along with two types of chromosomes to explore different mappings using the Hybrid Partitioning approach and search for the optimal mappings. The project is based on the AutoDice [2] framework, which is used to partition the

CNNs and create the implementations of the mappings to run on real distributed systems. The goal of this research project is to answer the following research questions:

- Does the Hybrid Partitioning offer benefits over other partitioning strategies and does it improve upon the Vertical partitioning strategy which has shown great potential?
- Is a naive chromosome able to provide optimal mappings in a reasonable amount of generations using a genetic algorithm? If not, can a suitable chromosome be developed for such a hybrid partitioning strategy?
- Does the Hybrid Partitioning benefit from the use of both CPUs and GPUs in edge devices and is it possible to find optimal mappings when using both?
- Is the development of an accurate enough analytical model possible that can be used during the DSE?

Through the conducted experiments, it is shown that the Hybrid Partitioning strategy has the potential to outperform other strategies, such as the Vertical Partitioning in some aspects. Furthermore, the performance of each developed chromosome is intensely analyzed and the superiority of the advanced chromosome is highlighted. The accuracy of the analytical model is also showcased focusing on its usefulness for the DSE process. Additionally, the benefits of utilizing both CPUs and GPUs in the hybrid partitioning approach have been investigated, aiming to identify optimal mappings that leverage the capabilities of both processing units within edge devices.
The remainder of this thesis is organized as follows: Section 2 presents the necessary background information. Section 3 explores some related work and research in this area. Section 4 describes the methodology employed in this project to answer the research questions formulated above. Section 5 explains the experimental setup and presents the results. Finally, Section 6 provides a discussion on the findings and avenues for future research.

2 Background

In this section, the foundational knowledge required for this master's thesis will be established, providing the necessary background on the key topics that form the basis of our project. The section begins by discussing the concept of edge computing and presenting its significance and applications. Next, it dives into the inner workings of convolutional neural networks (CNNs), their architecture and functionality. The subsequent subsection focuses on the various partitioning strategies available for CNNs, thereby examining how they enable efficient utilization of resources and improved performance. Finally, genetic algorithms are explained and NSGA-II is introduced, which is the specific genetic algorithm employed in this project.

2.1 Edge Computing

Edge computing emerged in the 1990s from Content Delivery Networks (CDNs) that wanted to accelerate performance by using techniques such as prefetching and caching content close to the user [3]. By placing nodes at the edge of the network, closer to users than centralized data centres, CDNs could achieve reduced end-to-end latency and improved bandwidth, both of which are crucial for various applications.
The edge is comprised of a diverse range of devices including network infrastructure, cloudlets and micro-datacenters that collectively form a heterogeneous environment. In this environment, end devices transmit data for processing, while the actual computation takes place on edge devices, as can be seen in Figure 1.
Research on the subject has bloomed, showing different types of use cases for such nodes that are in close proximity to the end user and the many benefits of the paradigm. These use cases include speech recognition and image recognition. Many people argue [4],[5], that nowadays devices that used to only be consumers have become also producers, pushing more and more data to the cloud. Devices like smartphones are one good example, that upload photos and videos to social media. This requires massive amounts of network bandwidth and is a case where edge computing can remove some strain off the network, by first applying compression before uploading content to the cloud. This also gives higher availability by avoiding the cloud, which may be unreachable due to network outages. Moreover, edge computing facilitates the enforcement of privacy policies and restrictions related to data privacy by eliminating the need to transfer data to the cloud. This ensures that data can remain within specific regions or even countries, thus avoiding data leakages and helping to comply with regulations.
All of the aforementioned reasons highlight the advantages of leveraging the edge for offloading CNN inference to resource-constrained devices at the edge. Given the abundance of devices situated at the edge with varying capabilities, partitioning CNNs and executing the inference on these devices presents several benefits that were outlined above.

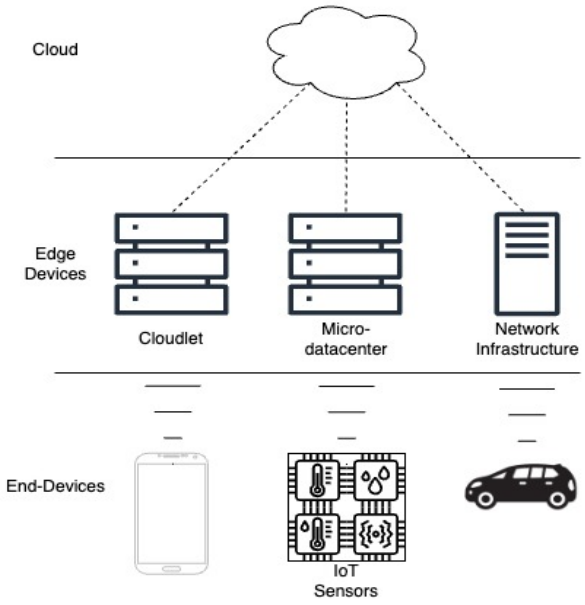


Figure 1: A simple visualization of edge computing showcasing 3 different types of end devices, smartphones, IoT applications and cars, communicating with edge devices to execute computation.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have emerged as a powerful form of deep learning models with multiple use cases, most notably image and pattern recognition, and they can even be used in speech recognition. The basic architecture of CNNs consists mainly of 3 types of layers:

- Convolutional Layers
- Pooling Layers
- Fully-Connected Layers

Convolutional layers are responsible for creating feature maps. They are made up of weights and bias and they perform a weighted sum operation on local patches of the input data. The resulting sum is then passed through a non-linear activation function, commonly the Rectified Linear Unit (ReLU). The convolutional layer, as described in [6], is organized in feature maps and each unit in a feature map is connected to local patches in the feature maps of the preceding layers through the weights. This design is based on the understanding that images exhibit spatially correlated local values, and features within an image can appear multiple times at different locations. By connecting each unit to local patches, the convolutional layer becomes capable of detecting conjunctions or combinations of features.
The second type of layer, the Pooling layer, takes a feature map as input which the layer divides into regions and computes a summary statistic on the regions. Such can be, for example, the Max Pool where it selects the largest item in the selected region. The benefit of this layer is the reduction in dimensionality of the processed data by lowering the spatial complexity which in turn lowers computation and memory requirements. In addition, the Pooling layers help CNNs to capture abstract features and make them more robust to distortions of the input.

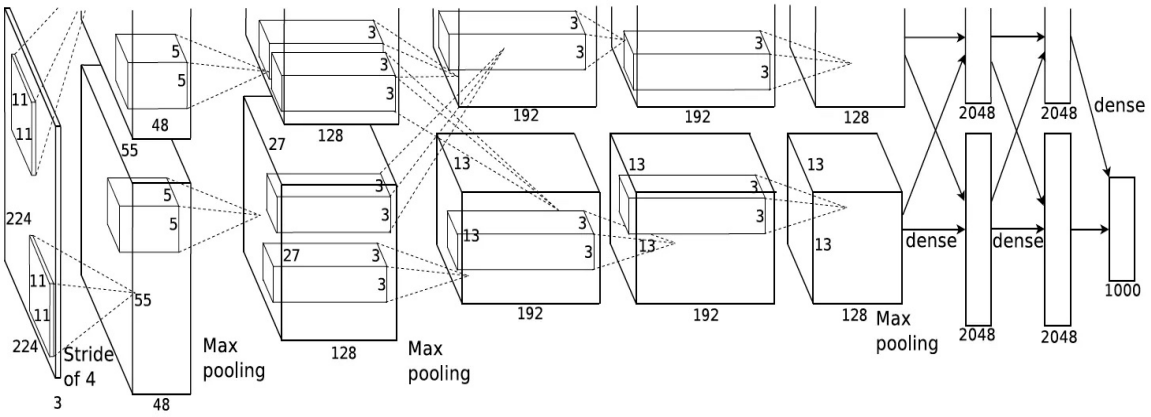


Figure 2: This diagram is taken from the original paper presenting AlexNet [8]. It shows the sequence of layers that are split between two GPUs for training.

Typically, the CNN models are composed of a sequence of stacked convolutional and pooling layers. As explained in [7], the initial convolutional layers focus on detecting low-level features that are used by the subsequent layers together in order to obtain higher-level abstract features. Lastly, Fully Connected (FC) layers are usually situated at the end of the network, after the series of convolutional and pooling layers. These FC layers enable classification and regression tasks by connecting every neuron of the previous layer to the current one which aggregates the information from the spatially localized features for decision-making.
In Figure 2, we can see the topology structure of one of the most influential CNNs, AlexNet, as it was presented in the original paper [8]. Figure 2 presents the stack of Convolutional layers and Pooling layers which is followed by a series of three fully connected layers.

2.3 CNN as a Computational Model

In this thesis, CNNs are treated as direct acyclic graphs (L, E) following the same notation as in [9], where L is the set of layers that are treated as nodes of the graph and E is the set of edges that are the connections between the layers in the CNN. Every layer in the graph, $l_i \in L$, receives input data X_i and produces output data Y_i by executing an operation op_i on X_i. Depending on the type of layer l_i, which is characterized by op_i, layers have different attributes. The characteristics of the Convolutional Layer and the Fully Connected layer are introduced because their notation is required to explain how these layers are split using the Hybrid Partition explored in this thesis.
Convolutional layers execute $op_i = convolution$. They have a sliding window which is the Weights Θ_i along with a stride s_i to go over the input tensor X_i. Furthermore, the areas which are covered by Θ_i can overlap and the X_i tensor can be extended or cropped using padding pad_i in cases where Θ_i cannot cover the whole X_i properly. A bias tensor can be defined as b_i, which is present in some Convolutional Layers and is added while sliding over X_i to form a weighted sum. The tensors of the Convolutional layer have the format $[H, W, C]$ where H is the height of the tensor, W is the width, and C is the number of output channels. Therefore, the complete notation for a Convolutional layer is:

- Input tensor X_i: $X_i^{[H_i^X, W_i^X, C_i^X]}$
- Output tensor Y_i: $Y_i^{[H_i^Y, W_i^Y, C_i^Y]}$
- Sliding window Θ_i: $\Theta_i^{[N_i^\Theta, H_i^\Theta, W_i^\Theta, C_i^\Theta]}$, where N_i^Θ is the number of neurons and $N_i^\Theta = C_i^Y$, $C_i^\Theta = C_i^X$
- Bias tensor b_i: $b_i^{[C_i^b]}$ where C_i^b is the same as the N_i^Θ
- Padding pad_i
- Operation op_i: $op_i = Convolution$

On the other hand, Fully Connected (FC) layers execute $op_i = matrix\ multiplication$. They take an input tensor X_i which is multiplied by their weight tensor Θ_i and a bias term is added from their b_i tensor to produce an output tensor Y_i. The notation for the Fully Connected layer is:

- Input tensor X_i: $X_i^{[H_i^X, 1]}$, where H_i^X is the height of the input tensor
- Output tensor Y_i: $Y_i^{[H_i^Y, 1]}$, where H_i^Y is the height of the output tensor
- Weight tensor Θ_i: $\Theta_i^{[N_i^\Theta, C_i^\Theta]}$ where N_i^Θ is the number of hidden units (neurons) and C_i^Θ is the number of output units
- Bias tensor b_i: $b_i^{[C_i^b]}$ where C_i^b is the same as C_i^Θ

2.4 CNN Partitioning

CNN partitioning involves dividing a CNN model into smaller sub-networks/partitions. This process is driven by the need to deploy the CNN model on distributed environments or resource-constrained devices to execute the inference process. The aim of the CNN partitioning process is the optimization of energy consumption and memory usage of devices as well as the reduction of the CNN inference latency.

Reflecting on the fact that modern CNNs can consist of tens or even hundreds of layers, it may be challenging to execute a CNN on memory-limited devices. Partitioning the CNN on multiple devices reduces the memory consumption on each device, thereby allowing memory-limited devices to cooperate in the inference process.

Another crucial aspect to consider is the energy consumption. Many scenarios exist where the device executing the inference is battery-powered, meaning that the energy budget is limited. Therefore minimizing the energy consumption is essential. When partitioning the CNN model onto multiple devices, we distribute the workload. This means that each device will require less power to execute the inference.

Finally, situations exist where executing the inference on a single device results in prohibitively high inference time. By efficiently partitioning the CNN model, a significant speedup can be achieved through pipelining and parallelization.

The following subsections provide an overview of the four primary approaches to CNN partitioning. Every approach handles the partitioning in a different way, thereby providing different advantages.

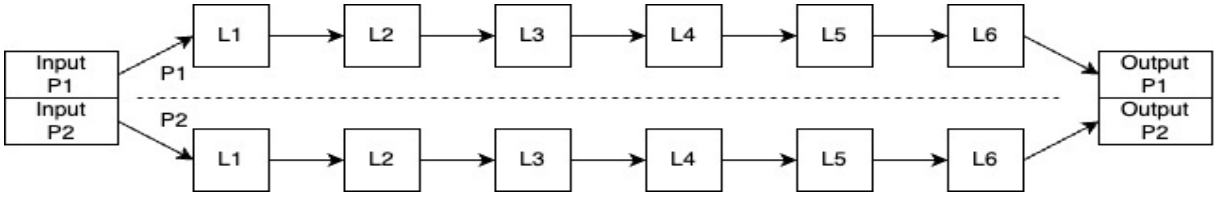


Figure 3: The diagram shows how the data partitioning strategy works with two partitions, using a network with 6 layers. It splits the input into two partitions, each processing it independently.

2.4.1 Data Partitioning

The majority of the computation in a CNN is typically found within the Convolutional or Fully Connected Layers. These layers involve multiplying the input data with the corresponding weights to generate the output. This partitioning strategy takes this into account and splits the input data into chunks rather than the model. Each partition contains all the layers of the CNN with their respective weights, allowing for parallelization of the intensive computation parts. However, this approach introduces dependencies between the layers, requiring synchronized communication to transfer input data across the layers. These dependencies mean that the input of one layer depends on the output of a preceding layer. In addition, it is not suitable for use when the devices are memory limited and cannot fit the whole CNN. The approach is presented in Figure 3 with a network of 6 layers and two partitions. Each partition takes part of the input which is processed independently and produces 1 output.

2.4.2 Sequential Partitioning

When partitioning CNNs, two very important aspects that need to be taken into account are the required communication between partitions and memory utilisation. Taking this into consideration leads us to the sequential partitioning strategy, which aims to maintain consecutive layers within the same partition rather than dividing layer weights or the input. This technique ensures that communication is only required between layers residing on separate partitions, thereby minimizing the overall communication overhead. Additionally, this strategy guarantees that all the required data for a partition, except the initial input of that partition, reside in memory. While this policy does not achieve significant parallelization, it proves useful in scenarios where the devices of a distributed system are memory bound because the network is divided into multiple chunks, thereby lowering the overall required memory on a single partition. Figure 4 presents an example of this strategy using a 6 layer CNN model split into two partitions. Partition *P1* takes the first three consecutive layers and *P2* takes the last three layers.

2.4.3 Horizontal Partitioning

Horizontal Partitioning offers an alternative approach to CNN partitioning, which goes in contrast to the Data Partitioning strategy introduced in Section 2.4.1. While Data Partitioning divides the input data, the Horizontal Partitioning strategy focuses on dividing the layers themselves, specifically the Convolutional and Fully Connected layers, while keeping the input to the layers as a whole. The objective is to create multiple partitions, each containing all the layers but with a reduced size of the two types of splittable layers (Convolutional and Fully

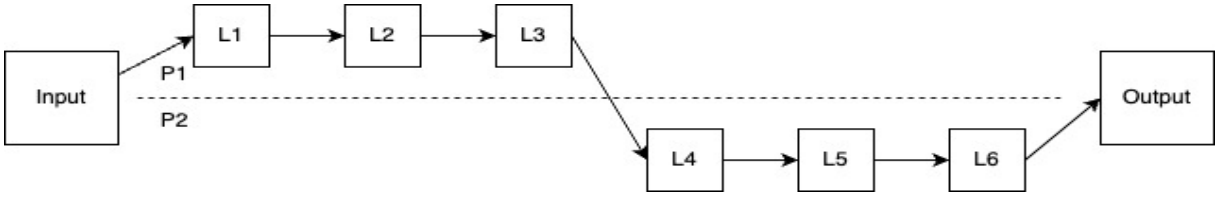


Figure 4: The diagram shows the sequential partitioning strategy with two partitions, using a network with 6 layers. In this example, the first three consecutive layers are given to the first partition while the next three consecutive layers are given to the second partition.

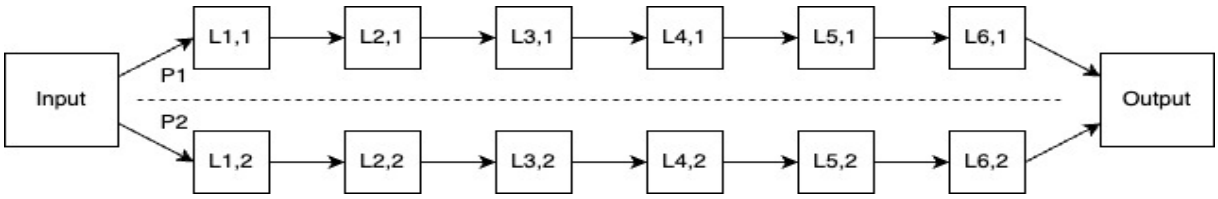


Figure 5: The above diagram presents a CNN with 6 layers and two partitions. Each layer is divided and a piece is given to the corresponding partition.

Connected layers). As a result, the memory requirements for each partition are reduced as the partitions store only a portion of the weights of the splittable layers. Moreover, the bulk of the computation is parallelized, as multiple devices collaborate to generate the output. On the other hand, this strategy introduces overhead in the form of synchronized data communication where partitioned layers exist. This is because their outputs need to be concatenated to form the correct result before passing it on as input to the next layer. The strategy is presented with an example in Figure 5, where a CNN with 6 layers is divided into two partitions by splitting its layers.

2.4.4 Vertical Partitioning

Vertical Partitioning, similar to the Sequential Partitioning introduced in Section 2.4.2, does not involve splitting of the input or layers into chunks. Instead, it focuses on partitioning the network itself as a whole. However, unlike Sequential Partitioning, it does not aim to keep consecutive layers together within each partition. Each partition contains a subset of layers that are not necessarily consecutive and the input and layer weights are kept untouched. By distributing the CNN model across partitions, each device only needs to store a subset of the layers, resulting in reduced memory usage. In addition, this technique can benefit from reduced data exchange between devices, further optimizing the memory utilization. Furthermore, depending on the specific scenario and CNN network, a small speedup can be achieved through task parallelization and pipelining. An example of the strategy is presented in Figure 6 which shows a 6-layer CNN partitioned into two devices. Notice that layers on the partitions are not necessarily consecutive, for example, *P2* executes *L1*, *L4* and *L5* and *P1* executes *L2*, *L3* and *L6*.

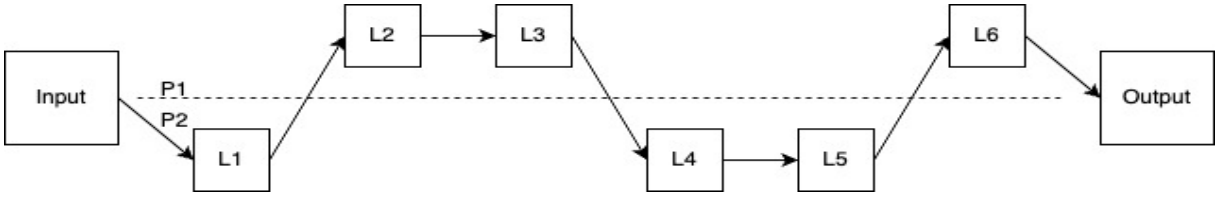


Figure 6: A 6-layer CNN network which is partitioned using Vertical Partitioning between two devices.

2.5 Genetic Algorithms

Genetic Algorithms draw inspiration from the principles of biological evolution. In this theory, individuals placed in a resource-limited environment compete, adapt, and reproduce, resulting in the selection of the fittest individuals to define the next generation's population. Similarly, genetic algorithms are employed to address a wide variety of problems within a similar context. Based on [10], the basics of Genetic Algorithms (GA) can be defined. First, a population is initialised randomly, consisting of individuals that compete for limited resources. Through natural selection, only the fittest individuals survive and progress. These individuals represent solutions to the problem at hand, defined in a format that can be manipulated by computers. They are often referred to as genotypes or chromosomes, encoding the actual characteristics or phenotypes. It is important to note that the population in most cases is kept constant, which emulates the limited resources. An evaluation function is used in order to define the quality of the individuals. To advance the population, two selection operators are employed. The first operator is called Parent Selection Operator and it is responsible for the parent selection which creates children for the next generation. The higher the quality of the individuals, the higher the chance of selection as a parent. The second operator is the Survivor Selection Mechanism, applied after the parent selection and creation of offspring, based on the quality of the individual decides the survival to the next generation. To introduce variation and promote adaptation, two variation operators are utilized. The mutation operator is applied to a single individual and creates a new child called a mutant which depends on random choices made on that individual's genotype. The second operator, called recombination or crossover, involves two individuals whose information is randomly selected and merged to create one or two children.

2.5.1 NSGA-II

The Non-dominated Sorting Genetic Algorithm 2 (NSGA-II) [1], is an extension of NSGA [11] designed for solving multi-objective optimization problems, that involve multiple objectives to be optimized simultaneously. Each solution in the problem is assigned fitness values for each objective. Dominance is a key concept in NSGA-II, and is defined as follows: a solution is considered dominant if it has at least the same or higher scores for all objectives, and for at least one objective higher score compared to another solution. A solution is non-dominated if no other solution exists which satisfies the above definition, therefore the solution cannot be improved in any aspect without compromising another. The set of all non-dominated solutions is referred to as the Pareto front.
In NSGA-II offspring is created from the current population, followed by the non-dominant sorting process which organizes individuals into different fronts based on their level of non-domination. Equation 2.1 presents the decision process that chooses which individuals proceed to the next generation. The algorithm first advances individuals in ascending order of non-

$$i \prec_n j \ if \ (i_{rank} < j_{rank}) \ or \ ((i_{rank} == j_{rank}) \ and \ (i_{distance} > j_{distance})) \tag{2.1}$$

The equation describing the partial rank $\prec_n$ for the advancements of individuals to the next generation in NSGA-II as presented in [1]. Denoted as *rank* is the non-dominant rank and *distance* is the crowding distance of the individual.

domination rank. However, since the size of the population is N, if a front cannot advance all individuals, a crowding distance measure is used to select the individuals that will advance, ensuring diversity in the population. This measure is an approximation of how closely solutions are located to each other and is determined by calculating the average distance between pairs of points on both sides across each objective. For each objective, individuals are sorted based on their fitness values, and the minimum and maximum individuals are assigned an infinite distance, while the others are assigned the absolute normalized difference of the objective values of the left and right individuals. The final distance of a solution is calculated as the sum of the distances of each objective. Individuals are selected to advance in descending order of their distance until the total population size reaches N at which point the algorithm proceeds to the next generation. NSGA-II combines non-dominated sorting, crowding distance, and selection based on dominance to efficiently identify a diverse set of non-dominated solutions for multi-objective optimization problems. Furthermore, it is versatile allowing it to be applied to a wide range of problems and it has been extensively studied and tested throughout the years proving its effectiveness.

2.6 AutoDice

This thesis is based on the AutoDice framework [2] which facilitates automated CNN partitioning into multiple sub-models along with the code generation, packaged for collaborative inference execution on edge devices. Furthermore, the framework goes beyond basic code generation for the devices and leverages OpenMP [12] and MPI [13] directives in order to take advantage of parallelism opportunities while supporting GPU acceleration by implementing VULKAN [14] and CUDA [15] APIs.
To utilize the framework, four input specifications are needed. First of all, a trained CNN model to be partitioned needs to be provided in the ONNX [16] format. Secondly, a Platform Specification file in .txt format should be provided, which includes a list of all available devices. Finally, a Mapping Specification composed of two .json files has to be provided. The first mapping file consists of key-value pairs with the device as the key and a list of layers as the value, indicating which layers each device will execute. If a layer is present on more than one device, the layer is horizontally partitioned and the second file, horizontal.json, needs to be provided. In the horizontal.json file, the keys represent the names of the layers to be horizontally partitioned and the values are lists containing the type of horizontal partition, split points, and the devices to be used. Based on the aforementioned input specifications, the framework will partition the provided CNN model into multiple CNN models, one for every device that is present in the mapping files. Next, the system generates a *MPI-rankfile*, which is used to map the previously generated models onto devices and processing units. Additionally, it creates the *sender.json* and *receiver.json* files, which define the communication interactions between processes. The *sender.json* file contains key-value pairs indicating that for instance, process X needs to send the data of buffer B to process Y. The *receiver.json* contains the

corresponding key-value pairs indicating which process waits for what input and from where. In the end, the framework generates the necessary code for CNN inference on edge devices. It produces a single .cpp file containing separate blocks of code belonging to each process. The code is subsequently compiled and packaged with the *MPI-rankfile* and sub-models, ready to be distributed to each device.

As one can see, the framework hides the total complexity of the process for generating all the required code for distributed CNN execution. Furthermore, it manages all dependencies, compilation, and it supports a wide range of CNN partitioning techniques. These qualities make it an excellent choice for this project, as it simplifies the actual implementation process and allows for fast testing of mappings on the real distributed systems. Last but not least, the code generated by AutoDice offers state-of-the-art inference performance by not only leveraging the capabilities of various APIs, including OpenMP [12], MPI [13], CUDA [15], and VULKAN [14] but also by incorporating platform-specific implementations of layer operations that further enhance the efficiency and effectiveness of the generated code.

3 Related Work

This section reviews some of the prior research related to CNN partitioning. CNN partitioning has received significant attention, with researchers developing new methods and techniques. However, this project stands out by addressing all three crucial design objectives: energy consumption per device, main memory usage per device, and inference latency. Additionally, while most existing projects rely on a single partitioning technique, our developed method combines two techniques in a new hybrid to take advantage of the benefits of both.

The work in [9] is an in-depth investigation of the four basic partitioning strategies we mentioned before in Section 2.4. The study focuses on the energy consumption effects of the strategies and provides useful analytical models for each technique to model the consumed energy along with computation and communication. In contrast, our project explores the effects of a hybrid partitioning strategy. Furthermore, it not only investigates the energy consumption but also the performance and memory consumption, while providing two different chromosomes for Design Space Exploration(DSE) and testing them on a wide range of cases.

In [17], the authors introduce the vertical partitioning strategy and primarily explore its potential for reducing the memory consumption per device and improving inference performance. Their goal is to evenly distribute the memory usage on the devices while we are focused on improving the aformentioned three different objectives. While our work uses the well-established NSGA-II along with a naive and an advanced chromosome, [17] uses a naive chromosome with a simple custom GA. Lastly, the main difference of our work is that it advances the proposed approach in [17] by combining it with Horizontal Partitioning.

The related work in [18] introduces a 2-stage DSE method with NSGA-II, first utilizing an analytical model as the fitness function, followed by evaluations on real devices at the second stage. Similarly to this thesis, [18] introduces a novel chromosome to explore mappings, although it can handle only the Vertical Partitioning strategy. In addition, [18] uses the AutoDice framework which is also used in our work, but the experiments are mainly focused on the DSE strategy using only ResNet. In our case, we use a single-stage DSE for our experiments but evaluate our approach on three different CNN models, utilizing both a novel advanced and a naive chromosome in the genetic algorithm.

The Data Partitioning strategy is explored in [19] which proposes a new technique for data partitioning called fused tile partitioning. This technique splits the input of convolutional layers on edge devices in such a way which minimizes the communication between them. In the strategy, a stack of convolutional and pool layers is parallelized which not only optimizes the latency but also the communication time and memory requirements. On the other hand, [19] is only focused on splitting the first few layers while we split the whole CNN model. Furthermore, the approach in [19] also requires a powerful gateway device which executes the merging of the intermediate data after the partitioned layers as well as the rest of the layers.

The papers [20] and [21] propose techniques that use the horizontal partitioning strategy. [20] uses mobile devices connected via WiFi Direct. The authors propose a framework using map-reduce for parallelization, where multiple workers handle specific parts of the layer input channels while a master device handles the merging. The paper focuses mainly on partitioning Convolutional Layers and Sparse Fully Connected layers. In contrast, our hybrid partitioning approach is not focused on specific layers and most importantly, it does not rely on a master node. Furthermore, while [20] only experiments with VGG, we also experiment on AlexNet and DenseNet. Last but not least, [20] is not concerned with the energy efficiency of the solutions.

[21] is a framework for CPU-GPU co-execution which uses horizontal partitioning. The authors

conduct an in-depth analysis of other available solutions and identify their bottlenecks. Two very important and relevant bottlenecks to our work are the overhead for data sharing along with uneven partitioning of the workload caused by inaccurate models. Furthermore, [21] explains that splitting a layer horizontally using the height dimension gives the benefit of the partitions requiring fewer data because they do not use the whole feature map, while splitting the Output Channel dimension gives better resource utilization. The framework in [21] uses horizontal partitioning along with chains meaning that a partition does not contain a single layer but a series of partitioned layers whose result is merged later. In contrast, our work does not employ such chains but uses horizontal partitioning on the output channel along with vertical partitioning. [21] is focused on CPU-GPU co-execution in order to lower the overall inference latency while our thesis is focused on partitioning CNN models and executing inference on multiple edge devices to not only lower inference latency but also the per-device energy consumption and main memory usage.
Finally, the papers [22] and [23] propose hybrid techniques for partitioning as in our thesis. [22] is an extension of [19] which takes advantage of both layer input partitioning (LIP) and layer output partitioning (LOP) that both belong to the Horizontal Partitioning strategy. This hybrid approach can split two consecutive layers without the need to merge because the first layer can be split using LOP and the second using LIP. Unlike [19], it also supports the partitioning of Fully Connected layers. [22] lacks in-depth experimental evaluation though. It only experiments with YOLOv2 using emulated devices. As opposed to our approach, it also requires a device that will collect the results from the first layers in order to re-distribute them later. On the other hand, [23] proposes and implements a new inference framework which uses dynamic scheduling. It combines Data Partitioning with Sequential Partitioning in order to minimize the data dependencies from one layer to the next. Unlike our method, it uses a master-worker approach, requires user input to decide the number of partitions, and is not concerned with energy efficiency.

4 Methodology

In this section, the methodology used in this research project is introduced. First, it dives into the Hybrid Partitioning strategy, providing an overview of its design and function, and discussing the considerations that guided its development. Next, it explains how NSGA-II is utilized for Design Space Exploration (DSE) along with the two types of developed chromosomes, examining their respective strengths and limitations. Finally, we delve into our analytical model which is a crucial part of the DSE process.

4.1 Hybrid Partitioning

In this research project, a novel hybrid partitioning approach has been developed that combines two prominent strategies mentioned in Section 2.4. The motivation behind this combination is to leverage the strengths of both strategies while mitigating the inherent limitations present in each strategy when employed in isolation. By combining two main partitioning strategies, this work aims to create a more comprehensive and robust partitioning scheme which can effectively address the challenges of energy consumption, memory utilization, and latency associated with CNN inference at the edge.

Based on the four different main strategies introduced in Section 2.4, there exists a multitude of potential combinations that could have been used to formulate a hybrid partitioning strategy. For this research, the Horizontal Partitioning using Layer Output Partitioning and the Vertical Partitioning strategies have been chosen. Although at first, this choice may seem arbitrary, it is made after careful consideration of the different strategies and an elimination process which involves assessing their characteristics and limitations.

When considering the Data Partitioning Strategy, it becomes evident that one of its notable advantages lies in its potential for parallelization. By dividing the data among multiple devices, this strategy processes the input in parallel which can also contribute to potential energy savings because the total time spent computing on each device is reduced. However, it is important to note that it does not provide any benefits in terms of reducing memory usage as it requires the undivided weights to be present on each device. Another drawback of the strategy is the introduction of high communication overhead. Due to the need for synchronization and merging of results from each device, significant communication and computational overhead is introduced. The process of splitting the data and distributing it to each device adds further complexity and communication requirements.

The Sequential Partitioning Strategy offers a different approach compared to the Data Partitioning Strategy, particularly in terms of communication and memory requirements. This strategy reduces the overall communication overhead by only necessitating data sharing when transitioning between devices, and reduces memory requirements per device depending on the number of partitions that are created. This presents an important trade-off because creating a high number of partitions lowers the memory usage per device but increases the communication overhead and vice versa. Additionally, it is important to state that the amount of parallelization exploited when using this technique is very limited, except for potential pipelining benefits.

The Vertical Partitioning Strategy offers distinct advantages over the Sequential Partitioning strategy. While both strategies aim to divide the network into sub-graphs, vertical partitioning provides greater flexibility and a wider range of benefits. One key advantage of the Vertical Partitioning strategy is its flexibility in dividing the network into non-consecutive parts. This provides greater granularity in terms of possible mappings since it does not have the strict re-

quirement of consecutive layers within a partition. By identifying layers that can be executed in parallel or through pipelining, vertical partitioning can effectively exploit parallelization opportunities and optimize overall performance. Furthermore, due to its flexibility, it can distribute the workload and memory requirements evenly among the partitions.

When examining the Horizontal Partitioning Strategy, we face four possible options for splitting a layer's weight tensor. As the weights of a convolutional layer $l_i \in L$ are represented by $\Theta_i^{[N_i^\Theta, H_i^\Theta, W_i^\Theta, C_i^\Theta]}$, we have the flexibility to choose the dimension along which we want to perform the split, each one with its own opportunities and drawbacks. This can be better visualized when looking at Figure 7a, which illustrates all the dimensions of the weight tensor of a convolutional layer along with the output. On the left side of the figure, we see the weight tensor, which is comprised of the kernels grouped into input channels. Every group in the weight tensor will create an output channel, which is shown on the right side of the figure visualizing the output tensor Y_i. First, considering partitioning by height (H_i^Θ) involves dividing the tensor into multiple parts while keeping the input channels, width, and output channels intact. Each device would receive a portion of the tensor and compute the corresponding segment of the feature map. However, this technique introduces complexity due to the nature of the convolution operation. In order for every device to compute its share of the feature map, intermediate data are required to be sent between the different partitions, thereby introducing synchronization and communication overhead. After finishing each part of the feature map, each device again needs to send the results back in order to merge and create the complete result. The same characteristics apply to partitioning on the width dimension (W_i^Θ) as well.

Next, consider the Layer Input Partitioning (LIP) technique, which involves splitting the weight tensor based on its input channels (C_i^Θ). By partitioning the input channels, each device receives a portion of the weight tensor, while maintaining the total height, width, and output channels for all partitions. This partitioning does not require any intermediate data to be transferred because the segments that each device will be computing are independent. However, it introduces communication and complexity because the input needs to be divided appropriately for each device to compute its section. Once the result is available, it again needs to be transferred back for merging.

The last Horizontal partitioning option is the Layer Output Partitioning (LOP) which involves splitting the output channels (N_i^Θ) among different devices while preserving the input channels, height, and width. Figure 7b shows a convolutional layer's weight tensor split Horizontally with LOP into two partitions. Similar to LIP, this technique does not introduce any intermediate results that require synchronized communication and only requires the result of each device to be communicated back but this time for concatenation. Furthermore, this approach does not require partitioning of the input, as each device operates on the complete input. Consequently, LOP simplifies the partitioning process and reduces the communication overhead compared to the other horizontal partitioning techniques mentioned above.

Now, after the aforementioned analysis of all basic partitioning strategies, the process of eliminating strategies can begin. The first strategy to be eliminated is Data Partitioning because it only offers parallelization but introduces high communication overhead and no improvements in memory consumption. Next, the Sequential Partitioning strategy is disqualified in favour of the Vertical Partitioning which offers higher flexibility and a wider range of advantages. Therefore, two basic strategies are left, namely Vertical Partitioning and Horizontal Partitioning. Using the same process of elimination, the choice of the type of Horizontal partitioning can be made. Weight tensor partitioning on the height/width dimension can be easily eliminated because it requires high communication overheads. A closer look needs to be taken into the

aspects of LIP and LOP in order to make a choice between them. It is clear that both types of horizontal partitioning require an equal amount of computation on each device to generate the result. This is because they do not modify any of the required operations in the convolution operation because the height and width will be the same. Splitting the channels in both cases only lowers the amount of convolutions that need to be computed on each device. The same goes for the memory consumption. Since both techniques split channels, we can create mappings with both that require the same memory for every device. Their crucial difference lies in the communication overhead. Equations 4.1-4.4 describe the required data communication in both LOP and LIP techniques, assuming that the device which sends the input and receives the output back for merging or concatenation is different from the ones which execute the convolution. Equations 4.1 and 4.2 give the input and output data size respectively, assuming a 4-byte float for every number in the input/output tensors. Looking at Equation 4.3 for the communication required in LOP, we see that we need to send the total bytes for the input N times, where N is the number of partitions created for the layer. The second part of the equation gives us the output size communication overhead. Since each device computes $(OS/N)*4$ bytes, the total communication will be equal to the total output size. Equation 4.4 shows the communication for LIP. Since each device needs to receive $(IS/N)*4$ bytes for input the total data transferred will be equal to the total input size. The output data size communication overhead is equal to the output data size multiplied by the number of devices that execute the convolution because each device will take part of the input channels but will produce the normal output tensor that needs to be merged. Equations 4.3 and 4.4 are very similar, each one having an opposite part multiplied by N. If we examine closely some of the convolution operations that usually take place, the number of output channels is much greater than the number of input channels, especially in the early stages of the CNN models. Therefore, the total communication for LOP will be lower than for LIP.

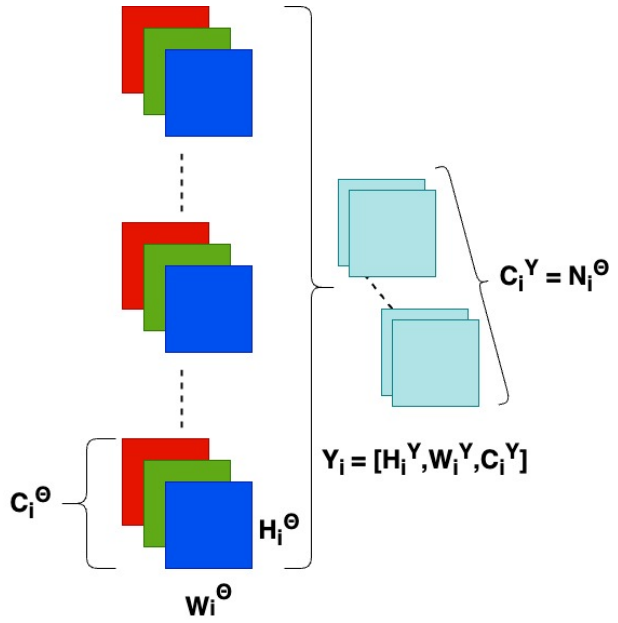
$$InputSize : IS = (H_i^X * W_i^X * C_i^X) * 4 \tag{4.1}$$

$$OutputSize : OS = (H_i^Y * W_i^Y * C_i^Y) * 4 \tag{4.2}$$

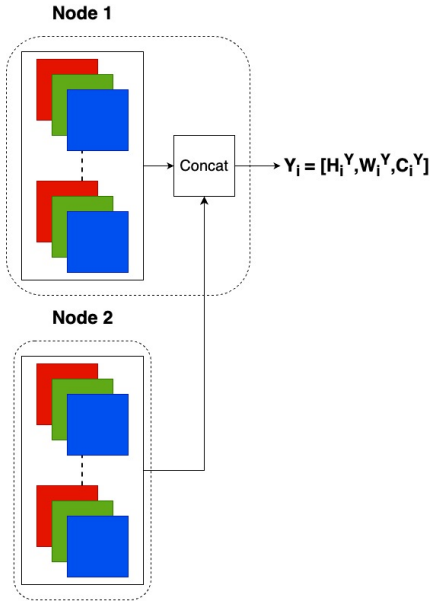
$$Comm^{LOP} = IS * N + OS \tag{4.3}$$

$$Comm^{LIP} = IS + OS * N \tag{4.4}$$

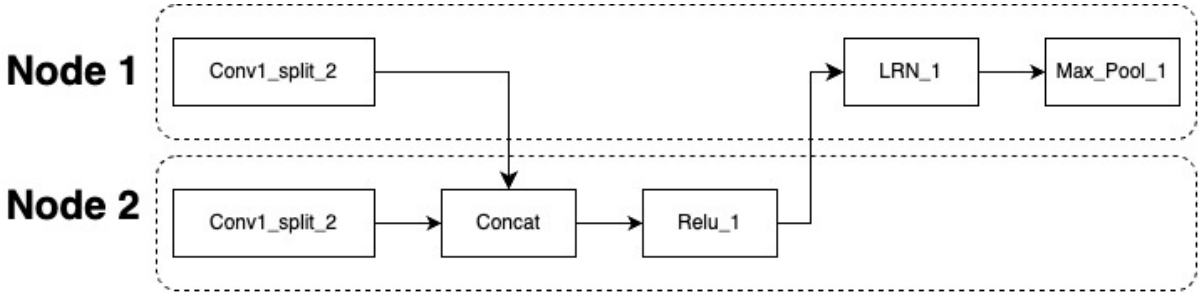
Based on the analysis provided above, our choice of combining the Vertical Partitioning with the Horizontal Partitioning using the LOP technique is motivated and justified. Both the Vertical Partitioning and Horizontal Partitioning strategies outperform the Data Partitioning and the Sequential Partitioning, making them the more favourable options. Among the Horizontal Partitioning techniques, LOP and LIP are better when compared to Height/Width weight partitioning. The decision to choose LOP over LIP is motivated by the slightly lower data communication overhead. Combining these two strategies provides several advantages. Firstly, the memory requirement per device can be evenly distributed thanks to both strategies. Vertical Partitioning gives us the ability to exploit pipelining and low communication overhead while distributing the computation load required for layers that cannot be partitioned with Horizontal Partitioning (layers which do not use weights, e.g Pooling or LRN). Secondly, our choice of Horizontal Partitioning further lowers the memory requirements, latency, and communication overhead while exploiting parallelization of the most computationally demanding layers. Figure 7c illustrates the result of our Hybrid Partitioning of the first four layers of AlexNet into two partitions. It shows that the first convolution layer is split into two parts, the results of which are concatenated on Node 2 which also executes the Relu layer. Then, Node 1 continues the



(a) Convolution Layer Weight Tensor



(b) Horizontal Partitioning LOP of a Convolution Layer



(c) Hybrid Partitioning: Horizontal & Vertical

Figure 7: Figure 7a shows the anatomy of a convolutional layer's weight tensor with notation. Figure 7b shows how horizontal partitioning LOP splits the weights of a convolutional layer. Figure 7c gives a small example of the first four layers of AlexNet when partitioned with our Hybrid Partitioning approach.

inference by executing the LRN and Max Pool layers. The horizontally partitioned convolutional layer is similar to the example in Figure 7b.

Lower overall main memory usage is expected to be achieved when compared to other techniques by employing this hybrid approach. Additionally, we foresee that the inference latency will be reduced due to the lower communication overhead and increased parallelization. Finally, due to the anticipated lower latency and the even distribution of memory and computation on the devices, lower energy consumption is expected to be observed on each device.

4.2 NSGA-II based DSE

The DSE methodology used in the project, which exploits NSGA-II to find optimal CNN mappings based on the Hybrid Partitioning, is introduced. Our methodology takes a CNN in Direct Acyclic Graph (DAG) format with (L, E) and partitions it into a maximum of n sub-graphs, where n is equal to the number of devices, $n = |Devices|$, $Devices = \{device_1, \cdots, device_n\}$. A specific Convolutional or Fully Connected layer can exist in multiple sub-graphs. The amount of sub-graphs that the layer is present determines the amount of LOP splits that will be made on the layer's weights. A partitioning is denoted as ${}^kL = \{{}^kL_1, \cdots, {}^kL_k\}$ where $k <= n$, each ${}^kL_i \epsilon {}^kL$ contains a subset of the layers in L. Multiple subsets can contain the same layer at the same time such that ${}^kL = {}^kL_1 \cup {}^kL_2 \cup \cdots \cup {}^kL_k$, $x = {}^kL_y \cap {}^kL_z$, iff $x \epsilon L$ and $op_x =' Convolution'$ or $op_x =' FullyConnected'$, where ${}^kL_y \epsilon {}^kL, {}^kL_z \epsilon {}^kL$. The following example should make the notation clear: We have a CNN with 5 layers, of which the first is a Convolution and the last is Fully Connected, $L = \{l_1, l_2, l_3, l_4, l_5\}$. If we have four devices then $Devices = \{device_1, device_2, device_3, device_4\}$. A valid CNN mapping could be: ${}^3L = \{{}^3L_1, {}^3L_2, {}^3L_3\}$ and ${}^3L_1 = \{l_1, l_2, l_3\}$, ${}^3L_2 = \{l_1, l_4, l_5\}$, ${}^3L_3 = \{l_5\}$ which is presented in Table 1. Therefore, this particular mapping gives $Device_1$ the first three layers as shown in the second column, $Device_2$ the first layer, fourth and fifth, shown in the third column, and $Device_3$ only gets the last layer in column 4. This means that the first layer will be horizontally partitioned between $Device_1$ and $Device_2$ while the last layer is horizontally partitioned between $Device_2$ and $Device_3$. In addition, $Device_4$ does not run any layers which is the reason that column 5 in Table 1 is empty, meaning that it is not used in this particular mapping.

Table 1: Example hybrid mapping of a CNN with five layers on four devices.

Devices	$Device_1$	$Device_2$	$Device_3$	$Device_4$
Mapping	l_1,l_2,l_3	l_1,l_4,l_5	l_5	

In our project, two distinct methods to perform the Design Space Exploration (DSE) are developed. The first method utilizes a naive chromosome which is not highly efficient as it does not exclude any possible inefficient mappings from the Design Space. For this reason, we have developed a more advanced chromosome which limits the search space in an attempt to find optimal solutions faster. Both of these chromosomes are used in combination with NSGA-II which uses our custom analytical model as a fitness function and will be presented in Section 4.3. Using NSGA-II, our goal is to minimize three conflicting objectives: energy consumption per device, peak main memory usage per device, and the latency of CNN inference. It is important to note here, that the chromosomes that will be presented do not select where the Concat operation will be placed for the concatenation of the results after a horizontal split. AutoDice handles the placement of the Concat operation and attaches it to the device which will execute the next layer. If the layer being partitioned is the last, then the device executing the previous layer will perform the Concat operation. This is done in order to further minimize the communication between devices. Each design point (mapping) is evaluated using the analytical model and the result is returned back to the GA in order to determine the mutations, parents to generate offspring, and the next generation until the search converges.

4.2.1 Naive Chromosome

Our initial approach to devise a chromosome involved developing a naive encoding method that could potentially capture and explore the entire Design Space. Such a naive encoding is

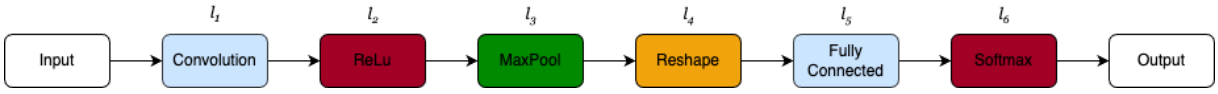
also used in [17], where a simple chromosome is introduced that is able to encode a vertically partitioned CNN without excluding any possible partitioning solution during the exploration. In our case, the development of a way in which the chromosome could also encode Horizontal Partitioning of a Convolutional Layer or Fully Connected layer, in addition to Vertical Partitioning of layers onto devices, is needed.

Our simplest solution is to statically decide how many horizontal splits are allowed at each splittable layer, say x splits. When considering the Vertical Partitioning in its simplest form, the most naive chromosome has one position for every layer of the CNN model and the value of each position in the chromosome is the device which is assigned to execute the layer. The positions of the chromosome follow the same order as the layers of the CNN model, meaning that the indices of the chromosome correspond to the indices of the CNN layers. Thus, adapting this approach to support the combination of Vertical Partitioning along with x Horizontal splits on Convolutional and Fully Connected layers, yields a chromosome which has x positions for every splittable layer and one position for every non-splittable layer. Consequently, if $x = 4$ and a CNN has 3 splittable layers and 4 non-splittable layers, the chromosome's length would be 16. Rather than the user deciding the number of allowed splits, the solution was abandoned in favour of giving every splittable layer n positions, where $n = |Devices|$. This way, the GA is allowed to decide the number of splits at each splittable layer. If the GA assigns a different device for all n positions of a particular splittable layer, then that layer is split Horizontally n times. The GA can decide to split a specific splittable layer less than n times by assigning a device multiple times in the n positions of the layer. Using this approach, the chromosome still follows the same order as the layers of the CNN. Based on the design described above, the chromosome's length is given by the following equation:

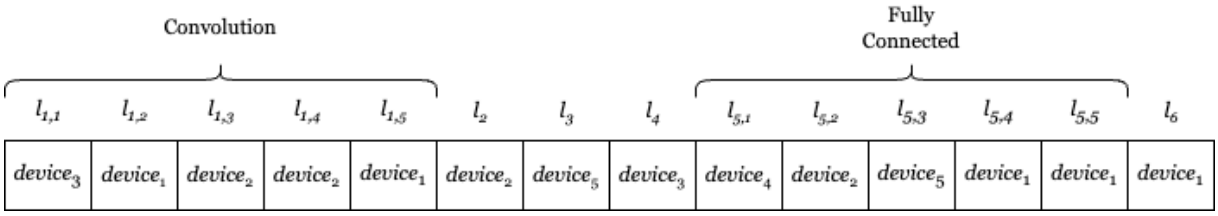
$$Length = (sp * n) + ol - sp \tag{4.5}$$

where sp is the number of splittable layers and ol is the original number of layers in the CNN. Thus, the GA will give a sequence of numbers, each number indicating the $device_{id}$, and the sequence is the same sequence as the CNN's layers, only now it is expanded to accommodate the possible splits on each splittable layer. Figure 8a shows an example CNN model that contains 6 layers. One of those layers is a Convolutional layer and one is a Fully Connected layer. Figure 8b shows an example of the naive chromosome used to encode a mapping of the CNN model in Figure 8a. If we look at the first five positions in the chromosome shown in Figure 8b, we understand that those are the splits of the first layer, which is the Convolutional layer, and that the GA decided to only split it 3 times, because $device_1$ and $device_2$ are used twice and $device_3$ is used once in the 5 available positions. In addition, layer 2 which is the ReLU, is assigned to $device_2$, layer 3 which is the MaxPool is assigned to $device_5$ and layer 4 is assigned to $device_3$. This means that these three layers are vertically partitioned. The Fully Connected layer is next which is mapped to $device_4$, $device_2$, $device_5$ and $device_1$, and thus means that it is Horizontally partitioned into 4 splits. Finally, layer 6, the Softmax layer, is mapped to $device_1$.

This chromosome is described as naive because it quickly results in an exponential increase in the search space when devices or splittable layers are added. If we take AlexNet for example, and only split Fully Connected layers horizontally using five devices and the rest of the layers vertically, the length of the chromosome, according to Equation 4.5, will be 36 because the CNN model has 24 layers and 3 of them are Fully Connected splittable layers. If we also add the Convolutional Layers as splittable, the total chromosome length becomes 56. Therefore, the total Search Space size will be 5^{56}. This might not be a problem for AlexNet which is a



(a) An example CNN network.



(b) A mapping of the CNN network shown in Figure 8a.

Figure 8: Figure 8a presents a 6 layer CNN network containing one Convolutional layer and one Fully Connected layer. Figure 8b presents an example mapping of the CNN network in Figure 8a using the Hybrid Partitioning strategy that is encoded using the naive chromosome.

relatively small model, especially if we only use five devices, but when considering larger models and more devices, e.g. DenseNet, the search for Pareto-optimal solutions will not converge in a reasonable time when using the naive chromosome.

4.2.2 Advanced Chromosome

The need for a more efficient encoding in a chromosome arises when exploring mappings of larger CNN networks onto a higher number of devices, as the size of the Design Space becomes enormous. Several ways have been researched over time for similar optimization problems, such as the conversion of the search space from discrete to continuous [24] or Monte-Carlo Tree Search to divide the search space [25].
The naive chromosome, by its nature, does not exclude any possible mappings from the search which also includes inefficient mappings. However, since the requirements of the search are known, it is possible to define what constitutes a good and potentially efficient mapping and immediately identify and discard the inefficient mappings without exploring them. By focusing only on favourable and potentially efficient mappings, the search space is significantly reduced. In the context of this project, a potentially efficient mapping should aim to reduce data communication time and evenly distribute main memory usage and workload among devices, which will consequently reduce energy consumption per device, main memory usage per device, and inference latency.
The main point of CNN hybrid partitioning strategy lies in distributing horizontally splittable layers across multiple devices while also evenly distributing the remaining non-splittable layers. Vertical partitioning of layers among devices to minimize communication and evenly distribute memory is achieved in [18] with the introduction of the Split Point encoding chromosome. A similar encoding approach is adopted in this research by extending the chromosome Split Point encoding in [18] to include both Horizontal and Vertical Partitioning strategies. A split point is defined in [18] as a number indicating a layer in the CNN which predecessor layers after the previous split point or from the start of the CNN, up to that point are mapped onto the same device. This means that with only a few split points a CNN can be vertically partitioned on multiple devices. In [18], the length of the chromosome is equal to the number of devices, thus the positions in the chromosome correspond to the devices and the split point number

written in each position means that the corresponding device is responsible for the layers up to and including that point.
The extension of this Split Point encoding technique was made possible by recognizing that it could be used to encode the Horizontal Partitioning, and by tweaking the definition of the split point, and introducing more restrictions it can be further adapted to work with the Hybrid Partitioning. Our advanced chromosome is based on the fact that the CNN network is expressed as a DAG, therefore the layers have a specific order. The restrictions applied are the following:

1. Every splittable layer can have a maximum of p horizontal splits, $p <= n$, where n is the number of devices.

2. Non-splittable layers after a splittable layer are assigned to the same device as the first split of the splittable layer.

The first restriction applied means that every splittable layer can have a maximum of p horizontal splits and p is decided beforehand by the user. This restriction lowers the size of the Design Space by not allowing the GA to explore mappings which have more horizontal partitions than p, which can be lower than n. It also lowers the number of symmetric mappings that will be explored. This is because, in the naive chromosome, every splittable layer gets n positions in the chromosome, and the GA may decide that a layer will be partitioned horizontally 3 times for example. If n is 5, then there are many symmetric mappings that can be created. Using this restriction, if p is lower than n, the amount of possible symmetric mappings is lowered. The second restriction applied is that the GA does not directly decide where the layers are vertically partitioned. The naive chromosome allows for every individual non-splittable layer to be assigned to a different device. This is restricted using the second restriction. To lower the size of the Design Space, we decided that the non-splittable layers that exist between splittable layers are grouped together, therefore those non-splittable layers will be assigned to the same device. This is done to also lower the amount of data communication required. We decided that these groups of non-splittable layers will be assigned to the device which is responsible to execute the first split of the previous splittable layer. Because of this, the concatenation operation after the horizontal split is also on the same device, thereby further lowering the communication required. Based on this restriction, it becomes clear that the advanced chromosome cannot encode the mapping presented in Figure 8b which uses the naive chromosome. This is because first of all, the first split of the Convolutional layer, $l_{1,1}$, is assigned to $device_3$. Based on the restriction described above, this means that the next non-splittable layers must be assigned to that same device. This is not the case with the example of Figure 8b. The same goes for the non-splittable layer l_6 after the Fully Connected layer l_5. Because the first part of the Fully Connected layer, $l_{5,1}$, is mapped to $device_4$, using our advanced chromosome layer l_6 must be assigned to $device_4$.
An adopted mapping is presented in Figure 9 which tweaks the example of Figure 8b to work with the advanced chromosome. Figure 9 shows that along with the first split of the Convolutional layer, $l_{1,1}$, the next non-splittable layers, l_2, l_3, l_4 are also assigned to $device_3$. In addition to this, along with the first split of the Fully Connected layer, $l_{5,1}$, the next non-splittable layer, l_6, is also assigned to $device_4$. The presented mapping does not violate any of the aforementioned restrictions of the advanced chromosome and still partitions the network using the Hybrid Partitioning strategy.
Based on the above discussion, we can define which mappings are possible and which are not when using this type of advanced chromosome. It is clear that this chromosome is very

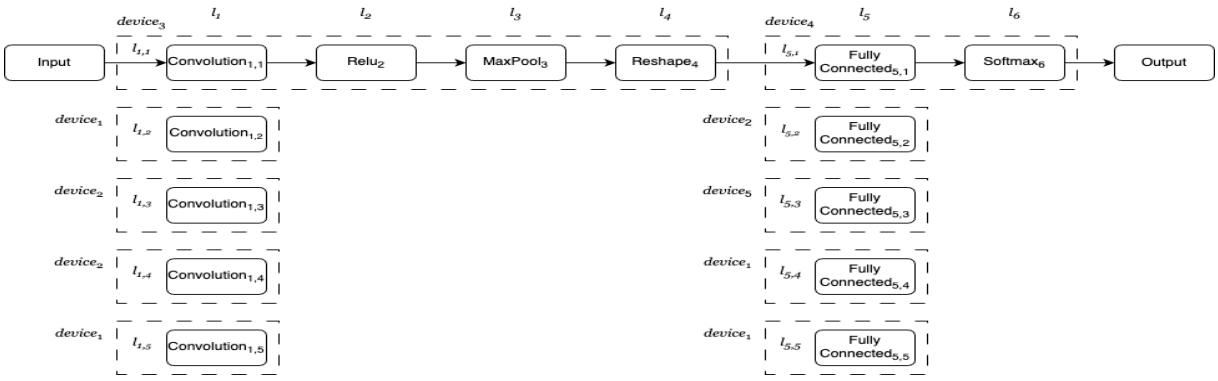


Figure 9: A valid Hybrid Partitioning of the CNN network of Figure 8a using the advanced chromosome.

restrictive in terms of the Vertical Partitioning, while the Horizontal Partitioning is restricted only by p, the number of maximum horizontal splits in a layer. Using this chromosome, it is not possible to encode mappings where non-splittable layers between two splittable layers are assigned to multiple devices. This chromosome groups such layers and therefore they are assigned to a single device. It is also not possible to assign such layers to a device which is not used by the splits of the previous splittable layer. An edge case can be defined where the first splittable layer comes after non-splittable ones. Such cases are still handled correctly by our implementation, which will assign those non-splittable layers to the first split of the first splittable layer.

The encoding of the advanced chromosome can now be defined. Since the partitioning of non-splittable layers is decided by the Horizontal Partitioning of the splittable layers, the chromosome does not use the original index of layers in the CNN network. In the example CNN of Figure 8a, the layer index numbers are between 1-6, because there are 6 layers. The advanced chromosome only needs to use the splittable layers, therefore the splittable layers are re-indexed in the correct order. This means that the index of l_1 in the example, the Convolutional layer, remains number 1, but the index of l_5, the Fully Connected layer, becomes number 2. This can be thought of as having a new layer list that only contains the horizontally splittable layers in their correct order. If more splittable layers existed in the network, those would also be in the index. To explain how the split points work, the horizontal splits are discarded for now. The chromosome has n positions, where n is the number of devices. Each position belongs to a device. Each position gets a number from the indices of splittable layers after the aforementioned re-indexing. An extra number is added to the possible indices, number 0, which is used when a device does not get a layer assigned. Figure 10 shows an example of the split point encoding that uses the index described above which contains only the splittable layers of the CNN network of Figure 8a. As shown, devices $1, 3, 5$ have the number 0 on their position, thus they do not participate in the mapping. Because $device_2$ has number 1 on its position, the first splittable layer, l_1, is assigned to it along with the non-splittable layers that follow, l_2, l_3, l_4. Because $device_4$ has 2 on its position, l_5, which is the second splittable layer is assigned to it, along with the next group of non-splittable layers, l_6. If this example was modified and $device_2$ also had 0 on its position, then that would mean that $device_4$ which has 2 on its position is assigned all splittable layers up-to and including the second splittable layer, and therefore it would also take the non-splittable layers l_2, l_3, l_4 which are between the two splittable and the non-splittable layer l_6 which is after the second splittable. Thus, a split

	$device_1$	$device_2$	$device_3$	$device_4$	$device_5$
Index	1	2	3	4	5
	0	1	0	2	0

Figure 10: An example of a split point encoding which uses only the re-indexed splittable layers of the CNN network of Figure 8a.

point is defined as a number in the index of the list of the splittable layers of the CNN, and shows that the device is responsible for the splittable layers between the last split point and the current split point, along with the non-splittable layers that exist after the current split point until the next split point is found.

Now that the concept of a split point is explained, we can modify the chromosome described to form the advanced chromosome which supports p horizontal splits. This chromosome has n positions for each split, and its length is given by Equation 4.6.

$$Length = p * n \tag{4.6}$$

Based on the above, the encoding scheme uses an array of length $p * n$, where p represents the number of splits and n represents the number of devices. Each split requires n positions to indicate the assignment of layers to the n available devices. This encoding can be thought of as a two-dimensional array with p rows and n columns, where each row represents a split and each column represents a device. The encoding uses the re-indexed splittable layers along with the number 0 to indicate that the device does not participate in a certain split. Furthermore, the definition of the split point is slightly adjusted. A split point denotes that a device is responsible for handling the current split of the horizontally splittable layers between the layer indicated by the previous split point up to and including the layer indicated by the current split point. If the current split is the first, then the device is also responsible for the non-splittable layers that exist after the layer indicated by the split point until the next split point in the current split.

Figure 11 shows the advanced chromosome encoding for the example mapping in Figure 9. Figure 9 shows that the first partition of the first splittable layer goes to $device_3$ and the first partition of the second splittable layer goes to $device_4$. Thus, in the first row of Figure 11, devices $1, 2, 5$ have 0 because they do not participate in the first split of any splittable layers. $device_3$ is assigned 1 which is the new index number of the first splittable layer and $device_4$ gets 2, the new index number for the second splittable layer. Because of this, layers $2, 3, 4$ are assigned to $device_3$ and layer 6 is assigned to $device_4$. The next rows in Figure 11 show the rest of the horizontal splits on the splittable layers. The second row, which concerns the second split, assigns 1 to $device_1$, thus $l_{1,2}$ is mapped to $device_1$ as shown in Figure 9. The same goes for $device_2$ which is assigned 2, meaning $l_{2,2}$ is assigned to $device_2$. The third row in Figure 11 concerns the 3rd split and is straightforward. Rows 4 and 5 of the figure are more interesting. In row 4, we see that $device_2$ is assigned number 1, and $device_1$ is assigned 2. Therefore, $l_{1,4}$ is assigned to $device_2$ and $l_{2,4}$ is assigned to $device_1$. Finally, row 5, only has a single device with an assigned number other than 0. $device_1$ is assigned 2, meaning that the fifth split of the splittable layers 1 and 2 are mapped to that device.

To demonstrate the efficiency of this advanced chromosome, AlexNet is considered as an example which has 24 layers, of which 5 are Convolutional and 3 are Fully Connected. Using

	Device1	Device2	Device3	Device4	Device5
Split1	0	0	1	2	0
Split2	1	2	0	0	0
Split3	0	1	0	0	2
Split4	2	1	0	0	0
Split5	2	0	0	0	0

Figure 11: The equivalent advanced chromosome encoding for the example in Figure 9.

the Hybrid Partitioning method, Convolutional and Fully Connected layers are allowed to be horizontally partitioned. Five devices are used to partition the CNN. Recall that the size of the naive chromosome's search space is 5^{56}. This is because each position in the chromosome can have 5 different values, equal to the number of devices. The total number of positions is 56: there are 8 splittable layers, meaning $8 * 5$ positions for these layers, the network has 24 layers, thus $24 + (8 * 5) - 8 = 56$. Considering the advanced chromosome, the size of the search space becomes 9^{25}. This is because there are 8 splittable layers and we also need the number 0, thus 9 possible values for every position. The number of positions is equal to $5 * 5$, which is the number of horizontal splits allowed multiplied by the number of devices. Thus, the search space size is 9^{25}, which is lower than 5^{56} by 15 orders of magnitude.
The development of this chromosome is one of the crucial parts of the work done in this research project because it lowers the Design Space complexity and allows the utilization of the developed Hybrid Partitioning strategy to be used with large state-of-the-art CNNs. Based on the characteristics and restrictions of the chromosome, the resulting mappings found by the GA should not only lower the data communication between devices but also potentially distribute evenly the workload among them.

4.3 Analytical Model

The fitness function of a GA plays a crucial role in a DSE method, particularly in this project, as it aims to optimize three distinct objectives. These objectives depend not only on the execution platforms but also on the specific implementation that is used. Since the AutoDice framework is utilized, the implementation for the CNN inference is done by the ncnn [26] package, which is an inference package optimized for inference on mobile platforms offering high performance. The resulting implementations generated by AutoDice are tailored to the specific platforms provided and encompass platform-specific optimizations. For instance, a convolution operation might leverage platform-specific memory alignment and intrinsic operations. Thus, capturing the performance characteristics of a mapping is a complex problem.
An analytical model is developed in this project in order to be used as the fitness function during the DSE with NSGA-II. Mappings are evaluated using the model which returns back the maximum expected energy consumption, maximum expected main memory usage and expected inference latency. In order to evaluate a mapping, the original graph of the CNN needs to be provided along with the available devices, the mapping and a coefficients' file. Based on this information, our analytical model can analyze the mapping and provide the

expected performance indicators.
Our approach taken in this research project to develop the analytical model begins with an elaborate tuning process. The developed model uses the distributed system on which the mappings will be deployed to perform a model tuning process resulting in a .json file with coefficients. The file includes parameters for all of the analytical model's parts which will be explained in the following subsections.

4.3.1 Inference Latency

The model's estimation of the CNN inference latency relies on the number of Multiply-Accumulate (MAC) operations. A MAC operation involves calculating the product between two numbers and adding the product to the accumulating variable. This choice is motivated by the fact that MAC operations are not only prevalent in CNN executions but also serve as a reliable performance indicator. The input CNN graph contains information about all the layers in the CNN including their characteristics and attributes. By extracting the information about a specific layer in the graph, the expected MAC operations can be calculated. For most layers in a CNN, the MAC operations are largely dependent on the input and output tensor shapes, meaning the model is able to get an approximate count for calculating the inference latency. For example, an implementation of a Fully Connected layer which executes a matrix multiplication, would require $(H_i^X * 1) + (H_i^Y * 1)$ MAC operations. Using the graph, the model is able to calculate the total amount of MAC operations that will take place on a specific device and therefore can approximate the inference latency.
In order to compute the inference latency, the MAC operations need to be converted to approximate execution time. One approach would be to divide the calculated MAC operations by the frequency of the executing device. However, this approach has two inherent issues. First of all, the model cannot precisely determine the amount of MAC operations that will take place because it is highly dependent on the implementation. The AutoDice framework includes optimized implementations based on a layer's characteristics, making the calculation of the actual number of MAC operations that would occur challenging. The second issue originates from the nature of modern processing units, either CPUs or GPUs. A unit's frequency does not only fluctuate depending on the instruction being executed, but also modern architectures include optimizations which allow for multiple instructions to be executed at once or for operations to be executed faster. Therefore, basing the inference latency solely on the frequency of the execution unit would yield sub-optimal results.
For this reason, our model uses calibrating coefficients that are generated through the tuning process. For the inference latency parameters, the CNN is run on a single device of the distributed system without any partitioning. If a heterogeneous system is provided, this process can be easily adapted to allow for different parameters per platform, which for this project is done to obtain different model parameters for CPU and GPU execution. Using the AutoDice framework, the code for the CNN is generated in benchmarking mode which executes the inference 24 times, 4 times warmup and 20 times for benchmarking. Using the per-layer execution time the tuning process creates a coefficient for every layer which will be used in the inference latency calculation. Equation 4.7 presents our approach to time calculation. The frequency of the execution unit is calibrated using a coefficient which is different for every layer and every execution unit that is used. In theory, this should eliminate the deviation in the amount of actual MAC operations and dynamic frequency adjustments. Furthermore, since the Convolution operation is the most important, for that layer there are two different coefficients. This is be-

cause the AutoDice system provides two different implementations for the Convolution based on the layer's characteristics, im2col and Winograd convolution. These differ significantly in the amount of MAC operations and therefore the model uses different coefficients and MAC calculations for each type of convolution.

$$time = \frac{MAC}{frequency * layer\_coefficient} \tag{4.7}$$

4.3.2 Communication Latency

To ensure accurate performance evaluation of the mapping, our analytical model takes into account the impact of data communication on overall performance. Based on the amount of data that is communicated, a delay is calculated using the data size and the communication medium's speed, as shown in Equation 4.8. The delay calculation is based on parameters depending on the distributed system's characteristics. Specifically, our model uses a different communication medium's speed to calculate the delay depending on whether the communicating processes are on the same device, on networked devices, or when a CPU process needs to communicate with a GPU process.

$$Comm\_latency = \frac{data\_size}{comm\_medium} \tag{4.8}$$

Although this might seem like a good approximation for the communication delay occurring when devices exchange data, there is a key piece missing. Because our partitioning strategy uses Horizontal partitioning with LOP, there might be cases where the devices incur a small synchronization delay. When the concatenation operation is assigned to a device which is also executing part of the partitioned layer, a possibility exists that the device has not finished its processing in time and therefore the other devices must wait before sending their part of the layer. The situation can also happen in reverse, where the concatenating node needs to wait for a device which has not finished in time. This is taken into account in the analytical model which uses the communication and inference delays in order to calculate the time which each device will finish processing their part. Based on the result of the calculation, a small penalty is added to the nodes if their result is not ready in time.
The resulting communication delay for each device and the inference latency are summed up to form the total inference latency for each device.

4.3.3 Memory Utilization

To determine the memory usage, our model utilizes the provided CNN model graph and associated information to calculate the precise amount of memory required by each device for storing weights and data associated with their mapped layers. In addition, during communication between devices, either for result concatenation or because the next layer is executed on the receiving device, the sender needs to store the information in a buffer until the receiving device is ready to accept it. Similarly, the receiver needs to allocate space for buffers to store the incoming data. Thus, whenever communication occurs, the model includes the memory usage of the sender and receiver, which is equal to the amount of data sent or received. Taking such estimated memory by the model and comparing it to the actual memory usage when the mapping is executed on the real system would show a very large accuracy error. This is because, in a real system, other factors also influence the memory usage. Firstly, the real implementation

requires the loading of libraries and other data which use a substantial amount of memory. Moreover, the real implementation might need to store temporary data during the calculations and therefore the real memory usage further deviates from the described estimation. Lastly, techniques are being used in the real implementation that allow devices to free up space by discarding data that are not further needed, for example weights of previous layers that have been used and not needed in the future. All of these factors make calculating the expected memory usage of a mapping challenging.
For this reason, during the tuning process, our model creates coefficients and biases for the memory usage calculation depending on the platform of execution. During tuning, the CNN is mapped on a single device except for a single layer. For each different platform, a Convolutional layer, a Fully Connected layer or a ReLU layer are run independently on a different device. This allows the model to capture a more accurate representation of the memory usage in the coefficients because these three layers differ significantly from each other. The Convolutional layer's weights are substantially different from the FC layer's weights and the ReLU layer is used as a representation of the other types of layers which do not use any weights. Furthermore, to create the coefficients, the model also makes use of the memory usage of the device that executes the rest of the layers. Equation 4.9 presents our memory calculation. During tuning, for each platform, the three different mappings are created and executed. The execution is once again done using the benchmarking mode for 24 inference sessions. The first four sessions are discarded and the memory used for the rest of the sessions is averaged. Thus, the model can solve a system of equations using Equation 4.9 where one equation is for the device running the single layer and the second equation is for the device executing the rest of the layers. This calculation results in the coefficient and bias which are averaged for the three mappings.

$$memory = expected\_memory * platform\_coefficient + platform\_bias \tag{4.9}$$

Using the coefficient and bias along with Equation 4.9, our model should in theory approximate the real memory for each device as closely as possible.

4.3.4 Energy Usage

Approximating energy usage accurately is a challenging task due to its dependence on various factors, including the frequency of the executing device and the specific instructions being executed. Furthermore, in this case, it is also influenced by the different implementations which can be generated through the AutoDice framework, as well as the specific hybrid partitioning approach we utilize. Our developed model uses the total inference latency expected for every device, which is the amount of time a device spends executing its part of the mapping, in order to approximate its energy usage. The approximated energy usage is the result of the product of the inference latency of a device and a platform energy coefficient as shown in Equation 4.10. The energy coefficient is created for every platform available in the distributed system during the tuning process. To create the coefficient, the CNN network is executed as a whole on a single device for every platform in benchmarking mode and the energy consumption is measured. Then, we use our model along with the previously calibrated coefficients for inference latency, communication latency and memory usage, to predict the total execution time for each device. Using the results, a platform energy coefficient is created for every platform by solving Equation 4.10. This coefficient is used to approximate the real energy consumption based on the execution time of each device.

$$energy = expected\_inference\_latency * platform\_coefficient \tag{4.10}$$

5 Experimental Evaluation

This section describes the experimental setup and presents the results obtained from the experiments conducted to evaluate the effectiveness of the proposed hybrid partitioning strategy, the two developed chromosomes used in DSE, and the analytical model.

Table 2: CNNs used for experiments with their characteristics and performance when not partitioned. Column 4 shows the storage size of the network (weights, biases) while column 6 shows the peak main memory usage during inference.

Network	Num. Layers	Num. Params	Size (MB)	CPU Exec.		
				En. (J)	Mem. (MB)	Perf. (ms)
AlexNet	24	60.8 million	238	0.352	152.746	28.3
VGG-16	47	138 million	528	1.959	446.852	122.46
DenseNet-121	910	8 million	32	1.026	95.969	138.68

5.1 Experimental Setup

The experiments are designed to validate the objectives of this thesis, i.e. to answer the research questions posed in Section 1. First of all, they evaluate the effectiveness of the Hybrid Partitioning and whether it shows potential improvements over the Vertical Partitioning strategy. In addition, they asses the efficiency and the DSE performance improvements of the proposed advanced chromosome in terms of convergence when compared to the naive chromosome. Furthermore, the experiments set out to show whether using GPUs in addition to CPUs offers potential improvements in the performance of the Hybrid Partitioning in terms of energy consumption per device, main memory usage per device and CNN inference latency. Lastly, they aim to show the accuracy of the analytical model and whether it can be used for DSE to evaluate and compare the performance of mappings in real-world scenarios.
All the conducted experiments have been implemented in Python version 3.9.0 in combination with the package pymoo [27] version 0.5.0 for the NSGA-II implementation and onnx [16] version 1.13.0 for managing the CNNs.
The experiments also include real-world tests where mappings have been implemented using AutoDice on a real distributed system to determine their real-world performance characteristics, energy consumption per device, main memory usage per device and inference latency, and compare them to the predicted characteristics given by the analytical model. When a mapping is run on the real system, it uses the benchmarking mode provided by AutoDice which executes the inference 4 times as a warmup and an extra 20 times for measuring the different performance indicators. The utilized real distributed system is comprised of 8 NVIDIA Jetson Xavier NX devices [28]. Each device is equipped with a 6-core NVIDIA Carmel ARMv8 CPU, an NVIDIA Volta GPU with 384 cores, and 8GB of RAM. For the experiments in this project, the CPU and GPU of each device are considered as 2 distinct devices, therefore we have 16 devices available in total.

Also, all of the experiments have been conducted using three different CNNs, namely AlexNet [8], VGG-16 [29] and DenseNet [30]. The selection of these CNNs is motivated by the importance of experimenting with CNN models of varying characteristics, not only for the hybrid partitioning strategy but also for the chromosomes in a wide range of situations. The characteristics of these CNNs are presented in Table 2. Rows 3, 4 and 5 correspond to the three networks used. Column 1 shows the network name, column 2 the number of layers present in the network, column 3 the number of parameters used in the network and column 4 the storage size of the network (storage needed for weights, biases, etc.) in Megabytes. The last 3 columns show the performance characteristics of the CNNs when executed on a single device, un-partitioned, with the implementation created by the AutoDice framework exploiting 6 CPU cores. Column 5 shows the energy consumed by the device during inference in Joules. Column 6 shows the peak main memory usage in Megabytes consumed by the device during inference. Lastly, column 7 shows the total inference latency measured in milliseconds. From the table, it is clear that the storage size of a CNN network is different than the peak main memory consumption. This is because the implementation used optimizes the main memory usage by discarding data that are not further needed and by loading data in stages. The storage size of AlexNet is 238 Megabytes, while the peak main memory usage is lower, i.e. 152.746 Megabytes. The same is true about VGG. When examining DenseNet, it is apparent that the peak main memory usage is higher than the storage size. This is because the implementation must also load other libraries and may take space for temporary data during calculation. Therefore, since the storage size of the network is so small in the case of DenseNet, the peak main memory usage becomes higher than the storage size.
As we can see, AlexNet, despite having the smallest number of layers is the second largest network in terms of storage size, while DenseNet has the smallest number of parameters but the largest number of layers. Not only do the three networks have a difference in storage size and amount of layers, but they also have different topologies. In DenseNet for instance, the layers are densely connected together, where each layer is connected to other layers that come after it in a feed-forward manner, unlike AlexNet and VGG where a layer is connected only to its subsequent layer. Furthermore, the CNN networks also employ different types of layers. In DenseNet, Batch Normalization layers are used whereas the other two CNN networks do not make use of such layers. The networks also differ in the amount of Fully Connected layers in their topologies, with AlexNet and VGG having three Fully Connected layers at the end, while on the other hand, DenseNet has none.
The DSE has been conducted using the developed analytical model as the fitness function for NSGA-II for 100 generations and a population of 1000 individuals, thereby allowing the search algorithm to explore a wide range of solutions. The naive chromosome does not require any inputs or parameters to be set. On the other hand, the advanced chromosome requires the number of possible horizontal splits for splittable layers as input. Therefore, the advanced chromosome has been used with a range of possible splits between 2 and 6. When the DSE finishes 100 generations, from the set of Pareto-Optimal points found, the points with the lowest energy per device, the lowest memory usage per device and the one with the lowest inference latency are selected and implemented using AutoDice on the real distributed system.

5.2 Experimental Results

The experimental results for each CNN model used in the experiments are presented in the following subsections.

5.2.1 AlexNet Results

First of all, the results for AlexNet are presented. Figure 12 contains three plots showing the DSE convergence when using the analytical model as the fitness function, and comparing the two chromosomes when using the Hybrid Partitioning with horizontally partitioned Convolutional and Fully Connected layers. In this experiment, only the CPUs of the devices are used. The plots compare the naive chromosome against the advanced chromosome with 2, 4 and 6 horizontal splits. The x-axis of all three plots shows the number of generations the DSE run. The first plot compares the convergence of the energy consumption per device during DSE and the y-axis shows the energy measured in Joules. The plot shows that all three advanced chromosomes lead to faster DSE convergence than the naive chromosome. The advanced chromosome with 2 splits exhibits faster DSE convergence than all others and finds the mapping with the lowest energy consumption per device in 100 generations. The DSE with the advanced chromosome with 6 splits manages to find a mapping that achieves a lower energy consumption per device in the total number of generations compared to the mapping found by the DSE using the advanced chromosome with 4 splits. The naive chromosome on the other hand achieves the highest energy consumption per device as it converges much slower than the other three. The second plot compares the peak main memory usage per device convergence during DSE and its y-axis shows the peak main memory usage in Megabytes. As can be seen, again the naive chromosome exhibits the slowest convergence. The advanced chromosome using 2 splits shows a flat line, meaning that its best achieved value has been found early in the generations and has not been improved after 100 generations. The DSE using the advanced chromosome with 4 splits finds the mapping with the lowest peak main memory usage while the DSE utilizing the 6 splits chromosome finds a mapping that achieves the second lowest main memory usage. This is expected as these two chromosomes offer much more freedom when compared to the chromosome with 2 splits, and therefore can divide the memory usage more evenly amongst the devices. Lastly, the third plot compares the inference latency convergence during DSE and its y-axis shows the inference latency in seconds. As shown in the plot, the DSE using the advanced chromosome with 2 horizontal splits finds the mapping with the lowest inference latency after 100 generations and also converges faster than the DSE runs with the other chromosomes. As expected, the DSE runs using the advanced chromosomes with 4 splits and 6 splits find mappings that achieve the second and third faster inference latency in 100 generations respectively. The naive chromosome, again exhibits much slower convergence than the other chromosomes. Based on the above results, it is clear that the advanced chromosome with 2 splits can find better mappings in terms of energy consumption per device and inference latency when compared to the other three chromosomes, while the advanced chromosome with 4 splits is better for finding mappings in terms of memory usage per device in such restricted DSE run time in the case of the AlexNet CNN when using only CPUs.

The second experiment is presented in Figure 13, which contains the same three plots described above, only this time the mappings explored during the DSE process use both CPUs and GPUs. This is done for two reasons. First to compare the chromosomes when there is higher complexity in the design space because now the devices are essentially doubled. The second reason is to check whether using both CPUs and GPUs gives benefits to the Hybrid Partitioning for AlexNet. The first plot presents the energy consumption per device, with the y-axis showing the energy consumption in Joules. As with the previous experiment, the DSE with the advanced chromosome converges faster than the DSE using the naive one. Again the

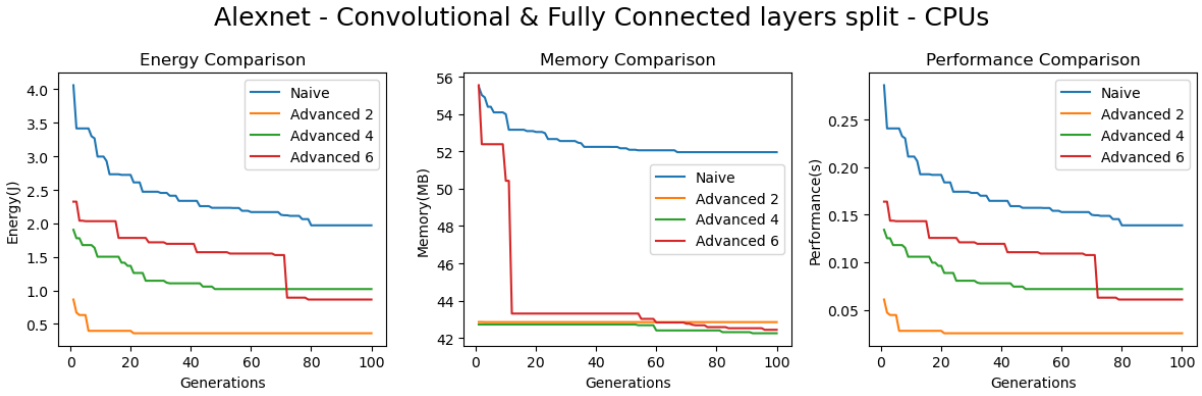


Figure 12: AlexNet DSE convergence for hybrid partitioning when splitting Convolutional and Fully Connected layers and using only CPUs.

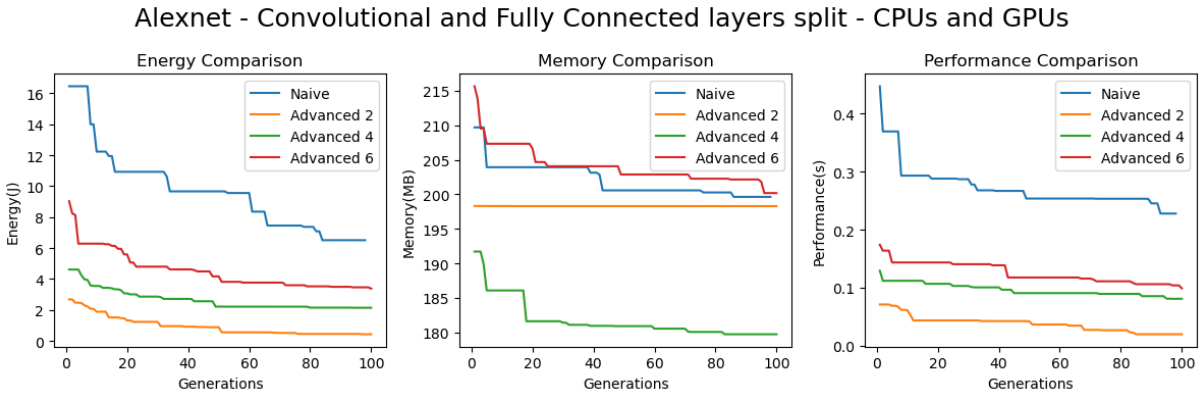


Figure 13: AlexNet DSE convergence for hybrid partitioning when splitting Convolutional and Fully Connected layers and utilizing CPUs and GPUs.

DSE using the advanced chromosome with 2 horizontal splits finds the mapping with the lowest energy consumption per device after 100 generations. In this experiment, the DSE utilizing the advanced chromosome with 6 splits does not find a mapping that achieves a better result when compared to the mappings found with the DSE using 4 splits advanced chromosome. In terms of energy consumption per device, it is evident that none of the chromosomes tested can achieve better energy consumption per device when using both CPUs and GPUs when compared to using only CPUs for AlexNet. The second plot shows the peak main memory usage per device in Megabytes. This time, the DSE using the advanced chromosome with 4 splits converges faster than the DSE using the other chromosomes. The advanced chromosome with 2 splits shows a flat line. On the other hand, the naive chromosome manages to outperform the advanced chromosome with 6 splits in terms of memory usage during the DSE process in 100 generations. The plot shows that in 100 generations the memory usage per device when using both CPUs and GPUs cannot be lowered to the same point when compared to using only CPUs. The inference latency is presented in the third plot and measured in seconds on the y-axis. As expected, the DSE runs using the advanced chromosomes converge faster than the DSE using the naive chromosome. After 100 generations the advanced chromosome with 2 splits finds the mapping with the lowest inference latency, followed by the mappings found from the DSE of the 4 and 6 splits chromosomes. In this case, the advanced chromosome with 2 splits manages to find a mapping after 100 generations which achieves a lower inference latency when compared to only using CPUs. Thus, using both CPUs and GPUs for AlexNet when performing the DSE in 100 generations provides a benefit in inference latency.

The third experiment is shown in Figure 14. This experiment compares the Vertical Partitioning strategy using the Split Point encoding [18] against the Hybrid Partitioning using only CPUs. For the Hybrid Partitioning, the advanced chromosome with 2 splits has been selected and both Convolutional and Fully Connected layers are horizontally split. The plots again show the DSE convergence for 100 generations, comparing the energy consumption per device in Joules, peak main memory usage in Megabytes and inference latency in seconds, shown on the y-axis of the plots. As can be seen from the first plot, which shows the energy consumption convergence, the Hybrid Partitioning does not manage to outperform the Vertical Partitioning when executing DSE for 100 generations, but it comes close to it. The DSE using the Vertical Partitioning on the other hand converges much faster than the DSE using the Hybrid Partitioning, thereby finding the best mapping it can in 100 generations very early in the DSE process. On the other hand, comparing the peak main memory shown in the second graph, the DSE using the Hybrid Partitioning outperforms the DSE utilizing the Vertical Partitioning by a big margin. Lastly, for the inference latency, again the DSE with the Vertical Partitioning outperforms the DSE using the Hybrid Partitioning by a small margin in the 100 generations. Therefore, for AlexNet, the Hybrid Partitioning can achieve lower peak main memory usage per device in 100 generations of DSE when compared to the Vertical Partitioning which in turn is better for energy consumption per device and inference latency.

The fourth experiment conducted for AlexNet is a comparison between the real values measured on the real distributed system and the corresponding predicted values provided by the analytical model. For every chromosome tested and using CPUs only in the first experiment, the best design points found in terms of energy consumption per device, memory usage and inference latency are selected from the set of Pareto-Optimal points. These design points have been implemented using AutoDice and tested on the real distributed system. The results are shown in Table 3. The table's rows show the results for the best points in terms of energy consumption, memory usage and inference latency for every chromosome utilized during the DSE. The parts

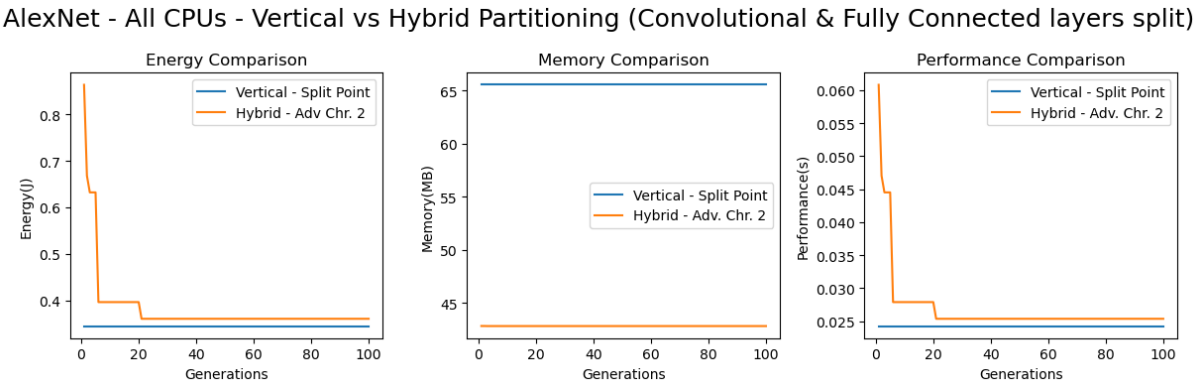


Figure 14: AlexNet DSE convergence comparing the Vertical and Hybrid Partitioning. The Vertical Partitioning uses the Split Point encoding [18]. The Hybrid Partitioning uses the advanced chromosome with 2 horizontal splits for Convolutional and Fully Connected layers. Only CPUs are used.

Table 3: AlexNet results for the Hybrid Partitioning using CPU only. Best points found for each objective from the set of Pareto-Optimal points with their real and model evaluations.

	Chromosome	**Energy (J)**		**Memory (MB)**		**Latency (ms)**	
		Real	Predicted	Real	Predicted	Real	Predicted
Best Energy Point Found	Naive	1.337	1.971	46.027	55.424	83.7	138.758
	Advanced - 2	**1.102**	0.360	97.086	45.616	30.24	25.382
	Advanced - 4	1.126	1.020	74.945	45.559	26.63	71.812
	Advanced - 6	1.178	0.863	41.836	58.263	30.15	60.804
Best Memory Point Found	Naive	1.571	3.316	44.211	51.969	103.82	233.537
	Advanced - 2	1.113	0.668	72.402	42.842	34.78	47.021
	Advanced - 4	1.405	3.146	67.078	42.243	71.69	221.560
	Advanced - 6	1.394	3.544	70.723	42.430	83.08	249.575
Best Latency Point Found	Naive	1.355	1.971	43.801	55.424	85.54	138.758
	Advanced - 2	1.127	0.360	94.656	45.616	30.68	25.382
	Advanced - 4	1.111	1.020	74.996	45.559	**26.41**	71.812
	Advanced - 6	1.201	0.863	**41.789**	58.263	29.66	60.804

of the table with a stronger border indicate the real and predicted values of the objective for which the solution achieved the best score among all the Pareto-Optimal points and thus was selected. This means that three parts of the table have a stronger border, one for each objective. One point out of the set of all Pareto-Optimal points achieves the lowest energy consumption per device, one point achieves the lowest peak main memory usage per device and one point achieves the lowest inference latency. Thus, after the DSE process with a chromosome, the points are selected from the set of Pareto-Optimal points found and their real values measured on the distributed system and corresponding predicted values from the analytical model in terms of their respective best objective are indicated in the parts of the table with the thicker border. The three highlighted values in the table are the best value for every objective. To better visualize the results, three more graphs have been plotted, that use the results found in the table. Figure 15 compares the best points found for energy consumption per device, showing

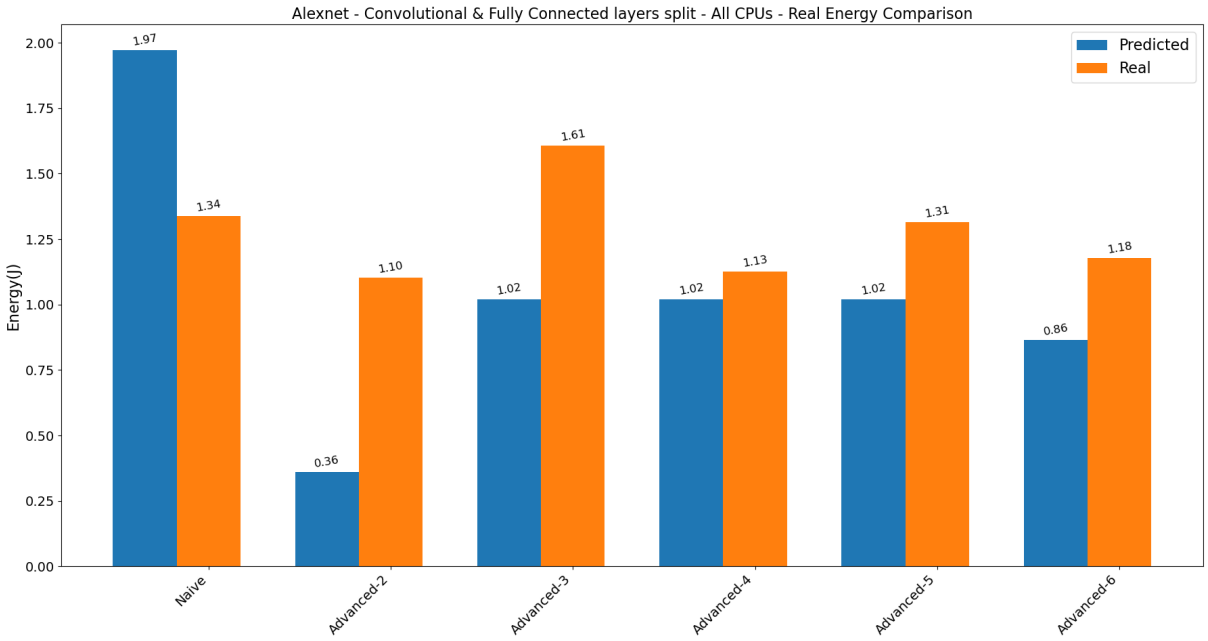


Figure 15: AlexNet comparison of predicted and real results for maximum energy usage per device when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

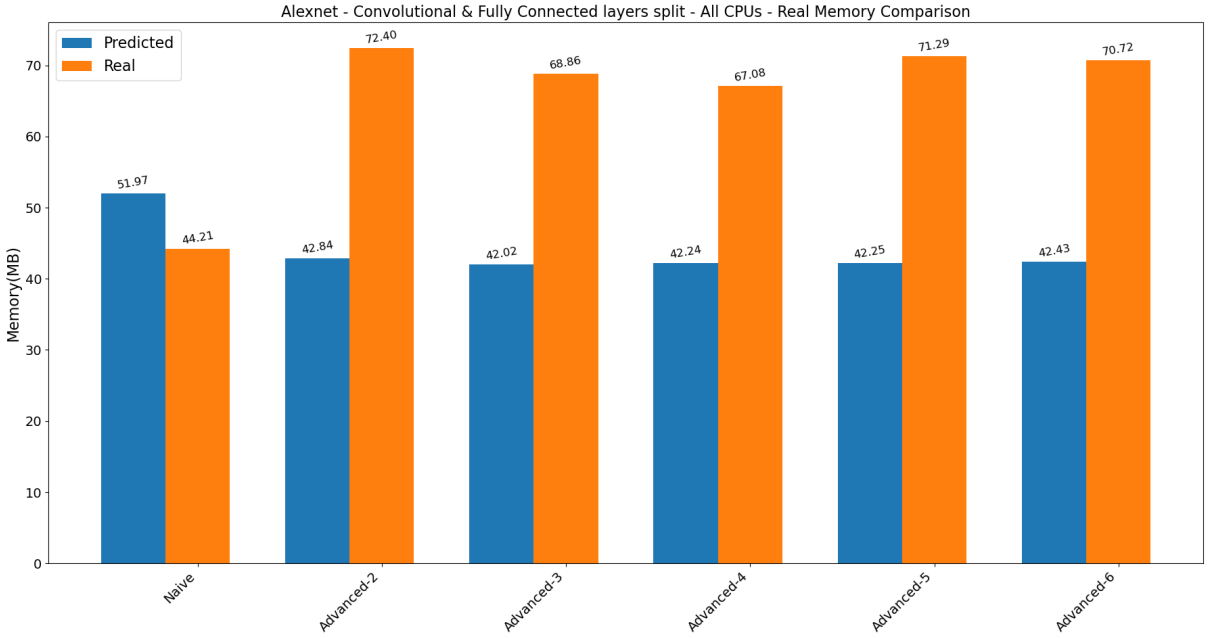


Figure 16: AlexNet comparison of predicted and real results for peak memory usage per device when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

their real and predicted energy values in Joules. Figure 16 shows the real and predicted memory usage per device in Megabytes of the best points found for main memory usage. Finally, Figure 17 shows the real and predicted inference latency in milliseconds of the best points found in terms of latency. The x-axis of all three figures denotes the used chromosome used to find each

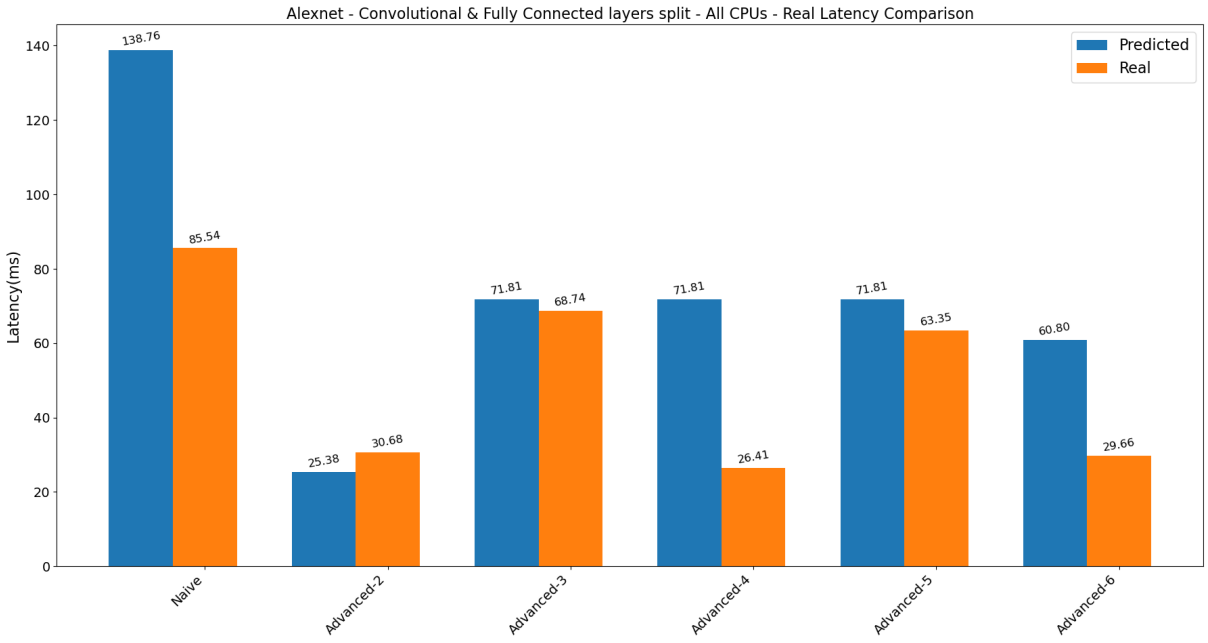


Figure 17: AlexNet comparison of predicted and real results for inference latency when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

point. As shown in Figure 15, the best point for the energy consumption per device found using the advanced chromosome with 2 splits outperforms all other points found using the other chromosomes in terms of energy per device in the real scenario, which was expected as it was also shown in Figure 12. The figure also shows that the model miss-predicted the performance of some solutions, namely for the points found by the advanced chromosomes with 3 and 5 splits, because their real energy consumption per device is higher in the real scenario than the value of the design point found with the 2 splits chromosome while the analytical model indicated that their value should be equal to that. In addition, the solution found by the advanced chromosome with 6 splits was expected to perform better than the solution found by the advanced chromosome with 4 splits, which is not the case in the real scenario. Figure 16 shows that the model miss-predicts the memory usage per device in every case. As shown in the figure, all points found by the advanced chromosomes were expected to deliver lower memory usage than the naive chromosome, but this is not the case in reality. Lastly, Figure 17 which presents the inference latency comparison, shows that the model closely predicts the performance of the solutions found by the advanced chromosomes with 2, 3 and 5 splits, but miss-predicted in all other cases. The issue is that while the advanced chromosome with 6 splits is expected by the model to perform better than the solution found by the chromosome with 4 splits, this is not the case. However, all of the solutions found by using the advanced chromosomes achieve a lower real inference latency than the naive chromosome. Comparing the real results of this experiment with the numbers shown in the last 3 columns of Table 2 shows that in 100 generations of DSE, the advanced chromosome is able to lower the inference latency when compared to the un-partitioned solution. Therefore, in terms of inference latency, the model works well to guide the DSE process towards lower latency because the DSE found a mapping with lower real inference latency than the un-partitioned real inference latency of the CNN. In terms of memory usage, it is clear that the analytical model does not work because it miss-predicted all cases. Finally, for the energy consumption per device, the model helps for

finding mappings with lower energy consumption per device, but in 100 generations, the DSE process was not able to find a partitioning that is better than the un-partitioned performance. Based on the results of this experiment, we can conclude that for AlexNet the analytical model works well to guide the DSE process towards design points with lower energy consumption per device and inference latency but does not work for finding design points with reduced memory usage per device.
Finally, Figure 18 presents the topology of AlexNet next to Table 4 which shows the mapping of the best found solution for AlexNet after the DSE process in terms of inference latency, which is achieved by the advanced chromosome using 4 horizontal splits. The table shows the layers in order and adds the concatenation operations wherever needed. The result of the Hybrid Partitioning is shown along with the original output sizes and modified output sizes of the layers. As can be seen from the table, the found solution splits the first, third and fifth Convolutional layers and all Fully Connected layers. All partitioned layers are divided into two splits rather than four. This may mean that the chromosome could not find a solution in 100 generations where four splits performed better than two.

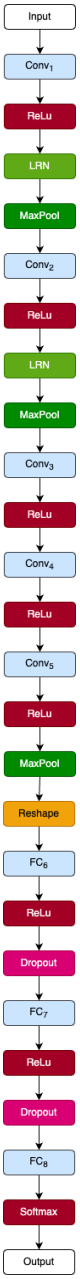


Figure 18: AlexNet non-partitioned topology.

Table 4: AlexNet best mapping found using the Advanced chromosome with 4 horizontal splits in 100 generations in terms of inference latency. Table 3 in row 13 shows the performance numbers of this mapping.

Layer_num	Layer	Original Output Size	Split Output Size	Devices
l_1	$Conv_1$	$Y_1^{[54x54x96]}$	$Y_{1,1}^{[54x54x48]}$ $Y_{1,2}^{[54x54x48]}$	dev_8 dev_2
l_2	Concat	-	$Y_2^{[54x54x96]}$	dev_8
l_3	ReLu	$Y_3^{[54x54x96]}$	-	dev_8
l_4	LRN	$Y_4^{[54x54x96]}$	-	dev_8
l_5	MaxPool	$Y_5^{[54x54x96]}$	-	dev_8
l_6	$Conv_2$	$Y_6^{[26x26x256]}$	-	dev_7
l_7	ReLu	$Y_7^{[26x26x256]}$	-	dev_7
l_8	LRN	$Y_8^{[26x26x256]}$	-	dev_7
l_9	MaxPool	$Y_9^{[26x26x256]}$	-	dev_7
l_{10}	$Conv_3$	$Y_{10}^{[12x12x384]}$	$Y_{10,1}^{[12x12x192]}$ $Y_{10,2}^{[12x12x192]}$	dev_1 dev_3
l_{11}	Concat	-	$Y_{11}^{[12x12x384}$	dev_1
$l12$	ReLu	$Y_{11}^{[12x12x384}$	-	dev_1
l_{13}	$Conv_4$	$Y_{13}^{[12x12x384]}$	-	dev_5
l_{14}	ReLu	$Y_{14}^{[12x12x384]}$	-	dev_5
l_{15}	$Conv_5$	$Y_{15}^{[12x12x256]}$	$Y_{15,1}^{[12x12x128]}$ $Y_{15,2}^{[12x12x128]}$	dev_5 dev_6
l_{16}	Concat	-	$Y_{16}^{[12x12x256]}$	dev_5
l_{17}	ReLu	$Y_{17}^{[12x12x256]}$	-	dev_5
l_{18}	MaxPool	$Y_{18}^{[12x12x256]}$	-	dev_5
l_{19}	Reshape	$Y_{19}^{[12x12x256]}$	-	dev_5
l_{20}	FC_6	$Y_{20}^{[4096x1]}$	$Y_{20,1}^{[2048x1]}$ $Y_{20,2}^{[2048x1]}$	dev_6 dev_4
l_{21}	Concat	-	$Y_{21}^{[4096x1]}$	dev_6
l_{22}	ReLu	$Y_{22}^{[4096x1]}$	-	dev_6
l_{23}	Dropout	$Y_{23}^{[4096x1]}$	-	dev_6
l_{24}	FC_7	$Y_{24}^{[4096x1]}$	$Y_{24,1}^{[2048x1]}$ $Y_{24,2}^{[2048x1]}$	dev_6 dev_4
l_{25}	Concat	-	$Y_{25}^{[4096x1]}$	dev_6
l_{26}	ReLu	$Y_{26}^{[4096x1]}$	-	dev_6
l_{27}	Dropout	$Y_{27}^{[4096x1]}$	-	dev_6
l_{28}	FC_8	$Y_{28}^{[1000x1]}$	$Y_{28,1}^{[500x1]}$ $Y_{28,2}^{[500x1]}$	dev_6 dev_4
l_{29}	Concat	-	$Y_{29}^{[1000x1]}$	dev_6
l_{30}	Softmax	$Y_{30}^{[1000x1]}$	-	dev_6

5.2.2 VGG Results

The next CNN model which has been included in the experiments is the VGG-16 CNN model. The same experiments executed with AlexNet have also been executed with VGG-16. The results of the first experiment are presented in Figure 19 which shows the comparison of the DSE convergence of the chromosomes, comparing the energy consumption per device, the peak main memory usage per device and inference latency. In this experiment, both Convolutional and Fully Connected layers are horizontally partitioned and only CPUs are used. The first plot shows the comparison between the DSE convergence of the chromosomes in terms of energy usage per device. On the x-axis, the number of generations that the DSE run is shown while on the y-axis the energy is shown, measured in Joules. From the plot, it is evident that the DSE with the advanced chromosome converges faster than the DSE using the naive chromosome in terms of energy usage per device. The design point found after 100 generations by the DSE with the advanced chromosome using 2 horizontal splits outperforms all other design points found by the DSE runs using the other chromosomes and is followed by the design point found by the DSE using the chromosome with 4 horizontal splits. The second plot compares the maximum main memory usage per device, shown on the y-axis and measured in Megabytes. As expected, the DSE using the advanced chromosome converges faster than the DSE using the naive chromosome. The advanced chromosomes follow a similar trend, and after 100 generations the DSE using the chromosome with 2 horizontal splits finds the mapping with the lowest memory usage per device. The mappings of the DSE runs utilizing the other two advanced chromosomes follow closely in this particular objective. Finally, the third plot compares the inference latency convergence during the DSE. The chromosomes follow a similar trend to the energy comparison. In this case, the DSE using the advanced chromosome with 2 horizontal converges faster, finding the mapping with the lowest inference latency after 100 generations. The results of this experiment are as expected, meaning that the advanced chromosome exhibits faster convergence than the naive chromosome in all cases. In addition, due to the larger size of the VGG model, the chromosomes converge in order, meaning that the DSE using the advanced chromosome with 2 horizontal splits converges faster, then the DSE using the chromosome with 4 splits and the third is the DSE with the chromosome with 6 splits. Due to the large size of the design space, the DSE using the naive chromosome converges slower than even the DSE using the advanced chromosome with 6 splits, which allows more freedom when compared to the other two advanced chromosomes.

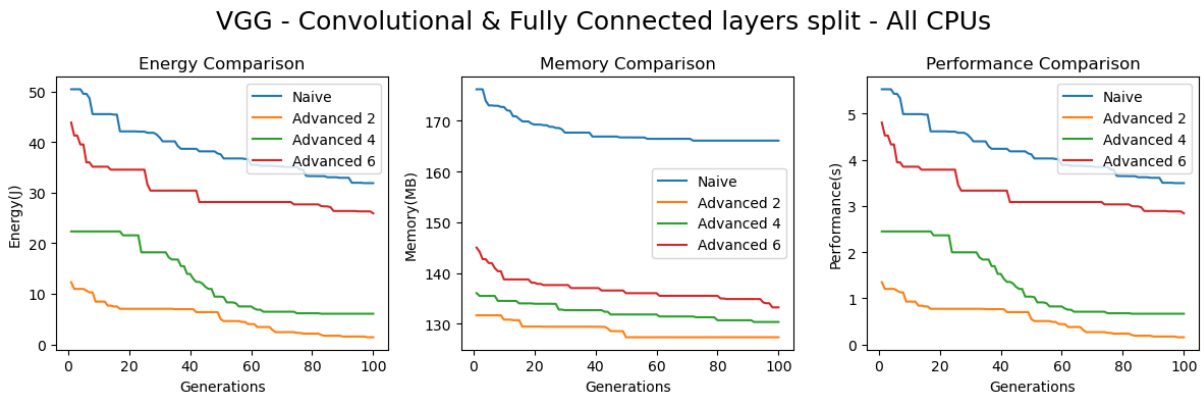


Figure 19: VGG DSE convergence for Hybrid Partitioning when splitting Convolutional and Fully Connected layers and using only CPUs.

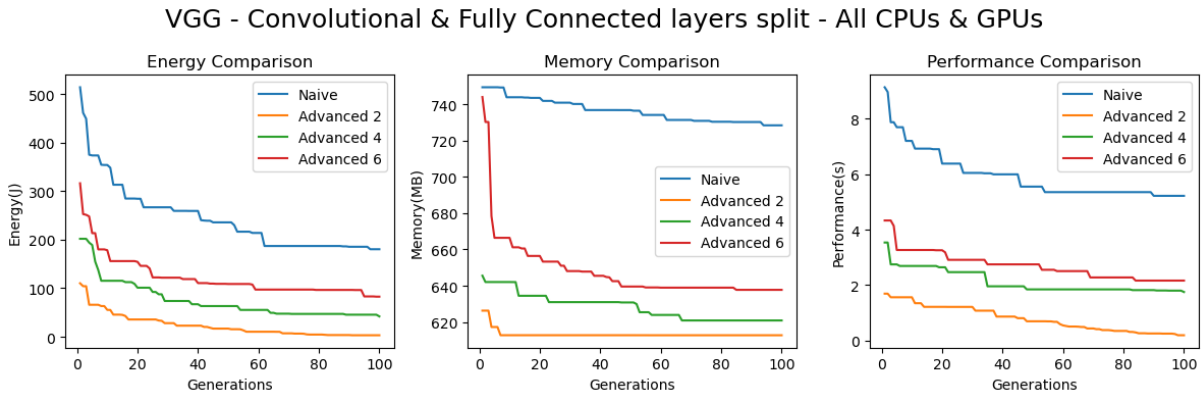


Figure 20: VGG DSE convergence for Hybrid Partitioning when splitting Convolutional and Fully Connected layers and utilizing CPUs and GPUs.

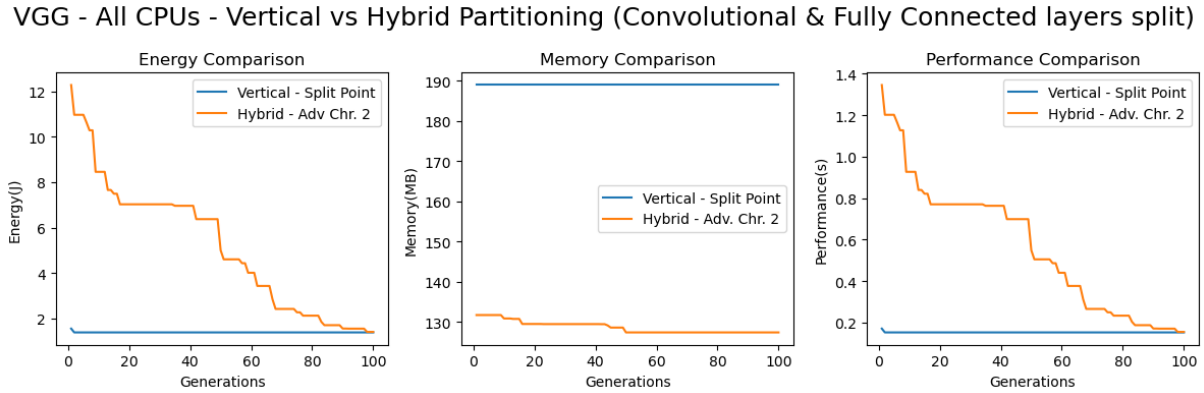


Figure 21: VGG DSE convergence comparing the Vertical and Hybrid Partitioning. The Vertical Partitioning uses the Split Point encoding [18]. The Hybrid Partitioning uses the Advanced Chromosome with 2 horizontal splits for Convolutional and Fully Connected layers. Only CPUs are used.

The results of the second experiment are presented in Figure 20. This experiment compares the DSE convergence in each objective. The difference from the first experiment is that now both CPUs and GPUs are used, meaning that the size of the design space becomes larger because the amount of available devices is doubled. The first plot compares the energy consumption per device and it is clear that once again the DSE runs using the advanced chromosomes converge faster than the DSE using the naive chromosome. The same is true about the peak main memory usage convergence shown in the second plot and the inference latency convergence in the third plot. In this experiment again, the chromosomes converge in order, with the first being the DSE using the advanced chromosome with 2 splits, then the DSE with 4 horizontal splits and then the DSE with 6 splits and last the DSE using the naive chromosome. In all cases, after 100 generations, the DSE using the advanced chromosome with 2 horizontal splits finds the mapping with the lowest value in each objective. This is because now the size of the design space is even larger than before and this chromosome is able to handle the large complexity well, thus the DSE converges faster than the other chromosomes. Comparing the results shown in Figure 20 with Figure 19, we can see whether using CPUs and GPUs offers benefits when compared to only CPUs. In terms of energy consumption per device and peak main memory usage per device, using only CPUs offers much better results after 100 generations. On the

other hand, in terms of inference latency, the mappings using both CPUs and GPUs are able to produce a close result after 100 generations when compared to using only CPUs, but still fall behind.

The experiment comparing the Vertical Partitioning and the Hybrid Partitioning when using the VGG network is next. In this experiment, the convergence of the two partitioning strategies is compared in 100 generations using the Split Point [18] encoding for the Vertical Partitioning and the advanced chromosome with 2 horizontal splits for the Hybrid Partitioning. The chromosome for the Hybrid Partitioning allows for horizontal partitioning of Convolutional and Fully Connected layers. The chromosome with 2 horizontal splits is selected because it found the best mappings after 100 generations in the first experiment. As with the first experiment, only CPUs are used in this experiment. Figure 21 presents the plots showing the convergence of the two strategies in the three objectives, energy consumption per device, peak main memory usage per device and inference latency. The first plot shows the energy consumption per device, with the energy shown on the y-axis and measured in Joules. From the plot, we can see that the DSE utilizing the Vertical Partitioning converges faster than the DSE utilizing the Hybrid Partitioning but after 100 generations the DSE with the Hybrid Partitioning manages to find a mapping with energy consumption close to the lowest energy consumption value found with a mapping of the Vertical Partitioning Strategy. The memory comparison is shown in the second plot, measured on the y-axis in Megabytes. The DSE using the Hybrid Partitioning in this case converges faster and finds a mapping with a lower peak main memory usage per device than the mappings found by the DSE with the Vertical Partitioning after 100 generations by a large margin. Lastly, the results for the inference latency are shown in the third plot, with the y-axis showing the inference latency measured in seconds. Here the trend is similar to the one in the energy comparison, with the DSE using the Vertical Partitioning converging faster and the DSE with the Hybrid Partitioning finding a mapping with an inference latency close to the lowest achieved inference latency value found by a mapping of the Vertical Partitioning DSE. Based on these results, the Hybrid Partitioning is shown to have good potential when used with the VGG network. After 100 generations it manages to get a lower peak main memory usage per device and it also manages to come close in terms of energy consumption per device and inference latency when compared to the Vertical Partitioning.

In the last experiment, the best found design points during the first experiment, in terms of energy consumption per device, peak main memory usage per device and inference latency, are selected from the set of Pareto-Optimal points, implemented using AutoDice and tested on the real distributed system. Recall that the first experiment uses only CPUs and allows horizontal splits to Convolutional and Fully Connected layers. These points were tested on the real distributed system and Table 5 shows the comparison between the real results and the corresponding predicted values by the analytical model. The rows of the table show the performance characteristics of the best found design points for each objective from the set of Pareto-Optimal points of each chromosome used. The energy is measured in Joules in the third and fourth columns, for the real and predicted values respectively. The fifth and sixth columns show the real and predicted values for the peak main memory usage in Megabytes. The last two columns show the real and predicted values for the inference latency in milliseconds. Figure 22 compares the real and predicted energy of the best found points in terms of energy consumption per device with each chromosome, with the y-axis showing the energy measured in Joules. On the x-axis, the chromosome used to find each point is shown. As shown in the figure, the mappings found using the advanced chromosomes with 3 and 4 horizontal splits achieve the first and second lowest real energy consumption per device. The analytical model

failed correctly predict the energy consumption per device because the design point found with the advanced chromosome using 2 horizontal splits is shown to have a lower predicted energy consumption per device compared to the design points of the chromosomes with 3 and 4 splits but in reality, this is not true. On the other hand, all of the mappings found with the advanced chromosomes manage to outperform the lowest achieved real energy consumption of the mapping found using the naive chromosome. Figure 23 compares the real and predicted peak main memory usage per device of the best found points using each chromosome, with the y-axis showing the memory usage measured in Megabytes, while the x-axis denotes the chromosomes used to find the mapping. In this case, the mapping found with the advanced chromosome with 2 horizontal splits achieves the lowest real memory usage as predicted by the model. However, it is clear that the model again fails to predict correctly the memory usage, shown by the real result of the mapping found with the advanced chromosome with 4 splits, which has a higher real memory usage compared to the design points found with the other chromosomes, even though its predicted memory usage is the third out of six lowest. Lastly, the inference latency comparison between the real and predicted values of the best found points is shown in Figure 24, measured in milliseconds on the y-axis and on the x-axis the chromosome used to find each solution is indicated. This time, the model predicts correctly the inference latency, even though the accuracy is not good. The mapping found with the advanced chromosome with 2 splits achieves the lowest real inference latency and also has the lowest predicted latency. Furthermore, all of the design points found with the advanced chromosomes manage to outperform the design point found with the naive chromosome, both in predicted and real inference latency, as was expected by the results of the first experiment shown in Figure 19. Checking the real best results in each objective indicated in Table 5 (highlighted values), we can see that they are achieved by design points which were selected because they also achieved the lowest predicted inference latency. Thus, this means that the analytical model miss-predicts when evaluating the energy consumption and memory usage of mappings because the lowest real values should be achieved by design points that were selected because they achieve the lowest predicted value in the corresponding objective. On the other hand, comparing these best real values of Table 5 with the real evaluation of the un-partitioned model found in Table 2, shows that for all three objectives, the Hybrid Partitioning manages to find a mapping after 100 generations using the analytical model in the DSE process that achieves lower real results compared to the un-partitioned CNN. Therefore, even though the analytical model miss-predicts in terms of the peak main memory usage and energy consumption per device, it is useful to guide the DSE process towards the right direction.

Table 5: VGG results for Hybrid Partitioning using CPU only. Best points found for each goal from the set of Pareto-Optimal points with their real and model evaluations.

	Chromosome	Energy (J)		Memory (MB)		Latency (ms)	
		Real	Predicted	Real	Predicted	Real	Predicted
Best Energy Point Found	Naive	5.937	31.886	264.070	172.587	1299.100	3492.830
	Advanced - 2	3.323	1.410	211.129	188.839	112.860	154.467
	Advanced - 4	2.615	6.054	153.090	181.769	355.750	663.188
	Advanced - 6	4.921	25.912	256.523	232.938	1027.370	2838.439
Best Memory Point Found	Naive	6.547	58.561	256.730	166.088	1464.290	6414.910
	Advanced - 2	3.298	8.456	201.711	127.324	284.170	926.329
	Advanced - 4	4.651	35.622	260.438	130.335	750.900	3902.117
	Advanced - 6	4.970	31.777	254.980	133.210	943.330	3480.954
Best Latency Point Found	Naive	6.081	31.886	263.543	172.587	1317.220	3492.830
	Advanced - 2	**1.736**	1.410	212.496	188.839	**112.000**	154.467
	Advanced - 4	2.556	6.054	**148.973**	181.769	351.460	663.188
	Advanced - 6	5.969	25.912	254.715	232.938	1034.950	2838.439

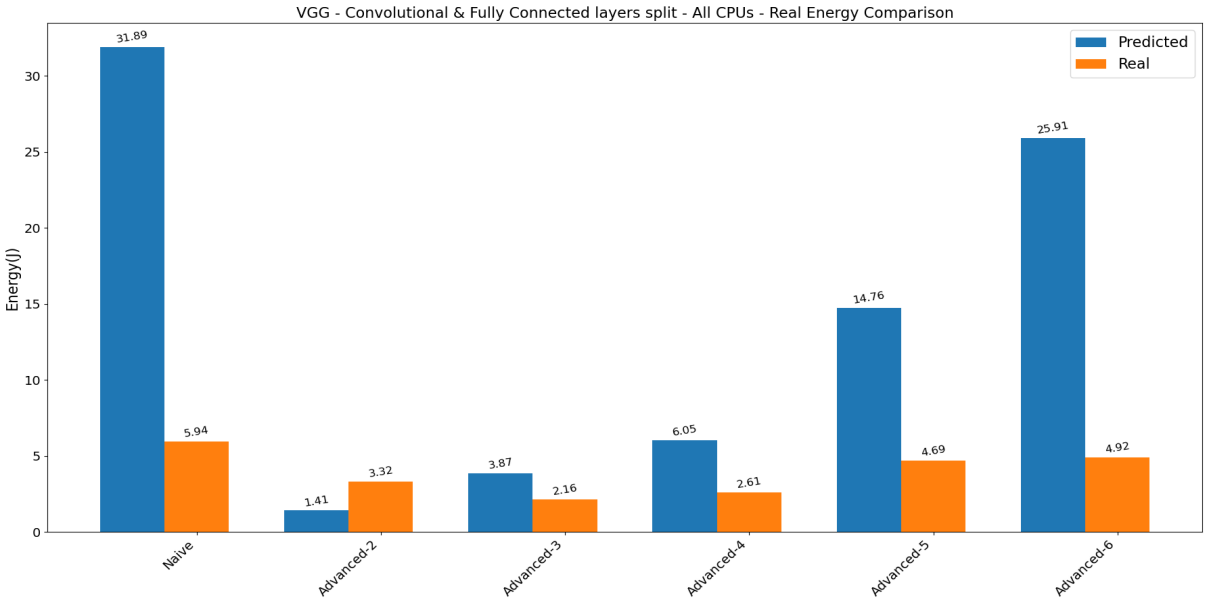


Figure 22: VGG comparison of predicted and real results for maximum energy usage per device when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

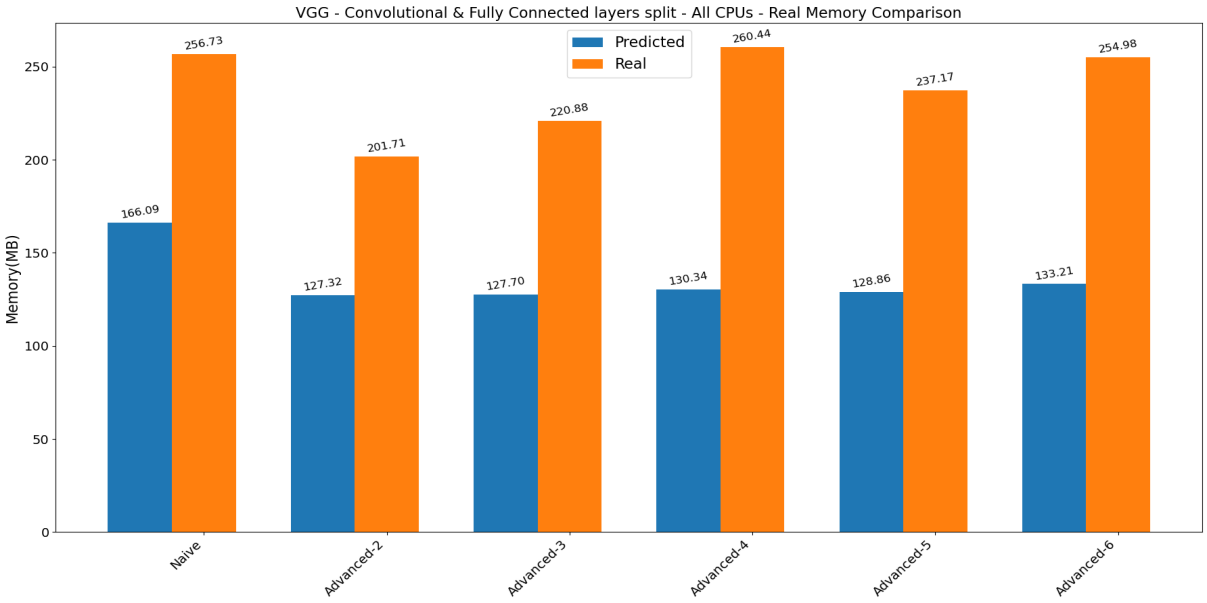


Figure 23: VGG comparison of predicted and real results for peak memory usage per device when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

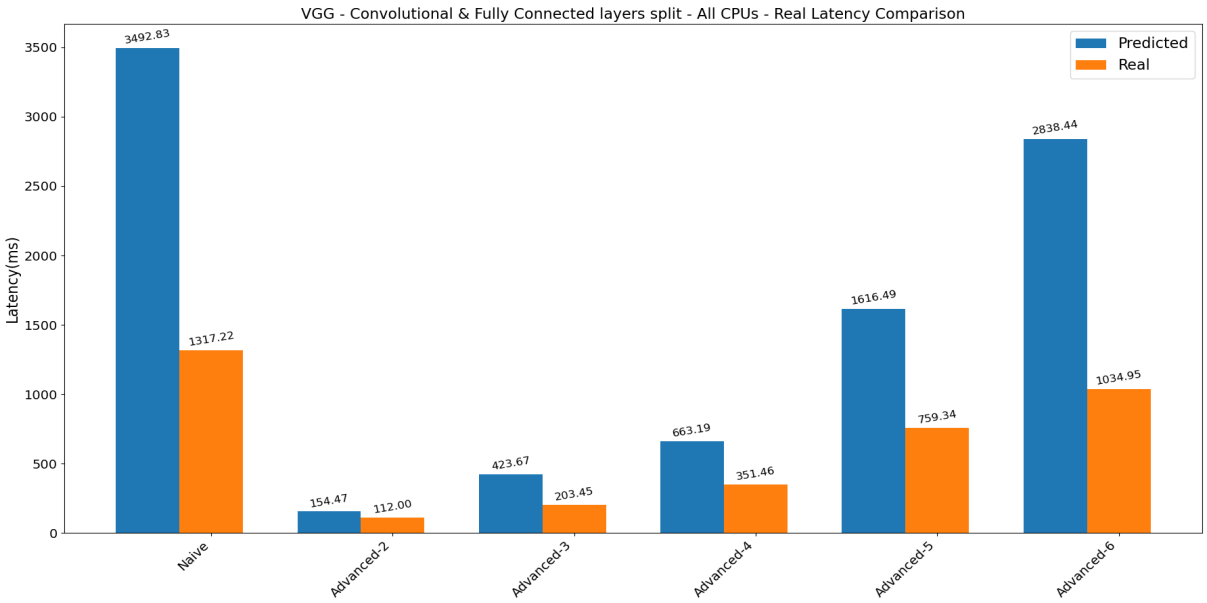


Figure 24: VGG comparison of predicted and real results for inference latency when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

5.2.3 DenseNet Results

The last CNN model included in the experiments is the DenseNet model. Since this CNN model has the most amount of layers, the size of the Design Space is larger than the other two CNN models included in the experiments. The first experiment is presented in Figure 25. The figure compares the convergence of the three objectives, energy consumption per device, peak main memory usage and inference latency, in 100 DSE generations. The first plot shows the comparison of the energy consumption convergence, measured in Joules on the y-axis. It is clear that the advanced chromosome exhibits faster convergence in terms of energy consumption than the naive chromosome. The chromosomes again converge in order of highest restriction, with the DSE using the advanced chromosome with 2 horizontal splits converging the fastest, while the DSE using the chromosome with 4 splits is second by a close margin. Due to the high complexity of the DenseNet CNN, it is evident from the plot that the DSE using the naive chromosome converges slower than the DSE runs using the advanced chromosomes, with the lowest energy value achieved by a mapping of the naive chromosome remaining very high after 100 generations. The second plot presents the peak main memory usage convergence during the 100 generations of DSE, with the y-axis showing the memory usage in Megabytes. The observations made about the energy consumption convergence remain true here. This time, the DSE runs utilizing the three advanced chromosomes shown find mappings with peak main memory values closer together after 100 generations. On the other hand, the DSE using the naive chromosome converges slowly, with the lowest main memory usage of the best mapping found being 179.430 Megabytes, while on the other hand, the DSE using the advanced chromosome with 2 horizontal splits converges the fastest, finding a mapping that achieves a peak main memory usage of 43.369 Megabytes, which is almost 4 times less than the naive chromosome's mapping. Lastly, the third plot presents the inference latency convergence in the 100 generations of DSE. The y-axis of the plot shows the inference latency in seconds. The plot follows the same trend as the energy comparison graph. The DSE using the advanced chromosome converges faster than the DSE using the naive chromosome, with the 2 horizontal splits chromosome exhibiting the fastest convergence out of all others. The DSE using the advanced chromosome with 2 horizontal splits finds the best mapping that achieves almost 10 times lower inference latency than the best mapping found by the DSE using the naive chromosome after 100 generations. It is clear from the experiment results that the advanced chromosome can handle the large complexity of the DenseNet CNN, allowing it to converge faster in all three objectives during the DSE compared to the naive chromosome.
Figure 26 presents the results of the second experiment. The difference between this experiment compared to the first is that now the DSE process uses CPUs and GPUs instead of only CPUs. This means that the size of the design space becomes even larger, with the number of available devices being doubled. The first plot compares the convergence of the energy consumption per device, measured in Joules on the y-axis. The plot shows a similar trend to the energy comparison of the first experiment. The difference is that this time, the DSE using the advanced chromosome with 2 horizontal splits converges faster with a larger margin than in the first experiment. Furthermore, this time, the best mapping found by the DSE using the naive chromosome achieved an energy consumption per device after 100 generations that is at least 7 times higher than the energy consumption per device achieved by the best mapping found by the DSE using the advanced chromosome with 2 splits. In this case, the mappings using both GPUs and CPUs, are not able to outperform the mappings found that use only CPUs after 100 generations. Moving to the second plot, which compares the convergence of peak main

memory usage per device, measured in Megabytes on the y-axis, we see a similar result. The DSE using the advanced chromosome outperforms the DSE using the naive chromosome. In this case, the three DSE runs using the advanced chromosomes converge at a similar pace and their memory usage per device values found by their best mapping after 100 generations are very close. Moreover, the memory usage per device of the best mapping found with the DSE using the naive chromosome is much higher when compared to that of the best mapping found by the DSE with the advanced chromosome with 2 splits. Furthermore, after 100 generations, the mappings found using CPUs and GPUs in combination do not give better results in terms of peak main memory usage when compared to the mappings that use only CPUs. Lastly, the third plot presents the inference latency, measured in seconds on the y-axis. This plot shows a similar trend to the energy consumption convergence graph. The three advanced chromosomes exhibit faster convergence than the naive. The best mapping found with the DSE using the advanced chromosome with 2 horizontal splits achieves the lowest inference latency value after 100 generations followed closely by the mapping found with the DSE using the 4 splits chromosome, while the mapping of the DSE using the 6 splits chromosome falls behind. The DSE with the naive chromosome again converges at the slowest pace, with its best mapping achieving an inference latency that is much larger when compared to any of the values achieved with the best mappings found by the DSE runs using the advanced chromosomes. In this case, using CPUs and GPUs, the advanced chromosomes are able to find mappings that come close to the achieved results of mappings that only use CPUs. Based on these results, it is clear that the advanced chromosome is able to handle the increased complexity of the design space better than the naive chromosome because it exhibits faster convergence in all cases. In addition, in terms of inference latency, after 100 generations the usage of both CPUs and GPUs shows promising results, although, in terms of energy consumption and memory usage, the results of the best found mappings are worse.

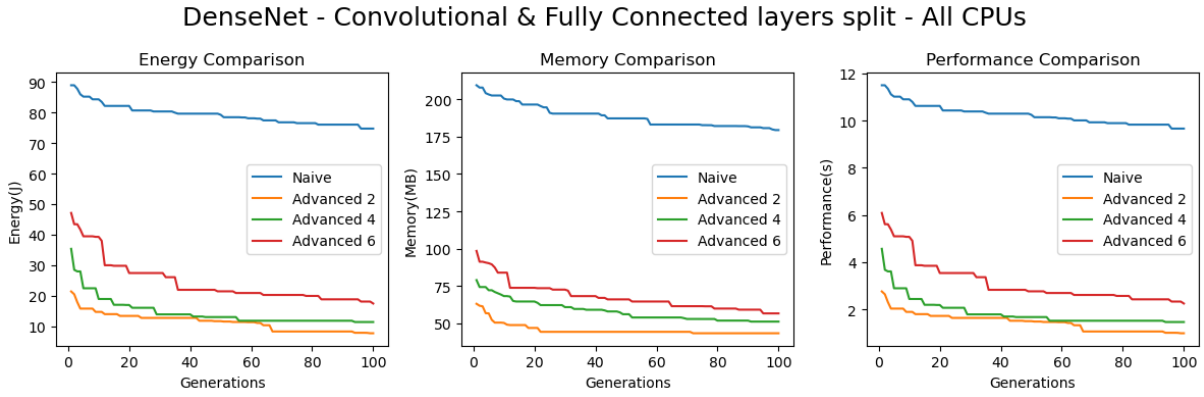


Figure 25: DenseNet DSE convergence for hybrid partitioning when splitting Convolutional and Fully Connected layers and using only CPUs.

The results of the third experiment using the DenseNet network are presented in Figure 27. In this experiment, the convergence of the Vertical Partitioning and the Hybrid Partitioning is compared based on energy consumption per device, peak main memory usage and inference latency. The Vertical Partitioning uses the Split Point encoding [18] while the Hybrid Partitioning uses the advanced chromosome with 2 horizontal splits because it found the best mappings in the first experiment. In this case, only CPUs are used to create the mappings. The first plot compares the energy consumption per device, measured in Joules on the y-axis. It is clear that

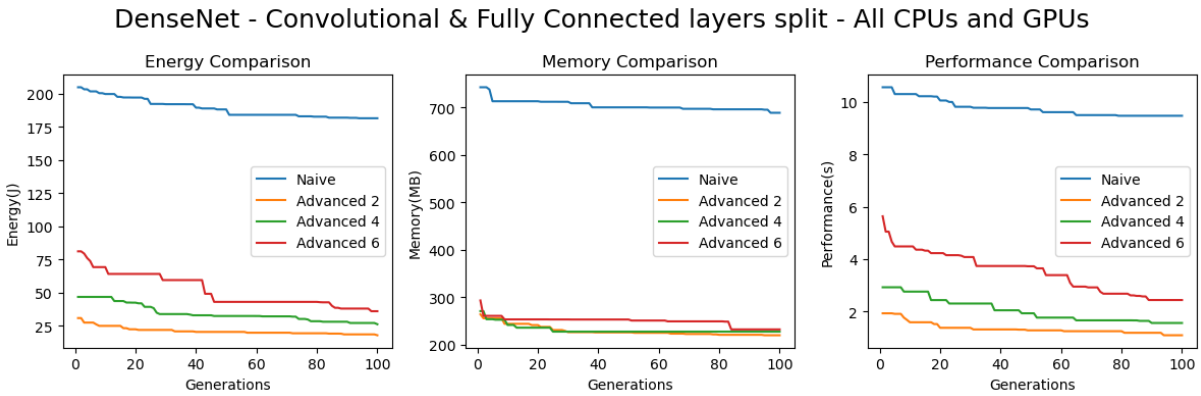


Figure 26: DenseNet DSE convergence for hybrid partitioning when splitting Convolutional and Fully Connected layers and utilizing CPUs and GPUs.

in 100 generations the DSE using the Vertical Partitioning converges much faster in terms of energy consumption, with its best found mapping achieving an energy consumption per device that is lower by 7 times when compared to the value of the best found mapping of the DSE using the Hybrid Partitioning. Moving to the second plot, it compares the peak main memory usage measured in Megabytes on the y-axis. Again, the best mapping of the DSE using the Vertical Partitioning outperforms the best mapping of the DSE using the Hybrid Partitioning by a margin of 2.2 times. The last plot compares the inference latency convergence, measured in seconds on the y-axis. The same observation is made here as well, which is that the DSE using the Vertical Partitioning converges faster than the DSE using the Hybrid Partitioning and this time the performance difference of the best found mappings is 7.2 times. Based on the above results, when using the DenseNet network, the DSE using the Vertical partitioning after 100 generations is able to find mappings that achieve better results than the DSE using the Hybrid Partitioning, outperforming it in all three objectives.

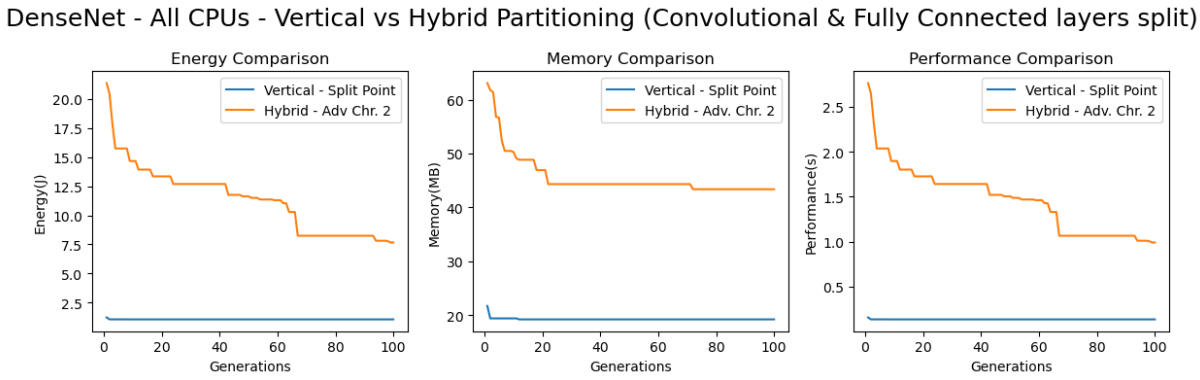


Figure 27: DenseNet DSE convergence comparing the Vertical and Hybrid Partitioning. Vertical Partitioning uses the Split Point encoding [18]. Hybrid Partitioning uses the Advanced Chromosome with 2 horizontal splits for Convolutional and Fully Connected layers. Only CPUs are used.

The results of the fourth experiment are shown in Table 6. In this experiment, the best design points found in terms of energy consumption per device, peak main memory usage per device and inference latency have been selected from the set of Pareto-Optimal points found with

Table 6: DenseNet Pybrid Partitioning results using CPU only. Best points found for each goal from the set of Pareto-Optimal points with their real and model evaluations.

	Chromosome	Energy (J)		Memory (MB)		Latency (ms)	
		Real	Predicted	Real	Predicted	Real	Predicted
Best Energy Point Found	Naive	18.699	74.729	240.289	185.586	4592.09	9661.204
	Advanced - 2	3.944	7.664	93.609	65.434	740.1	990.801
	Advanced - 4	5.982	11.342	78.406	65.452	1257.01	1466.380
	Advanced - 6	6.548	17.406	104.316	64.251	1417.06	2250.321
Best Memory Point Found	Naive	18.558	89.388	243.676	179.430	4603.02	11556.289
	Advanced - 2	**3.107**	18.184	**70.152**	43.369	**479.95**	2350.934
	Advanced - 4	3.926	22.495	77.777	51.167	734.89	2908.266
	Advanced - 6	5.129	43.050	107.508	56.660	1038.13	5565.628
Best Latency Point Found	Naive	18.371	74.729	242.832	185.586	4557.05	9661.204
	Advanced - 2	3.788	7.664	91.785	65.434	714.59	990.801
	Advanced - 4	5.971	11.342	78.648	65.452	1277.46	1466.380
	Advanced - 6	6.642	17.406	104.402	64.251	1426.0	2250.321

the DSE of each chromosome after the first experiment. Recall that the first experiment used only CPUs and partitioned Convolutional and Fully Connected layers horizontally. These design points have been implemented using AutoDice and were run on the real distributed system to collect their performance measures. The rows of the table show the real values measured on the distributed system and the corresponding predicted values by the analytical model, for each design point. The third and fourth columns show the real and predicted energy consumption per device in Joules of each design point. The fifth and sixth columns show the real and predicted peak main memory usage in Megabytes. The last two columns present the real and predicted inference latency measured in milliseconds. Figure 28 visualizes the results for the best design points found in terms of energy consumption per device and their results are also shown with the first square with highlighted border on Table 6. The second square with highlighted border on Table 6 shows the memory results for the best found design points in terms of peak main memory usage and they are also visualized in Figure 29. Lastly, the third highlighted square in the table shows the inference latency results of the best found design points in terms of inference latency from the set of Pareto-Optimal points of each chromosomes' DSE and the results are visualized in Figure 30. The x-axis of all three figures denotes the chromosome used to find each mapping. Examining Figure 28, we can see that the mapping found with the advanced chromosome with 2 splits is predicted to have the lowest energy consumption per device and this is true in reality. All of the results of the mappings found with the advanced chromosomes outperform the result of the mapping found with the naive chromosome, both in terms of real and predicted energy. It is also evident that the model miss-predicts in one case, where the mapping found with the advanced chromosome with 3 splits is shown to have a higher predicted energy consumption than the mapping found with the 4 splits chromosome but in reality, this is not true. Examining the memory usage results in Figure 29, we find that again all the results of the mappings found with the advanced chromosomes outperform the result of the mapping found using the naive chromosome both in predicted and real memory usage. Based on the chart, the analytical model this time does not produce any wrong results because the real values measured are in the same order as the predicted values

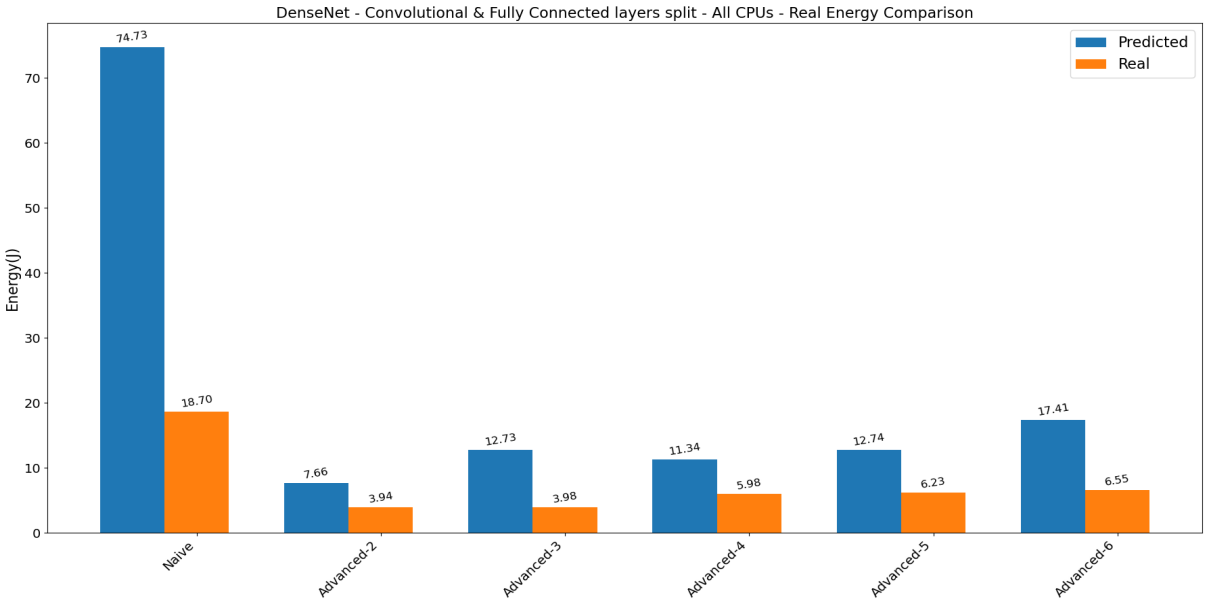


Figure 28: DenseNet comparison of predicted and real results for maximum energy consumption per device when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

of the analytical model. Lastly, the same observations made for the memory usage results are true about the inference latency results of Figure 30. The analytical model does not show any miss-predictions and all of the mappings found using the advanced chromosomes outperform the mapping found using the naive chromosome in the real results as well as the predicted results. Comparing the above results with the results of the un-partitioned CNN model shown in Table 2, shows that the analytical model works well for guiding the DSE process towards points with lower memory usage per device because the process found mappings with lower real memory usage per device than the un-partitioned network. In terms of energy consumption per device and inference latency, after 100 generations the DSE process cannot find a mapping that outperforms the performance of the un-partitioned CNN model in these two objectives. Furthermore, the predictions of the analytical model are usually not close to the real values obtained using the distributed system in most cases by a large margin.

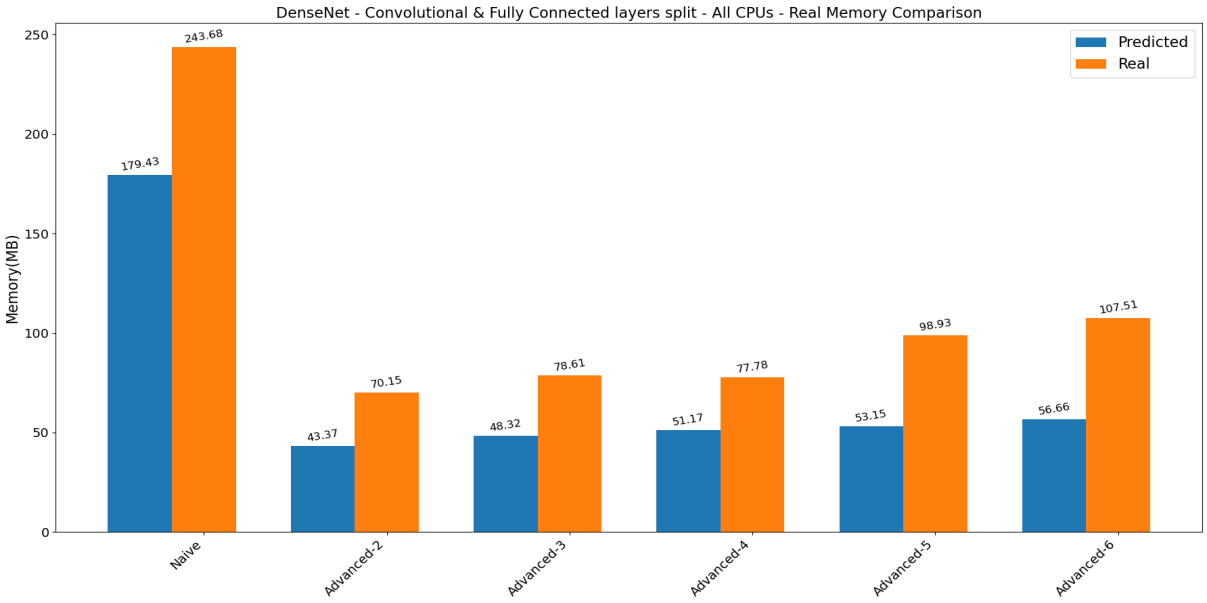


Figure 29: DenseNet comparison of predicted and real results for peak memory usage per device when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

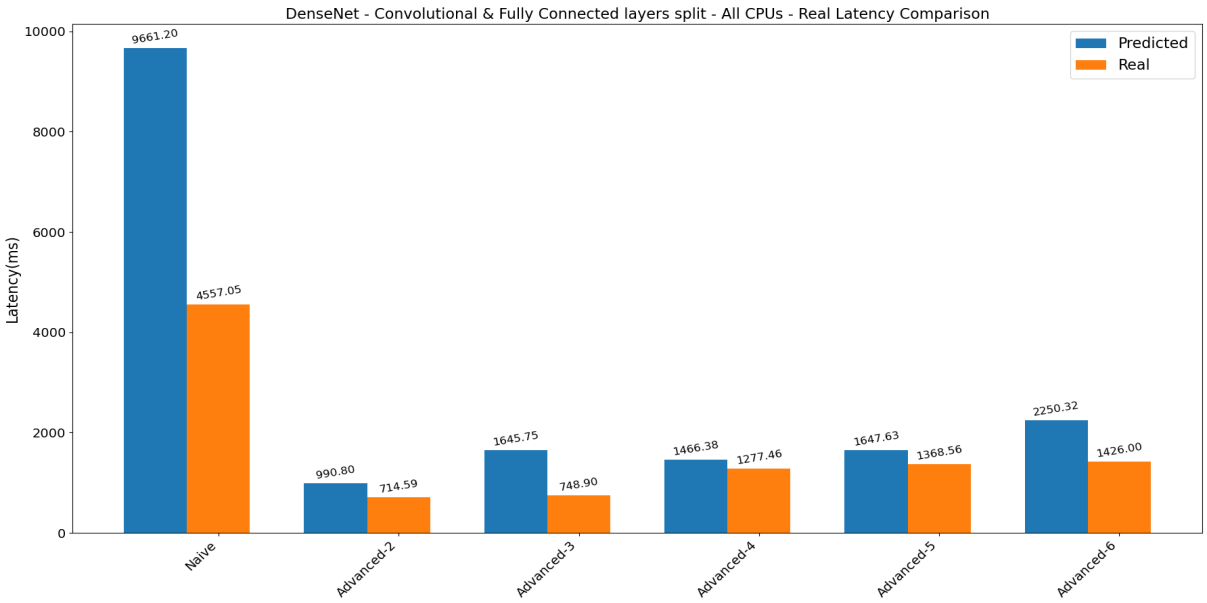


Figure 30: Densenet comparison of predicted and real results for inference latency when splitting Convolutional and Fully Connected layers with Hybrid Partitioning, using CPUs.

6 Discussion and Future work

In this section, the results presented in Section 5.2 and obtained from the experiments conducted on the different CNN models are analyzed. The performance of the advanced chromosome is compared to the naive chromosome, and the benefits and limitations of the proposed hybrid partitioning strategy are highlighted. Furthermore, the accuracy of the analytical model is discussed.

The potential of the presented hybrid partitioning strategy (described in Section 4.1) is demonstrated through the experimental results presented in this research. First of all, the Hybrid Partitioning is compared to the un-partitioned results for each CNN network. The real measured performance results of the un-partitioned CNNs are shown in Table 2. Table 7 shows the percentage improvement achieved when using the Hybrid Partitioning in each objective for every CNN network, compared to the un-partitioned values. The columns of the table show the CNN networks while the rows show the objectives. The dashes in the table are placed where there was no improvement for the CNN on a given objective. The table shows that for AlexNet, there is no improvement in terms of energy consumption per device. From Table 2 we see that for the un-partitioned AlexNet CNN the energy consumption is 0.352 Joules while checking the real results in Figure 15 we cannot find a case where 100 generations of DSE with the Hybrid Partitioning could find a mapping with lower real energy consumption per device. On the other hand, in terms of peak main memory usage per device, Table 7 shows a 72.64% improvement and for the inference latency a 6.68% improvement. The numbers can be found when comparing the real results of Table 3 with the results in Table 2. For the VGG network, Table 7 shows an 11.38% improvement in terms of energy consumption per device, 66.66% improvement for peak main memory usage and an 8.54% improvement in inference latency when using Hybrid Partitioning compared to the un-partitioned performance. These numbers can be derived when comparing the real results in Table 5 with the results in Table 2. Lastly, for the DenseNet CNN network, Table 7 shows a 26.9% improvement in the peak main memory usage per device when comparing the real results in Table 6 with the results in Table 2 and no improvement in terms of energy consumption per device and inference latency in 100 DSE generations. Therefore, based on the above results, we can conclude that the Hybrid Partitioning can deliver lower peak main memory usage per device in all three CNNs in just 100 DSE generations, while for AlexNet and VGG it can also improve in terms of inference latency when compared to the un-partitioned networks. To answer the research question of whether the Hybrid Partitioning offers benefits over the Vertical Partitioning, we need to examine Figures 14, 21 and 27. For AlexNet, Figure 14 shows that in terms of peak main memory usage, the Hybrid Partitioning can outperform the Vertical Partitioning. Furthermore, it also shows promising results in terms of energy consumption per device and inference latency because even though the DSE does not find a mapping with better results in these two objectives after 100 generations, it manages to come close to the results of the mapping found with the DSE using the Vertical Partitioning. For the VGG network, Figure 21 shows similar trends. In terms of peak main memory usage, the Hybrid Partitioning outperforms the Vertical Partitioning, while coming close in terms of energy consumption per device and inference latency. On the other hand, when examining Figure 27 for DenseNet, it is evident that the Hybrid Partitioning cannot outperform the Vertical Partitioning in any objective after 100 generations of DSE. This is because the DenseNet CNN has a high number of layers, thus the size of the design space becomes very large. The size of the design space is smaller when using the Vertical Partitioning strategy than the size of the space when using the Hybrid Partitioning, and therefore

the DSE with the Vertical Partitioning can outperform the DSE with the Hybrid Partitioning in 100 DSE generations. Based on the above discussion, we can conclude that the Hybrid Partitioning offers lower peak main memory usage per device than the Vertical Partitioning in both AlexNet and VGG CNNs. Furthermore, the results show the potential of the Hybrid Partitioning to outperform the Vertical Partitioning in terms of energy consumption per device and inference latency if we continue the DSE process for more than 100 generations. For the DesneNet CNN, the DSE using the Hybrid Partitioning needs more generations in order to find a mapping that comes close to the results of the mapping found with DSE using the Vertical Partitioning, but we cannot say for sure whether this will happen because the Vertical Partitioning might still improve on the objectives if more DSE generations are performed.

Table 7: The % improvement in each objective when comparing the Hybrid Partitioning strategy results (CPUs, Horiz. splitting Conv. and FC layers) with the un-partitioned results. Dashed line on the objectives which where not improved.

	AlexNet	**VGG**	**DenseNet**
Energy % Improvement	-	11.38	-
Memory % Improvement	72.64	66.66	26.9
Latency % Improvement	6.68	8.54	-

Based on the results presented in Section 5.2, it is evident that the DSE process utilizing our advanced chromosome consistently converges faster than the DSE with the naive chromosome and therefore finds better mappings in each objective in most cases. Starting with AlexNet, Figure 12 shows that DSE using the advanced chromosome converged faster in all objectives and found a mapping producing better results with the analytical model after 100 generations when compared to the mapping found by the DSE utilizing the naive chromosome. Comparing the real results shown in Figures 15, 16, 17 indicates that only in terms of the real measured peak main memory usage the mapping found by the DSE using the naive chromosome achieved a lower real value after 100 generations, while in the other objectives, the mappings found with the DSE runs of the advanced chromosomes produce lower values.
This can be explained in two ways. First, the mapping produced by the DSE using the naive chromosome achieves lower memory usage because the naive chromosome is not as restrictive as the advanced chromosomes and allows for greater flexibility and more granularity in terms of the vertical partitioning of layers, thus enabling a distribution of layers among the devices that equally divides the main memory usage in a way that the advanced chromosome cannot achieve. The second explanation involves the analytical model. As shown in Figure 16 the model inaccurately estimates the overall memory usage per device because it assigns a lower predicted main memory usage per device on each of the mappings found with the DSE runs using the advanced chromosomes than the mapping of the DSE found using the naive chromosome. Therefore, due to the miss-predictions of the analytical model, in the case of AlexNet, the DSE process using the advanced chromosomes might not be able to explore mappings offering lower memory usage per device. In the second experiment, where both CPUs and GPUs are used to handle the inference, the DSE utilizing the advanced chromosome finds mappings with better predicted scores after 100 generations than the DSE using the naive chromosome as

shown in Figure 13. This was expected because adding GPUs to the pool of devices increases the size of the design space, therefore the advanced chromosome is able to better handle the complexity.
As for VGG, the advanced chromosome demonstrates even better results. With the higher complexity introduced because of the CNN, both in terms of layers and amount of parameters, the DSE using the advanced chromosome shows great potential. In all three objectives, the DSE with the naive chromosome converges slower than the DSE utilizing the advanced chromosome. This can be seen in Figure 19 which compares the convergence in each objective for 100 generations. Furthermore, comparing the real results presented in Figures 22, 23 and 24, shows that in all three objectives, the mappings found with the DSE runs using the advanced chromosomes in almost all cases give lower values in each objective than the mappings found with the DSE utilizing the naive chromosome. The only case where the mapping found with the DSE using the naive chromosome gives a lower value is in terms of memory usage per device when compared to the mapping produced by the DSE using the advanced chromosome with 4 splits. In all other cases, the results of the mappings found with the DSE using the advanced chromosomes are better. Figure 20 presents the convergence after 100 generations of the DSE when using both CPUs and GPUs and again in this case, the advanced chromosome is able to handle the complexity and converge faster than the naive chromosome.
Finally, the DenseNet results align with those of VGG, with the difference being that DenseNet has more layers but fewer parameters. In both experiments depicted in Figure 25 and 26, the DSE with the advanced chromosome is able to converge faster and is able to find mappings that produce better results than the mappings found with the DSE using the naive chromosome after 100 generations in all three objectives. Comparing the real results of the experiment using only CPUs in Figures 28, 29 and 30 also verifies that the DSE using the advanced chromosome finds mappings that outperform the mappings found with the DSE using the naive chromosome after 100 generations in all three objectives.
Based on these results, it becomes clear that the DSE utilizing the advanced chromosome is superior to the DSE using the naive chromosome, finding mappings after 100 generations that produce better results. In addition, the best scores achieved in each objective are presented in bold in Tables 3, 5, 6. An important observation is that all of the best scores are achieved by mappings found with the advanced chromosome.
To answer the third research question, which investigates whether there are improvements when using both CPUs and GPUs when performing CNN inference with Hybrid Partitioning, we need to compare the first and second experiments of each CNN. First, for AlexNet, we need to compare the results in Figure 12 against the results in Figure 13. In terms of energy consumption per device, using both CPUs and GPUs cannot offer lower energy consumption after 100 generations although it comes close to the result of only using CPUs. In terms of peak main memory usage, after DSE of 100 generations using both CPUs and GPUs cannot offer lower memory usage compared to only using CPUs. Lastly, the inference latency of using both CPUs and GPUs is slightly lower than when using only CPUs after 100 generations. As for VGG, comparing the plots in Figure 19 against the plots in Figure 20 shows that in terms of energy consumption per device and peak main memory usage, the mappings using both CPUs and GPUs cannot outperform the mappings using only CPUs after 100 DSE generations. In terms of inference latency, both solutions end up close to each other after 100 generations but still using only CPUs outperforms the solution with both CPUs and GPUs. Lastly, for DenseNet, we need to compare Figure 25 and Figure 26. The comparison yields similar conclusions, where in terms of energy and memory, using both CPUs and GPUs cannot outperform the best achieved

values of using only CPUs after 100 DSE generations and in terms of inference latency using both CPUs and GPUs comes close to the result of using only CPUs. Therefore, based on the above analysis, using both CPUs and GPUs is possible to offer potential improvements in terms of inference latency, but more DSE generations are required in order to be able to confirm this. In terms of energy consumption per device and peak main memory usage after 100 DSE generations using only CPUs yields far superior results.

Regarding the analytical model developed in this research, the presented evidence indicates that it lacks the required accuracy to provide close prediction numbers to the real measured values. However, it only miss-predicts trends in limited cases, where it cannot predict correctly the performance of a solution and shows that a different solution can achieve better results when this is not true. This can be seen in the figures and tables comparing the real and predicted values of each CNN, Figures 15, 16, 17 and Table 3 for AlexNet, Figures 22, 23, 24 and Table 5 for VGG, Figures 28, 29, 30 and Table 6 for DenseNet. Several factors contribute to the deviation of the results given by the model. First of all, it relies heavily on the MAC operations calculated for each layer along with the calibration coefficients found during model tuning for the inference latency. However, there exists a very high variance when calculating the coefficients for the same type of layers. This variance, when averaged to retrieve the coefficient for a layer type, introduces errors in the model. The same is true for the memory and energy calibration coefficients. The different implementations available for every layer type contribute to the substantial variance, making accurate prediction of execution time for each layer a difficult task. Furthermore, accurately predicting memory usage is challenging due to the variety of implementations for each layer which affects memory usage and the difficulty of estimating the memory used by loaded libraries. The simple prediction method employed for energy consumption fails to accurately capture the complexity of the energy consumption, neglecting factors such as varying energy usage during startup, communication, and different operations.

However, the aforementioned limitations do not suggest that our analytical model is useless. The conducted experiments show that the model is still useful in guiding the DSE process towards the right direction. Since the model is based on MAC operations for the inference latency and the network speeds for the communication time, it can effectively steer the DSE process toward lowering the MAC operations and communication, therefore lowering the overall inference latency. This is because, for every mapping given, the model will give a different evaluation for inference latency based on the MAC operations and communication time spent on each device. The actual DSE process minimizes the inference latency, even though the predicted absolute values are not close to the real ones. The same is true for the main memory usage model. Since the model is based on the tensor sizes for main memory usage and the memory usage during communication, which are the two prominent memory usage reasons for a mapping, each different mapping will get a different evaluation in terms of memory. For the energy consumption model, the simple assumption that the more time spent operating on each device, the higher the energy consumption, is also moving the DSE process towards the right direction, although it cannot really capture the complexity of the energy consumption. All the calibrating coefficients help our analytical model adjusting to the case of the distributed system and adjusting the values between CPUs and GPUs, which are important because they affect execution time, communication time and energy consumption. Therefore, the model can be used for the DSE process up to a point, after which the DSE can be continued by numbers obtained from the real devices to get the optimal mapping.

As future work of this project, there are several potential avenues to explore. Firstly, the DSE

process itself can be enhanced. It is hypothesised that by allowing more generations to the DSE utilizing the advanced chromosome, it will be able to further improve the design points and still converge faster than the DSE using the naive chromosome. Furthermore, such experiments may show more improvements using the Hybrid Partitioning over the Vertical Partitioning. Additionally, exploring the use of the advanced chromosome with a higher number of horizontal splits, coupled with an extended number of DSE generations, could help determine the optimal number of splits for each CNN.

Further improvements can be made to the DSE process. One approach involves utilizing the current analytical model up to a certain point, after which the DSE can continue for a specified number of generations using the real distributed system as a fitness function. This hybrid approach would provide a better process for determining the optimal solution because the first phase of the DSE would guide the GA towards the correct direction and in the subsequent phase, the GA would be able to use the real evaluations of each found mapping and therefore find more optimal solutions. A similar approach can also be taken when using CPUs and GPUs together. In this case, the first phase of the DSE could be done using only CPUs and after the specified amount of generations passes, it can continue by using both CPUs and GPUs. This phased approach would aid the DSE convergence of the more complex scenario, as it would initiate the search for an optimal solution using both CPUs and GPUs from a favourable starting point, which thus may prove that there are more possible improvements when using both CPUs and GPUs.

Another improvement to the project involves the analytical model. It is clear that the current model struggles to capture the complexity of the mappings and provide absolute prediction numbers closer to the real measured numbers. This is due to the existence of non-linearities, for example in the energy consumption. To address this limitation, alternative modelling techniques could be explored such as training a machine learning predictive model to predict each performance aspect.

References

[1] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[2] Xiaotian Guo, Andy Pimentel, and Todor Stefanov. Automated exploration and implementation of distributed cnn inference at the edge. *IEEE Internet of Things Journal*, PP:1–1, 04 2023.

[3] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, jan 2017.

[4] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[5] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[7] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.

[9] Erqian Tang, Xiaotian Guo, and Todor Stefanov. The effects of partitioning strategies on energy consumption in distributed cnn inference at the edge. *arXiv preprint arXiv:2210.08392*, 2022.

[10] Agoston E. Eiben and James E. Smith. *Introduction to evolutionary computing*. Natural computing series. Springer, second edition edition, 2015.

[11] N. Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[12] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, jan 1998.

[13] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[14] The Khronos® Vulkan Working Group. Vulkan Specification. `https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html`. Accessed: July 16, 2023.

[15] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 836–838, 2008.

[16] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[17] Erqian Tang and Todor Stefanov. Low-memory and high-performance cnn inference on distributed systems at the edge. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, UCC '21, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Xiaotian Guo, Andy D Pimentel, and Todor Stefanov. Hierarchical design space exploration for distributed cnn inference at the edge. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 545–556. Springer, 2022.

[19] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

[20] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1396–1401, 2017.

[21] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. Codl: Efficient cpu-gpu co-execution for deep learning inference on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 209–221, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Rafael Stahl, Zhuoran Zhao, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Fully distributed deep learning inference on resource-constrained edge devices. In Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 77–90, Cham, 2019. Springer International Publishing.

[23] Shuai Zhang, Sheng Zhang, Zhuzhong Qian, Jie Wu, Yibo Jin, and Sanglu Lu. Deepslicing: Collaborative and adaptive cnn inference with low latency. *IEEE Transactions on Parallel and Distributed Systems*, 32(9):2175–2187, 2021.

[24] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[25] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. Learning search space partition for black-box optimization using monte carlo tree search. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 19511–19522. Curran Associates, Inc., 2020.

[26] Hui Ni and The ncnn contributors. ncnn, June 2017.

[27] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

[28] NVIDIA. NVIDIA jetson xavier nx. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/`.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[30] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.