Deep Learning on the Edge: A Flexible Multi-Level Optimization Approach

Nesma Rezk

Supervisors:
Magnus Jonsson
Mahdi Fazeli
Abstract

Recent advances in Deep Learning (DL) research have been adopted in a wide variety of applications, including autonomous driving, AI in health care, and smart homes. In parallel, research in high-performance embedded computing has resulted in advanced hardware platforms that offer enhanced performance and energy efficiency for demanding computations. However, the high demands of DL models for computational and memory resources are still a challenge for embedded computing.

Algorithmic optimizations can be used to reduce the computational and memory requirements of DL models. Hardware implementations and architectures can also be tuned to support DL applications’ requirements. This thesis identifies that insufficient coordination between hardware implementations and models’ optimizations limits the efficiency of the resulting implementations. In addition, the implementation methods themselves suffer from poor flexibility in adapting to changes in the model and application constraints. The overarching theme of this thesis is to study and propose methods for the efficient and flexible implementation of DL models on embedded platforms.

The work in this thesis bridges the gap between DL models’ algorithmic optimizations and embedded platforms’ hardware-specific optimizations, and investigates the features that need support from DL domain-specific architectures. In addition, a method for multi-objective quantization of DL models is proposed to address both the model error and platform performance metrics. Post-training optimization techniques are employed to facilitate the multi-objective optimization of the models because they do not require retraining after model optimization.

This thesis also reviews the optimization methods that are known to have been applied to improve the implementation efficiency of DL models. It highlights the most fruitful optimizations found in existing, highly efficient implementations, and applies them in the proposed methods. A method for mapping Convolution Neural Networks (CNN) on Epiphany, a manycore architecture, is proposed and evaluated. A method for quantization and approximation for RNN models in a post-training fashion is also proposed, and evaluated on four RNN models. The proposed quantization method is used in a hardware-aware multi-objective optimization for RNN models to be deployed on SiLago and Bit fusion architectures.
To my parents Mohammad and Eman
Acknowledgements

First and foremost, I would like to praise Allah the almighty, the most gracious, and the most merciful for his blessing given to me during my life and in completing this thesis.

I want to express my gratitude to my supervisors, Magnus Jonsson and Mahdi Fazeli, for believing in me and giving me a hand when it was dark. Thanks for your brilliant ideas and encouragement. Also, I would like to thank Tomas Nordström and Madhura Purnaprajna for all the thoughtful discussions we had. These discussions helped me build my ideas and reach my goals. And I want to thank Zain Ul-Abdin for his guidance during the period of my Ph.D. and Eren Aksoy for providing me with useful advice while working on this thesis. I appreciate the immense support from Stefan Byttner, Veronica Gaspes, Pererik Andreasson, and Håkan Pettersson that helped me complete this thesis and overcome all the struggles.

Great colleagues have surrounded me at Halmstad University during this journey. I express my gratitude for the beautiful friendship I have with Taqua and Amira. Special thanks to Mohammad Eldefrawy, Sundas, Nicolina, Felipe, Ali, Wagner, Olga, Rasool, Mark, Malin, Hazem, Sina, and all the wonderful colleagues I was lucky to meet every day. Also, I would like to thank Tiago Cortinhal for helping me set up the python libraries and Struan Gray for his generous proofreading whenever I ask for it.

I want to thank Prof. M. Watheq El Kharashi for his constant guidance and support since my graduation and till now. Also, thanks to my dearest friend Yasmeen Farag, who I am blessed to have in my life. No gratitude or acknowledgment would be enough for my parents. You have surrounded me with love, advice, and prayers in every phase of my life. Finally, thanks to Mohammad Zarzoura, my husband. You are a partner in every success I achieve in my life. Thanks for believing in me and supporting me all the time. My dear sons and brothers, Malek, Yamen, Essam Eldin, and Magd Eldin, you are the joy of my life.
List of Papers

The following four papers, referred to in the text by their Roman numerals, are included in this thesis.


PAPER III: Nesma M. Rezk, Tomas Nordström, Dimitrios Stathis, Zain Ul-Abdin, Eren Erdal Aksoy, and Ahmed Hemani, "MOHAQ: Multi-Objective Hardware-Aware Quantization of Recurrent Neural Networks," *Journal of Systems Architecture*, 2022. *(Accepted for publication)*

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>List of Papers</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Challenges facing neural networks implementations on embedded platforms</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Computation challenge</td>
<td>4</td>
</tr>
<tr>
<td>1.2.2 Memory challenge</td>
<td>5</td>
</tr>
<tr>
<td>1.2.3 Accuracy challenge</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Thesis Objectives</td>
<td>5</td>
</tr>
<tr>
<td>1.3.1 Implementation Efficiency</td>
<td>5</td>
</tr>
<tr>
<td>1.3.2 Flexibility</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Research Questions</td>
<td>8</td>
</tr>
<tr>
<td>1.5 Approach</td>
<td>8</td>
</tr>
<tr>
<td>1.5.1 Exploration of flexible methods for efficient realization of DL models on embedded platforms</td>
<td>8</td>
</tr>
<tr>
<td>1.5.2 Proposed flexible methods for efficient realization of DL models on embedded platforms</td>
<td>11</td>
</tr>
<tr>
<td>1.6 Contributions</td>
<td>12</td>
</tr>
<tr>
<td>1.7 Thesis Structure</td>
<td>12</td>
</tr>
<tr>
<td>2 Background</td>
<td>15</td>
</tr>
<tr>
<td>2.1 Neural Networks Layers</td>
<td>15</td>
</tr>
<tr>
<td>2.1.1 Convolution layer</td>
<td>15</td>
</tr>
</tbody>
</table>
2.1.2 Recurrent layer .................................. 15
2.1.3 Fully connected (FC) layer ........................ 19
2.1.4 Projection Layer .................................. 19
2.2 Architectures under study ............................. 21
  2.2.1 Epiphany architecture ............................ 21
  2.2.2 SiLago architecture .............................. 21
  2.2.3 Bit fusion architecture ......................... 22
2.3 Algorithmic Optimizations ............................. 22
  2.3.1 Quantization ................................. 22
  2.3.2 Delta networks method ........................... 23
2.4 Multi-objective optimization ........................... 24

3 Related Work ........................................... 27
  3.1 Optimizations applied to DL implementations ........ 27
  3.2 Flexibility in deep learning implementations on embedded platforms ................................... 28
    3.2.1 Flexible domain specific architectures .......... 28
    3.2.2 Hardware-aware optimization .................... 29
  3.3 Research Gaps ..................................... 30

4 Efficient Implementation of Deep Learning Models on Embedded Platforms .................. 31
  4.1 Study of the efficient implementations of DL models on embedded platforms ...................... 31
  4.2 Proposed methods for efficient implementations of DL models on embedded platforms ........ 33
    4.2.1 Mapping of CNN convolution to the Epiphany many-core architecture ......................... 33
    4.2.2 Optimization of RNN models .................... 35

5 Flexibility in Deep Learning Models Realization on Embedded Platforms .................. 39
  5.1 Flexibility of the DL hardware implementations ....... 39
    5.1.1 Study of flexibility in RNN implementations on embedded platforms ....................... 39
    5.1.2 Flexible implementation of CNN on the Epiphany architecture ................................. 40
    5.1.3 Flexible architectures for DL models ............ 41
  5.2 Flexibility of RNN models optimization ............... 42
    5.2.1 Multi-Objective Hardware-Aware Quantization (MO-HAQ) ........................................... 42
5.2.2 Comparison of RNN models under different quantization and delta method configurations

6 Summary of Contributions

6.1 Paper I: Recurrent Neural Networks: An Embedded Computing Perspective

6.2 Paper II: Streaming Tiles: Flexible Implementation of Convolution Neural Networks Inference on Manycore Architectures

6.3 Paper III: MOHAQ: Multi-Objective Hardware-Aware Quantization of Recurrent Neural Networks

6.4 Paper IV: Shrink and Eliminate: A Study of Post-training Quantization and Repeated Operations Elimination in RNN Models

7 Discussion and Limitations

8 Conclusion and Future Outlook

8.1 Conclusion

8.2 Future Outlook

References

Appendix Appendix A

Appendix Appendix B

Appendix Appendix C

Appendix Appendix D
# List of Figures

1.1 Deployment of NN models on edge devices .......................... 2
1.2 Overview of the thesis studies and methods. ...................... 9

2.1 Example of a convolution layer ...................................... 16
2.2 Different variations of an RNN layer. ............................. 20
2.3 Epiphany manycore architecture ..................................... 21
2.4 Example of a Pareto set for a problem that has two conflicting objectives .................................................. 25

4.1 Optimizing NN models to run on embedded platforms. ............ 32
4.2 Effective throughput of RNN implementations understudy [1]. 32
4.3 Energy efficiency of RNN implementations understudy [1] ....... 33
4.4 Streaming of CNN layers over a pipeline of cores. ................ 34
4.5 Estimated Performance of AlexNet and GoogLeNet for different precisions. ...................................................... 34
4.6 Quantization of the Integer Matrix to Vector ($M \times V$) multiplication in a recurrent layer gate ................................. 36
4.7 The potential of eliminated operations plotted vs. possible delta thresholds ......................................................... 37

5.1 Flexibility of implementations under study .......................... 40
5.2 Design flow of the CNN mapping method. ........................ 41
5.3 The Multi-Objective Hardware-Aware Quantization (MO-HAQ) method steps ...................................................... 44
5.4 Pareto optimal set for two objectives for Bit fusion architecture 45
5.5 The size of the compressed models at different error levels ..... 46
5.6 The number of operations in the compressed models at different error levels ......................................................... 46
List of Tables

1.1 Parts of research problem covered by each paper .......................... 13

2.1 The speedup and energy consumed by different types of low precision operations on the SiLago architecture .................. 22
1. Introduction

Health care, smart homes, and autonomous driving are just but a few application domains where Deep Learning (DL) has been used to support artificial intelligence [2–4]. DL’s ability to convert sensors’ raw data into high-level features is the reason behind its superior accuracy that can exceed humans [5]. For instance, autonomous cars are equipped with multiple sensors, such as radars, cameras, lidars, and microphones, that generate real-time loads of data. DL applications use inputs from the sensors to achieve the autonomous driving task via driver monitoring and speech recognition, object recognition and tracking, road users’ actions prediction, etc. However, the DL applications’ artificial intelligence comes at the cost of the requirement of enormous amounts of data and huge computing power.

The development of DL applications has two phases. The first phase is training, where an enormous amount of data is used to set the application parameters (weights) through multiple iterations [5]. Typically, training runs on the cloud. The second phase is called inference, where the trained DL application is used. The inference could run on the cloud or preferably on the edge near the sensors. For instance, in autonomous vehicles, sending cameras’ sensors’ outputs to be processed in the cloud has a high communication cost. In contrast, realizing the processing of the camera-sensors’ outputs in platforms embedded in the vehicle would save this communication cost and also enhance privacy and security. However, running DL applications on embedded platforms on edge comes at the cost of computational and memory resource limitations.

This thesis studies and proposes methods for the efficient realization of DL applications on embedded platforms. These methods work on the reduction of DL computational requirements with minor effects on their functional correctness (accuracy). The methods also address the hardware implementations of DL applications and the design requirements of DL domain-specific architectures. In addition, the thesis studies and suggests methods to cope with changes in the applications, the constraints, and the hardware platforms.
1.1 Motivation

The recent spread of deep learning models (deep neural networks) in many applications such as speech recognition [6], human activity recognition [7], and tracking [8] created an interest for supporting such models on edge devices [5]. However, Deep Learning (DL) models’ high computation complexity and memory requirements stand as an obstacle to realizing them on resource-limited embedded platforms on edge devices. Fortunately, DL models can be subjected to algorithmic optimizations to reduce their computational and memory demands without significantly affecting the models’ accuracy. For example, quantization [9] reduces the model parameters’ precision while pruning [10; 11] decreases the number of model parameters [1]. Algorithmic optimizations are usually applied during models’ training or require the retraining of the optimized models to retain the accuracy lost by these optimizations. Recently, post-training optimizations allowed skipping the retraining step by using more cautious methods during the optimization [12–14]. Post-training optimizations can be helpful in scenarios with no training data available, possibly for security reasons, lack of training time, or the lack of training servers.

![Diagram](image.png)

**Figure 1.1:** Deploying NN models on edge devices in their original forms leads to unsatisfying performance. To solve the problem, before deployment, algorithmic optimizations are applied to the models. Also, platform-specific optimizations are applied to embedded platforms.
Hardware-specific optimizations, such as pipelining and weight re-ordering, are applied to the embedded platform to enhance the performance of the implementation [1]. Therefore, by applying algorithmic optimizations and platform-specific optimizations, the efficient realization of such models on embedded platforms became possible, as seen in Figure 1.1. However, it is questionable if high efficiency and accuracy can be maintained if changes occur in the model’s layers or architecture, the application, or the target platform. In many research works, a model is selected, optimized, and implemented on a given platform without showing the flexibility of the solution to changes in the model or the application [15–17]. This thesis studies the flexibility support in implementations in literature and proposes methods for increasing the flexibility of solutions to changes in the model or the application.

Nevertheless, when we compress a given model using an optimization method $X$ and deploy it on a given platform $Y$, each layer in the model can be compressed using different compression levels. The total number of compression levels that can be applied to the model might be high. If we select only one of them, there is no guarantee that we will find the best optimization configuration for the model to run on platform $Y$. Also, each layer in the model has a different effect on energy savings and latency improvement on a given platform [18]. Thus, there is a growing research interest in incorporating the hardware platform in the process of optimizing the Neural Network (NN) in an automated manner (Hardware-Aware optimization) [18–21]. During the optimization, the degree of compression or approximation applied to each layer is determined to maximize the accuracy and hardware performance. Therefore, the model optimization techniques are flexible enough to provide an optimally modified model to run on different platforms under different application constraints.

It is better to consider the problem as a multi-objective problem as there exist two conflicting groups of objectives, the model’s accuracy, and the hardware efficiency objectives. However, most of the model compression/approximation techniques are time-consuming as they usually require model training. It is not possible to evaluate many candidate solutions and apply retraining for all of them. To overcome this problem, many hardware-aware optimization methods do the search for the optimum model compression configuration during training and provide us with one single solution. But, an automated method that provides the embedded system designer with a set of Pareto-optimal solutions gives the designer more freedom to trade-off between solutions [22]. Therefore, this thesis addresses the problem of multi-objective hardware-aware optimization of neural networks.

In this thesis, the proposed methods are applied to two kinds of NNs, Convolution Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).
CNNs are designed to work with image applications such as image classification [23] and image recognition [24]. CNNs rely on convolution layers to extract features from spatial inputs in the form of images or videos. On the other hand, RNNs are designed to work with applications that have sequential data inputs or outputs [25] by introducing feedback to capture the temporal relationship between sequences. Consequently, RNNs are frequently used for time-series applications such as speech recognition [6], language translation [26], and human activity recognition [7].

1.2 Challenges facing neural networks implementations on embedded platforms

From a computer architecture perspective, NNs are mainly composed of multiplications between high-precision values, non-linear functions, and derivation. The derivation is required during the back-propagation of the error to compute the weights update during the model training. On an embedded device, training is usually not required. The model is already trained, and all weight values are computed and ready to be used in inference. Therefore, inference on embedded platforms suffers mainly from multiplications between high-precision values and non-linear functions. Multiplications’ time and space complexity grow significantly in convolutions and matrix to vector multiplications. Therefore, computation and memory are two challenges facing the realization of NNs on resource-limited embedded platforms. To overcome these two challenges, optimizations are applied to models to decrease the model complexity. These modifications might have a negative effect on the model accuracy. Therefore, the accuracy is counted as a challenge facing the realization of the models on embedded platforms. In total, the three challenges are computation, memory, and accuracy challenges, as discussed in the next subsections.

1.2.1 Computation challenge

In CNNs, the convolutions take more than 90% of the computations [27]. For example, VGG-16 [28] has 15.3 GMAC (Giga Multiply And Accumulate) operations in the form of convolutions out of 15.5 GMAC total number of operations per one frame inference [29]. Even the hardware-friendly CNN named GoogleNet [30] has 1.43 GMAC operations. In RNNs, one LSTM (Long Short Term Memory) layer with 1024 hidden units and input of size 256 has 5.24 Mega MAC operations. More layers will consequently increase the number of operations in RNNs.

RNNs suffer from an extra problem that is not emerging in CNNs. The dependency of operations on the previous time-step output makes it difficult
to parallelize the computations of RNNs over multiple time steps. This dependency puts more constraints on the hardware implementations of such models.

### 1.2.2 Memory challenge

NNs require large memory space to store their weights. For example, VGG-16 requires 552 GB, and GoogleNet requires 5.72 GB to store the weights. The LSTM of 1024 hidden units and 256 input vector sizes would require 21 MB for one layer in the model. A deeper model would require even more. However, the on-chip memory (SRAM) size in an embedded platform is typically small, ranging from hundreds of kilobytes to a few megabytes [16; 29; 31–34]. Thus, it is not possible to store the model weights in the SRAM. The weights are stored in the off-chip memory. For example, in architecture with a DRAM connected to the NN accelerator via the system interconnect, the computation time increases due to the frequent access of the DRAM to load the weights. Consequently, the application is turned into a memory-bound application. In addition, DRAM communication is more energy-consuming than SRAM communication. For instance, reading one value from a DRAM can be over 100x more energy-consuming than reading the same value from the SRAM [29].

### 1.2.3 Accuracy challenge

To overcome the computation and the memory challenges, there is a need to decrease the NNs sizes and computation complexity. Thus, algorithmic optimization methods are applied to NNs. The compressed NNs might not be as accurate as the original models. Various techniques are applied to enhance the models’ accuracy after optimization, such as retraining the models for a few epochs after optimization [1]. In all cases, it is beneficial to define an application-related accuracy threshold that the optimized model accuracy must not go below. Different applications can have varying accuracy thresholds.

### 1.3 Thesis Objectives

This thesis studies and proposes novel methods for the implementation of deep learning models on embedded platforms. The studied and implemented methods are applied to fulfill two objectives. The first objective is implementation efficiency, and the second objective is flexibility.

#### 1.3.1 Implementation Efficiency

The term efficiency in computer science is defined as the ratio of useful work to resources allocated or the ratio of the output of a given system to its input.
put [35]. Efficiency can also be measured in terms of goals reached by using minimal amount of resources. To apply these definitions to the implementation efficiency of DL models on embedded platform, the inputs to the system are the DL models and the given platform resources. The outputs can be the throughput of the application on the given platform and the energy consumption.

This thesis considers implementation efficiency as the primary goal that is common in all the research work executing DL models on embedded systems and is quantified as follows:

- **Performance efficiency** in terms of throughput or delay thresholds. Throughput can be defined as the number of inputs/outputs processed per time unit. Throughput can be measured as the number of frames/sec in image applications, the number of predicted words/second in speech applications, and other possible units. A common unit for throughput that can be used to compare various applications is the number of operations per second. Applications that run in real-time are constrained by delay thresholds defined by the platform, sensors, system response time, and the application. It is not possible for a response to be delayed beyond the predefined delay threshold to preserve the functional correctness of the application.

- **Energy efficiency**. In embedded platforms, there is a need for keeping energy consumption low. However, the energy consumption constraint can vary between platforms and applications. For example, the energy consumption allowance in a wearable device is much lower than those in edge devices in cars and homes.

### 1.3.2 Flexibility

The definition of flexibility adopted in this thesis is "the quality of being able to change or be changed easily according to the situation," as in the Cambridge English dictionary.

There are various ways in which flexibility can be supported in the domain of DL applications. The design of DL models needs to be flexible to automatically support changes in the problem and the dataset, both of which might require a new model [36]. This flexibility can be achieved by using Neural Architecture Search (NAS) methods that automate the design of the optimal network topology [37; 38]. Another approach to add flexibility to DL models is transfer learning, a technique used to reuse a model trained for a specific task to cope with changes in the task without retraining the model from the beginning [39]. Yet another aspect of flexibility in connection to DL models is
the ability of the model to be resilient when algorithmic optimizations, such as compression, are used to fit the model on embedded platforms.

Also, hardware implementations need to support flexibility. In general, the choice of technology affects the degree of flexibility that the implementation can provide. This often leads to a trade-off with efficiency. Dedicated ASICs provide the lowest flexibility, reconfigurable FPGAs improve on this, and generic processors with software implementations are the most flexible. However, if we narrow the desired flexibility to be related to a specific domain, a better trade-off between efficiency and flexibility can be achieved.

This thesis focuses on the methods applied to realize the efficient implementation of DL models on embedded platforms. In this context, the input is a well-trained model. Therefore, the flexibility of the model design or training is not considered. Instead, the focus is on the flexibility of the optimizations applied to the model to realize its efficient implementation. Regarding hardware platforms, hardware implementations of DL models or components for putting together models should be able to accommodate changes in the model. In addition, the design of DL Domain Specific Architectures (DSAs) and accelerators should consider these changes. Designing an architecture for DL applications as a specific domain with known memory access patterns allows the enhancement of parallelism and the efficient use of the memory hierarchy [40].

This thesis discusses the flexibility objective from the following two perspectives:

- **Flexibility of the DL models.** In this thesis work, the input is a given pre-trained model that is subjected to algorithmic optimizations. Each layer in the model can be subjected to different possible optimization levels. The effect of different Neural Network (NN) layers’ optimization on latency improvement, energy savings, and accuracy drop is not static [18]. Considering the hardware platform and the application constraints during the optimization of DL models results in more optimally compressed models. However, changes in the hardware or the application make this compression solution not optimal anymore. Thus, to support flexibility in this context, the optimization of the DL models should be automated while considering the hardware architecture and the application constraints. Hardware architecture can be considered in many ways. The throughput and energy consumption of the implementations can represent the hardware during the optimization as objectives. Also, the implementation’s area or cost can be used as objectives.

- **Flexibility of the DL hardware implementations** on a given platform or accelerator. The implementation should be flexible enough to allow for changes in the layers, the number of layers, the types of layers, and
the optimization applied to different layers. Therefore, if a different model is to be used, the implementation stays possible and efficient. In addition, the architecture designed for deep learning should be flexible to support the algorithmic optimizations of DL models and support the variation in the degree of optimization. Otherwise, the optimization will not be as beneficial as intended. For example, deploying a quantized model on a platform that does not support quantization will not reach the maximum benefits of quantization. Also, deploying a model with a mixed-precision quantization configuration on a platform that supports only one precision configuration will limit the intended performance enhancements.

1.4 Research Questions

In this thesis, the focus is on the implementation of DL models on embedded platforms with limited resources. This thesis attempts to answer the following research questions:

RQ1a How to manage the high computation and memory demands of neural networks to achieve implementation efficiency on embedded platforms?

RQ1b How to enhance the flexibility of neural networks’ implementations on embedded platforms?

RQ2a How to apply multi-objective optimization to the problem of NN models’ mixed-precision quantization to consider both the model accuracy and hardware performance?

RQ2b How to optimize RNNs in post-training scenarios while preserving high accuracy?

1.5 Approach

This section presents the thesis approach to study and propose methods that satisfy the two thesis objectives as visualized in Figure 1.2.

1.5.1 Exploration of flexible methods for efficient realization of DL models on embedded platforms

As mentioned earlier, this thesis focuses on the implementation of NNs on embedded platforms. Figure 1.1 shows that a NN in its original form can-
not be efficiently deployed on embedded platforms. Thus, algorithmic optimizations are applied to the models to decrease their memory requirement and computation complexity. On the other hand, hardware-specific optimizations are applied to the platform to speed up the implementation and decrease its energy consumption. To answer RQ1a and RQ1b, I conducted a survey study exploring algorithmic optimizations and platform-specific optimizations and then analyzing the implementations that have applied those optimizations (Paper I [1]). The study highlighted optimizations and directions to achieve the thesis’s two main objectives. These optimizations and directions are as follows:

**Quantization (Implementation efficiency):**
Quantization is an algorithmic optimization method that is applied to NNs to decrease their size and computational complexity by reducing weights and activation precision. Low-precision quantization was ap-

---

**Figure 1.2:** Overview of the thesis studies and methods. All the thesis components focus on implementation efficiency and flexibility. The analysis of NN implementations selected promising optimizations tackled by the rest of the methods (in the blue ovals). $Ex$ is used to denote efficiency-related optimization $x$, $Fx$ is used to denote flexibility-related optimization $x$, and $EFx$ is used to denote efficiency and flexibility-related optimization $x$. 

- **Quantization**
- **On-chip Memory**
- **SRU**
- **Light & Compressed**
- **Hybrid Optimization**
- **Flexible Mapping**
- **Flexible Optimization**
plied by many of the most efficient implementations in the conducted study.

- **On-chip memory** (*Implementation efficiency)*:
  The analysis of implementation concluded that a memory-specific optimization has to be applied to achieve implementation efficiency. Using On-chip memory solely for storing the weights is one of the memory-specific optimizations used to avoid the frequent data loading from off-chip memory. Using on-chip memory saves a considerable amount of energy and converts the implementation into a compute-bound implementation. In the rest of the thesis work, the on-chip memory size has been considered a constraint for the model optimization problem.

- **Simple Recurrent Unit (SRU)** (*Implementation efficiency)*:
  SRU is used as an alternative for the most popular recurrent unit, which is the Long Short Term Memory (LSTM). The LSTM suffers from using the previous time step vectors in the matrix to vector multiplications. Matrix to vector multiplications are the bottleneck in the RNN computations. The parallelization of these computations over multiple time steps is a desirable optimization that cannot be achieved with the LSTM. SRU solves this problem by removing the previous time step vectors from the matrix to vector multiplications, and thus, their parallelization over time steps becomes possible.

- **Flexible mapping** (*Flexibility)*:
  In many of the implementations understudy in the survey article (Paper I), only one model is mapped into the hardware platform. There is a need to show how the mapping will change if changes happen to the model. These changes can be in the number of layers, in the size of layers, and in the optimization method applied to the layers.

- **Hardware-aware optimization** (*Flexibility)*:
  The model optimization that considers the hardware platform and the application constraints is a topic with growing research interest. Application constraints can be, for example, accuracy, throughput, or energy consumption constraints.

- **Hybrid Optimization** (*Implementation efficiency/Flexibility)*:
  The analysis of implementations (survey study) identified that applying an algorithmic optimization method to the model is essential to achieve high efficiency. There are three types of optimization methods that reduce the model requirements in different ways. Quantization reduces
the precision of weights and activation vectors, compression methods reduce the number of parameters in the model, and the delta-networks method eliminates the operations for redundant parts in the input vector. Mixing these optimizations with a tuning for the degree of optimizations will lead to higher efficiency and better opportunity for flexible optimization that can provide a different optimization configuration for different applications and hardware platforms.

• **Light vs Compressed layers** (*Implementation efficiency/Flexibility*):
  The survey study analyzed the recurrent layer alternatives designed to be light and consume less resources. It also analyzed the algorithmic optimization methods applied to the recurrent layers for the same purpose. A question is raised here whether applying algorithmic optimizations to the large recurrent layers or their light alternatives is more beneficial.

1.5.2 Proposed flexible methods for efficient realization of DL models on embedded platforms

The optimizations and directions highlighted by the survey study are applied in three methods as visualized in Figure 1.2. These three methods are:

• **Streaming Tiles** is a method for **Flexible mapping** of quantized CNNs on manycore architectures (RQ1b) [41]. The method achieves implementation efficiency (RQ1a) by mapping **Quantized** CNNs on a pipeline of Epiphany cores and using the cores’ on-chip memory for weight storage. The method increases the flexibility of the mapping by supporting two CNNs, two on-chip memory sizes, and multiple quantization precisions.

• **MOHAQ (Multi-Objective Hardware-Aware Quantization)** is a method for **Flexible** (RQ1b) **Quantization** of SRU-based RNNs to preserve high implementation efficiency (RQ1a) on different hardware platforms and under different application constraints [42]. A multi-objective search method is proposed to select the quantization configurations of the model (RQ2a). The memory size is considered as a constraint for quantization to store all the weights in the on-chip memory.

• **Shrink and Eliminate** is a method that applies **Hybrid optimizations** (**Quantization** and Delta method) on four RNN (LSTM-based, GRU-based, LiGRU-based, SRU-based) models to maximize their implementation efficiency (RQ1a) in post-training scenarios (RQ2b). The method compares the optimized models under different on-chip memory sizes and error constraints to support the **Flexibility of optimization** (RQ1b).
1.6 Contributions

The contributions of this thesis are summarized as:

- Highlighting the most promising directions for efficient and flexible implementations of RNNs on embedded platforms (Paper I) to answer RQ1a and RQ1b. These directions were concluded by conducting a study that analyzed the computational demands of different RNN layers, algorithmic and platform-specific optimizations, and then compared the state-of-the-art RNN implementations. The conducted study focuses on how different optimizations reduce the memory and computation requirements of the models while preserving high accuracy to address the thesis challenges.

- Proposing a flexible and efficient mapping of CNNs on the Epiphany manycore architecture by allowing variable precision with connection to RQ1a and RQ1b (Paper II). Quantized CNNs are selected for implementation as they have reduced computation and memory requirements. Pipelining and on-chip memory storage are used to decrease the memory requirements of the implementation further.

- Enabling the fast evaluation of candidate solutions for multi-objective optimization for bit-width selection for NN layers on multiple hardware models while considering the retraining effect (MOHAQ) with connection to RQ2a (Paper III). MOHAQ is a flexible compression method (RQ1b) that finds quantization solutions that minimize the model size and operations precision while preserving high accuracy to overcome the memory, computation, and accuracy challenges and to achieve implementation efficiency (RQ1a).

- Proposing a method for applying post-training quantization together with the delta networks method on four RNNs with connection to RQ1a, RQ1b, and RQ2b (Paper IV). The method addresses the thesis’s three challenges by decreasing the size of the model and the number and precision of operations while preserving high accuracy.

Table 1.1 summarizes the research questions covered by each paper, the type of the DL model under study, and the underlying platform.

1.7 Thesis Structure

The rest of this thesis summary is organized as follows. Chapter 2 gives a brief background of NN layers, hardware architectures, and optimization methods.
Table 1.1: Parts of research problem covered by each paper

<table>
<thead>
<tr>
<th>Paper</th>
<th>Questions</th>
<th>Models</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>I [1]</td>
<td>RQ1a &amp; RQ1b</td>
<td>RNN, RNN+CNN</td>
<td>ASIC, FPGAs, Others</td>
</tr>
<tr>
<td>II [41]</td>
<td>RQ1a &amp; RQ1b</td>
<td>CNN</td>
<td>Manycores</td>
</tr>
<tr>
<td>III [42]</td>
<td>RQ1a, RQ1b &amp; RQ2a</td>
<td>RNN</td>
<td>Silago [43], Bitfusion [31]</td>
</tr>
<tr>
<td>IV [44]</td>
<td>RQ1a, RQ1b &amp; RQ2b</td>
<td>RNN</td>
<td>-</td>
</tr>
</tbody>
</table>

used in this thesis. Next, Chapter 3 presents related research works. Then, Chapter 4 discusses the methods applied to achieve the first objective, "implementation efficiency", while Chapter 5 discusses the methods applied to achieve the second objective, "flexibility". Next, the contributions of each individual paper are presented in Chapter 6 followed by the thesis discussion and conclusions in Chapter 7 and Chapter 8, respectively.
2. Background

This chapter briefly presents the NN models, hardware architectures, and optimization methods used in this thesis.

2.1 Neural Networks Layers

This section covers the NN layers used in models under study in this thesis. First, the convolution layer used in CNN and RNN models is presented. Then, the recurrent layers used in RNN models are explained. Finally, the Fully Connected (FC) layer found in all the used models is presented.

2.1.1 Convolution layer

Convolution layers are used to process spatial inputs. CNN models mainly rely on stacking multiple convolution layers [23; 30]. Also, RNN uses some convolution layers as a feature extractor if the input is in the form of an image [7; 45], video [46], or spectrogram [16; 47]. The input to a 2D convolution layer is a channel of 2D matrices of size $N$. The output of the layer is a channel of 2D matrices of size $M$. In the layers, filters of size $K \times K \times N \times M$ are convoluted with the input. There is a 2D filter of size $K \times K$ corresponding to each input matrix in the input channel to produce each output matrix in the output channel.

Figure 2.1 shows an example of a convolution layer that has input channel size 3 and output channel size 5. Thus, in total, there are 15 filters ($3 \times 5$). The input channels matrices size is $R_I \times C_I$, and the output channel matrices size is $R_O \times C_O$. The output channel matrices size relies on the input, filter, and convolution stride size.

2.1.2 Recurrent layer

Recurrent layers are added to NN models to capture the temporal relation between input and output. In contrast to all other NN layers, the recurrent layers take two inputs: the previous layer output and the previous-time step output of the current layer. The most famous recurrent layer is the Long Short Term
Memory (LSTM) [48]. Later, alternative recurrent layers were proposed to either enhance the functionality of the LSTM or decrease its computational and memory demands. This thesis focuses on embedded systems implementations and thus focuses on LSTM alternatives that have less computational needs, such as GRU [49], LiGRU [50], and SRU [51]. The LSTM and the alternative recurrent layers are described below.

1. Long Short Term Memory (LSTM)

LSTM has two inputs, input vector $x_t$ and the previous time-step output vector $h_{t-1}$. It has three gates: input, forget, and output gate, and one memory state vector. The forget and the input gates control what will be forgotten and what will be updated in the memory state, and the output gate decides which part of the state will be in the output vector. From a computer architecture perspective, the state vector computations are similar to the three vector computations. Thus, I consider the LSTM to have four computational blocks, as visualized in Figure 2.2a.

Each computational block is a Matrix to Vector ($M \times V$) multiplication between a weight matrix with the combination of the input vector and the previous time-step output, addition of the result to a bias vector, and then application of a non-linear function to the result. There might be more element-wise operations in the computational blocks. The LSTM four computation blocks are as follows:
• **Forget gate:** The forget gate vector $f_t$ is computed as

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f),$$

(2.1)

where $x_t$ is the input vector, $h_{t-1}$ is the hidden state output vector, $W_f$ is the weight matrix, $b_f$ is the bias vector, and $\sigma$ is the sigmoid function.

• **Input gate:** The input gate vector $i_t$ is calculated as

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i),$$

(2.2)

using the weight matrix $W_i$ and the bias vector $b_i$.

• **State computation:** The computation of the new memory state vector $C_t$ starts with the computation of the candidate vector of the new state vector

$$\tilde{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c),$$

(2.3)

where $x_t$ is the input vector, $h_{t-1}$ is the hidden state output vector, $W_c$ is the weight matrix, and $b_c$ is the bias vector. Finally, the new memory state vector $C_t$ is computed as

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t,$$

(2.4)

where $C_{t-1}$ is the previous time-step state vector, $f - t$ is the forget gate vector, $i_t$ is the input gate vector, and $\odot$ is used to denote the element-wise multiplication.

• **Output gate:** The output gate vector $o_t$ is computed as

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o),$$

(2.5)

where $x_t$ is the input vector, $h_{t-1}$ is the hidden state output vector, $W_o$ is the weight matrix, $b_o$ is the bias vector, and $\sigma$ is the sigmoid function.

Then, the hidden state output vector $h_t$ (LSTM output) is the result of the element-wise multiplication with the output gate vector $o_t$ to the $tanh$ of the state vector $C_t$ as

$$h_t = o_t \odot \tanh(C_t).$$

(2.6)

2. Gated Recurrent Unit (GRU)

The GRU (Figure 2.2b) was proposed to enable the recurrent layer to capture dependencies in an adaptive manner at different time scales [52]. The design of the GRU has only two gates and a memory state (three computational blocks), making it smaller in size than LSTM. The three computational blocks are as follows:
• **Reset gate** The reset gate decides whether the input is the first symbol in the sequence or not. It is computed as

\[ r_t = \sigma(W_r[h_{t-1}, x_t]), \]  

(2.7)

where \( x_t \) is the input vector, \( h_{t-1} \) is the hidden state output vector, \( W_r \) is the weight matrix, and \( \sigma \) is the sigmoid function.

• **Update gate** The update gate determines which parts of the output will be updated. It is computed as

\[ z_t = \sigma(W_z[h_{t-1}, x_t]), \]  

(2.8)

where \( W_z \) is the weight matrix.

• **Output computation** To compute the GRU output vector \( h_t \), the output candidate vector is computed first as

\[ \tilde{h}_t = \tanh(W[r_t \odot h_{t-1}, x_t]), \]  

(2.9)

where \( x_t \) is the input vector, \( h_{t-1} \) is the hidden state output vector, and \( W \) is the weight matrix.

Then, the hidden state vector is computed as

\[ h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t, \]  

(2.10)

where \( z_t \) is the update gate vector and \( h_{t-1} \) is the previous time-step output.

3. **Light Gated Recurrent Unit (LiGRU)**

The GRU was modified to produce a lighter version called Light GRU (LiGRU) [50] by removing the reset gate (Figure 2.2c). The LiGRU has only two computation blocks, the update gate, and the output vector computation. The output gate candidate vector computations are modified to use the RELU (rectifier linear unit) activation function instead of the tanh function.

4. **Simple Recurrent Unit (SRU)**

The Simple Recurrent Unit (SRU) [51] was proposed to make the recurrent unit friendlier for parallelization. In LSTM and GRU, the computational bottleneck is the matrix to vector multiplications. The parallelization of this part is difficult as it depends on the previous time-step output \( h_{t-1} \) and state vector \( C_{t-1} \). The SRU removes previous time-step vectors from all matrix to vector multiplications and uses them only in element-wise operations.
The SRU has two gates and a memory state (three computational blocks), as seen in Figure 2.2d. The SRU gates (update and forget gates) are computed as

\[ f_t = \sigma(W_f x_t + v_f \odot c_{t-1} + b_f) \quad (2.11) \]

and

\[ r_t = \sigma(W_r x_t + v_r \odot c_{t-1} + b_r), \quad (2.12) \]

respectively. The previous time-step state vector \( C_{t-1} \) is element-wise multiplied with the parameter vectors \( v_f \) and \( v_r \) that are learned with weight matrices and biases during training.

The state vector \( C_t \) is computed as

\[ C_t = f_t \odot C_{t-1} + (1 - f_t) \odot (W_x t), \quad (2.13) \]

where \( C_{t-1} \) is the old state vector, \( x_t \) is the input vector, and \( f_t \) is the forget gate output vector. The SRU output vector \( h_t \) is computed as

\[ h_t = r_t \odot C_t + (1 - r_t) \odot x_t, \quad (2.14) \]

where, \( r_t \) is the update gate vector, \( x_t \) is the input vector, and \( c_t \) is the state vector.

**Bidirectional recurrent layers:** The recurrent layer input can be fed in two directions (past to future and future to past). In this case, the layer is called a bidirectional layer. The processing of the input in two directions temporally enhances the layer’s understanding of the data context. The bi-direction concept can be applied to all previously discussed layers such as BiLSTM [53] and BiGRU [54].

### 2.1.3 Fully connected (FC) layer

In the Fully connected layer, each input neuron is connected to each output neuron by a weight. This connection happens in the form of a matrix to vector multiplication of a weight matrix with the input vector. In the models used in this thesis, the FC layer is used to prepare the NN for the output function, for example, a Softmax, by changing its dimension from the recurrent layer output vector dimension to the output function dimension.

### 2.1.4 Projection Layer

The projection layer is a layer similar to the FC layer added before or after the recurrent layer. It is mainly used to decrease the size of the recurrent layer input vector size. Thus, increasing the number of the recurrent layer’s hidden cells becomes possible while keeping the layer’s total number of weights low.
Figure 2.2: Different variations of an RNN layer.
2.2 Architectures under study

In this thesis, three hardware architectures have been used in the proposed methods. The first is Epiphany, a manycore architecture [55]. The second is SiLago [43], a coarse-grained reconfigurable architecture. The third is Bit fusion [31], a systolic array architecture.

2.2.1 Epiphany architecture

Epiphany is a low-power manycore architecture [55]. It is mainly composed of a 2D mesh of RISC cores. The 2D mesh has 16, 64, and 1024 cores in Epiphany-III, Epiphany-IV, and Epiphany-V, respectively. Each core has a local memory of 32KB in Epiphany III and Epiphany IV, and 64 KB in Epiphany V. A low latency eMesh network-on-chip is used to connect the cores. The eMesh has three separate networks. The networks are for reading requests, writing on-chip, and writing off-chip. The Writing on-chip network is 16x faster than the reading on-chip network. Epiphany is considered in this work due to its superiority in energy efficiency. The architecture energy efficiency reaches 70 GFLOPS/watt in Epiphany IV and 75 GFLOPS/watt in Epiphany V. Figure 2.3 shows the Epiphany architecture and its core components.

![Figure 2.3: Epiphany manycore architecture [55]](image)

2.2.2 SiLago architecture

SiLago is a customized CGRA architecture [43]. It is built using two fabrics. The first fabric is for computations and is called Dynamically Reconfigurable Resource Array (DRRA). The second fabric is used as a variable-size streaming scratchpad memory and is called Distributed Memory Architecture (DiMArch). Silago supports three different precision operations, 16-bit, 8-bit,
and 4-bit operations. However, the two operands have to be with the same precision, as SiLago does not support mixed-precision operations.

Table 2.1 summarizes the speedup and the energy consumption of the arithmetic operations in the SiLago platform. The energy consumption is calculated using the reconfigurable multiplier and accumulator (MAC) post-layout simulations synthesized using a 28nm technology node. The energy consumption for the SRAM access was based on 28nm technology macro-generated tables. The speedup of different precisions operations over 16-bit operations is calculated as operations per clock cycle.

Table 2.1: The speedup and energy consumed by different types of low precision operations on the SiLago architecture.

<table>
<thead>
<tr>
<th>Operation</th>
<th>16x16</th>
<th>8x8</th>
<th>4x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC speedup</td>
<td>1x</td>
<td>2x</td>
<td>4x</td>
</tr>
<tr>
<td>MAC energy cost (pJ)</td>
<td>1.666</td>
<td>0.542</td>
<td>0.153</td>
</tr>
<tr>
<td>Loading 1-bit energy</td>
<td>0.08</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.2.3 Bit fusion architecture

Bit fusion is a systolic architecture designed to support variable precision operations required by neural networks [31]. The 2-D systolic array connects what is named Fused Processing Elements (Fused-PE). Each Fused-PE consists of 16 bit-brick. The bit-brick is a processing element capable of doing 1-bit or 2-bit MAC (Multiply And Accumulate) operations. Higher precisions MAC operations are supported by grouping the bit-bricks inside the fused-PE. To support 8-bit operations, all the fused-PE bit-bricks are connected. And, to support 16-bit operations, the fused PEs are used for four cycles. Thus, the speedup gained by using 2-bit operations over 16-bit operations is 64x.

2.3 Algorithmic Optimizations

In this section, I discuss the algorithmic optimizations applied to NN models in this thesis. These methods are applied to decrease the NN models’ computational complexity and memory requirement.

2.3.1 Quantization

Quantization decreases the NN model size and computation complexity by reducing the precision of its weights and activations. The precision can be reduced to a 16-bit fixed-point or to integer precisions from 8-bit to 1-bit. Quan-
zation to a 16-bit fixed-point usually does not affect the inference accuracy. Consequently, many accelerators rely on 16-bit fixed-point operations [56]. Integer quantization provides a more feasible realization on embedded platforms [14]. However, integer quantization causes a degradation in the model accuracy. The accuracy degradation can be retained by different techniques. The first technique is quantization-aware training, where the quantization is applied during the training of the model from the beginning. Integer weights are used in the forward computation, and the gradient computations and full precision weights are used in the backpropagation. Thus, the resulting trained model has full precision parameters that, when quantized, can keep high accuracy. The second technique is similar to the first technique. However, quantization is applied to a pre-trained model, and then quantization-aware training is applied only for a few epochs as a retraining step.

The third technique is called post-training quantization, where quantization is applied to a pre-trained model without any further retraining epochs. However, the quantization method itself is very cautious not to cause a big degradation in accuracy. Post-training quantization methods focus on the outlier values that consume the allowed precision. Applying clipping methods on the data ranges and saturating the outlier values narrow to the data range helps in overcoming the problem. Several methods have been used for the clipping threshold selection [12; 13] and showed a preservative effect on models’ accuracy levels. An alternative solution to clipping is the Outlier Channel Splitting (OCS) method. This method duplicates the outlier values of channels and halves the output values or their outgoing weights to get rid of the outlier values and preserve the functional correctness [12]. Unfortunately, the duplication increases the model’s size, and thus, the model consumes more memory. In this thesis, clipping is used during the post-training quantization, and the Minimum Mean Square Error (MMSE) method is employed for the clipping threshold selection [57].

2.3.2 Delta networks method

The Delta networks method is a technique that takes advantage of the temporal relation between the input data sequences to decrease the number of computations and memory accesses [58]. Successive input vectors have a high percentage of similar or close to similar values. For similar values in the successive input vectors, it is unnecessary to repeat the computations. Thus, the method first restructure how the $M \times V$ is carried out to reuse the previously computed values if the input values are similar. This change does not alter the function of the computations and does not affect the accuracy. The $M \times V$ equation is
initially defined as

$$MV_t = W_x x_t + W_h h_{t-1}, \quad (2.15)$$

where $MV_t$ is the $M \times V$ multiplication result in the current time-step, $W_x$ is the input weight matrix, $x_t$ is the input vector, $W_h$ is the recurrent weight matrix, and $h_{t-1}$ is the hidden state vector. The equation is modified to be

$$MV_t = W_x \Delta x + W_h \Delta h + MV_{t-1}, \quad (2.16)$$

where $\Delta x = x_t - x_{t-1}$, $\Delta h = h_{t-1} - h_{t-2}$, and $MV_{t-1}$ is the $M \times V$ multiplication result in the previous time-step. Later, an approximation is applied to eliminate computations corresponding to values in the input that are close to similar. Thus, every two values in the successive vectors will be compared, and if the difference is less than a delta threshold $\Theta$, computations will be eliminated, and the result from previous computations will be used. However, errors may accumulate over time-steps if the comparison is handled between two-time steps only. As an example, if the vector values increase every time step by values close to $\Theta$, the delta networks method will keep applying the approximation, and accuracy will be harshly affected. As a solution, memory states are proposed to store the last values that caused a change in the delta vector. The state vector values are computed as

$$\hat{x}_{i,t-1} = \begin{cases} x_{i,t-1} & \text{if } |x_{i,t-1} - \hat{x}_{i,t-2}| > \Theta, \\ \hat{x}_{i,t-2} & \text{otherwise,} \end{cases}$$

$$\hat{h}_{i,t-1} = \begin{cases} h_{i,t-1} & \text{if } |h_{i,t} - \hat{h}_{i,t-2}| > \Theta, \\ \hat{h}_{i,t-2} & \text{otherwise,} \end{cases} \quad (2.17)$$

where $\hat{x}_{i,t-1}$ and $\hat{h}_{i,t-1}$ are the states storing the last change of the element $i$ in the input and hidden state vectors, respectively. The delta vectors $\Delta x_{i,t}$ and $\Delta h_{i,t}$ are computed by comparing the current input and hidden state vectors with the state vectors. For each pair of elements $i$, if their absolute difference value is less than $\Theta$ value, the $i$ element is changed to zero in the delta vector, as follows:

$$\Delta x_{i,t} = \begin{cases} x_{i,t} - \hat{x}_{i,t} & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta, \\ 0 & \text{otherwise,} \end{cases}$$

$$\Delta h_{i,t} = \begin{cases} h_{i,t} - \hat{h}_{i,t} & \text{if } |h_{i,t} - \hat{h}_{i,t-1}| > \Theta, \\ 0 & \text{otherwise,} \end{cases} \quad (2.18)$$

### 2.4 Multi-objective optimization

In engineering and economic problems, whenever decision making is needed, optimization appears [59]. Optimization methods try to find the best choice
(solution) from multiple alternatives quantified by an objective function and bounded by constraints. In many cases, the problem has multiple conflicting objectives, and it is not enough to provide one solution as Single Objective Optimization (SOOP) does [22]. In such cases, there might not exist one optimum solution that optimizes all the objectives at the same time. Instead, there exists a set of non-dominated solutions where none of them can be enhanced by any other solution. For example, if a solution $S_1$ can enhance one objective in solution $S_2$ without making any other objective get worse, then $S_1$ dominates $S_2$ and replaces it in the non-dominating solution set. The non-dominating solution set is called the Pareto set and is visualized in Figure 2.4. Multi-Objective Optimization is the optimization method that can produce a Pareto-set for multi-objective problems.

![Figure 2.4: Example of a Pareto set for a problem that has two conflicting objectives. The two objectives have to be minimized.](image)

Genetic Algorithms (GAs) are optimization algorithms used for single and multi-objective optimization [60]. In GAs, survival is for the fittest, as inspired by natural selection. GAs are based on populations, where each population is a group of candidate individuals (solutions). The individual is a vector of values for the variables that need to be optimized, named chromosomes. The algorithm runs in an iterative manner. In each iteration, the current population’s solutions’ fitness values are evaluated, and then a new population (offsprings) is generated from the old population. The algorithm stops when a predefined criterion is met, or the required number of populations is completed. The offspring population is composed by selecting pairs of good solutions from the old population based on their evaluated fitness values and applying crossover and mutation to them. Crossover is a method that composes new individuals by mixing parts of old (parent) individuals. Mutation is a method that changes
some chromosomes in the offspring individuals. The GA keeps repeating the selection, crossover, and mutation until it completes the new population. Genetic operations (crossover and mutation) require the solutions to be in some representations that might not be the same used in the objective function evaluation. Thus, encoding and decoding functions are used to switch the solutions variables’ values between different representations. There are various GAs designed for multi-objective problems, such as NPGA [61], NSGA [62], and NSGA-II [63], which have been used in this thesis.
3. Related Work

This chapter discusses the work related to this thesis. First, it introduces the relevant work that applies optimization methods to realize the efficient implementation of DL models. Then, it presents some architectures that are relatively flexible DL architectures, and it discusses the related work in the domain of hardware-aware optimization of NN models. Finally, it highlights some research gaps in the literature.

3.1 Optimizations applied to DL implementations

Post-training quantization has been applied to CNNs for image applications [12; 13; 64–67] and RNNs for text/language applications [12; 68]. Ban-
ner et al. combined their proposed clipping method (ACIQ) with a bit channel allocation policy and bias correction to minimize the accuracy loss [64]. They carried out experiments on six ImageNet models using 4/8-bit for weights and activations. Zhao et al. updated the Net2WiderNet [69] layers transformations method to reduce the quantization error by Outlier Channel Splitting (OCS) [12]. OCS reduces the outlier neurons’ magnitude by duplicating them and halving the neurons’ output values or their outgoing weights. OCS was used solely or combined with clipping methods. They quantized the models to 8, 6, and 4-bit while keeping the accuracy high for ImageNet and WikiText-2 models.

Some work has been done to apply the delta networks method to RNN models and design delta-supported accelerators. Gao et al. designed a one layer GRU FPGA 16-bit accelerator [15]. The model was trained using the same delta threshold applied during inference using the TIDIGIT dataset [70]. They reached up to 5.7x speedup with a negligible error increase. EdgeDRNN is another delta-supported accelerator for GRU-based models [71] using an 8-bit integer for the weights and 16-bit fixed-point activations. A 2-layer GRU-based model was trained while applying the delta network method on the TIGIT dataset. The word error rate increased slightly from 0.77 to 1.3 for the arithmetic units to compute ten times faster. Jo et al. proposed an accelerator that uses 8-bit integer MAC operations and applies the delta network method [72]. They reduced the operations by 55% on an LSTM model on the
Librispeech dataset [73] with only a 1% increase in error.

Many implementations applied both algorithmic optimizations and memory-specific optimizations to achieve high efficiency. The on-chip memory has been used for storing the weights in most of these implementations [15; 16; 34; 56; 74]. An alternative memory-specific optimization is the compute-load overlap used in ESE accelerator [33]. The architecture had a scheduler that overlapped computation with memory accesses to eliminate the off-chip memory access overhead. Computation in memory is also a promising memory-specific optimization [75; 76]. The processing in memory can be supported by quantization to replace the MAC operation with XNOR and bit-counting operations [75]. Then, it would be sufficient to use an XNOR-RRAM based architecture.

3.2 Flexibility in deep learning implementations on embedded platforms

3.2.1 Flexible domain specific architectures

The design in MAERI [77] employed a configurable interconnection network topology to increase the flexibility of the accelerator. The accelerator in MAERI [77] supports both LSTM and CNN layers. The accelerator also supports both sparse and dense matrices. Sparse-TPU changed the design of the PEs in the TPU systolic array architecture to support the column packing algorithm used to handle sparsity in pruned matrices [78]. Interestingly, Sparten and Eyeriss V2 accelerators handle the sparsity in both of the activation vectors and the weights, not only the weights [79; 80]. The design in Bit fusion [31] supports varying precision models by allowing dynamic precision per layer for both CNN and RNN models. Similarly, the Microsoft NPU brainwave architecture [81] supports varying precision using a narrow precision block floating-point format [82]. To maximize the benefit of varying precision, an FPGA implementation [74] applied a parameterizable parallelization scheme. When lower precision is required, LSTM units are duplicated to exploit the unused resources to gain speedup. When higher precision is used, SIMD folding is applied to save resources for the needed high precision.

In addition, PIM (Processing In Memory) technology has shown recent advances in the domain of flexible DL architectures. Compute-SRAM is an architecture that supports varying precision for both weights and activations [83]. Also, Z-PIM supports sparsity in NNs in addition to the varying precision [84]. However, only varying weights precision are allowed, and the activations precision is fixed as a 16-bit fixed-point.
3.2.2 Hardware-aware optimization

Energy-aware pruning [18] is a pruning method that minimizes energy consumption on a given platform. The platform model was used to guide the pruning process by informing it which layer, when pruned, would lead to more energy saving. The pruning process stops when a predefined accuracy constraint has been hit. Netadapt [85] eliminated the need for platform models by using direct empirical measurements. Nevertheless, pruning in Netadapt is constrained by a resource budget such as latency, memory size, and energy consumption. Similarly, an energy-constrained compression method [21] used pruning guided by energy constraints. Energy results are predicted from a mathematical model for a TPU-like systolic array structure architecture. However, this compression method trains the model from the beginning. On the other hand, DeepIOT [19] obtains the memory size information from the target platform to compute the required compression ratio as a constraint.

HAQ (Hardware-aware Quantization) used reinforcement learning to select bit-width for weights and activations to quantize a model during training while considering hardware constraints [20]. Optimization variables are selected based on time constraints [86], where roof-line models are used for calculating the maximum achievable performance for different pruning configurations.

EMOMC (Evolutionary Multi-Objective Model Compression) [87], and HW_FlowQ [88] are methods that apply multi-objective optimization to search for the NN model compression parameters while having the hardware platform in the loop. They have different approaches to overcoming the difficulty of applying retraining during the search. EMOMC retrained 100 models using 100 different degrees of pruning and saved them in a library [87]. Later, during the search, the candidate solution corresponding model is loaded and used for evaluation. In addition, the EMOMC method searched for quantization levels for different layers during the search. However, no retraining has been applied to quantized solutions. HW_FlowQ used the pre-trained parameters as a base for quantization during the candidate solutions evaluation [88]. For each solution, fine-tuning is applied to retain the lost accuracy. By the end of the search, all of the solutions in the Pareto-set are fully trained, as the evaluated accuracy during the search was only an estimation. Wang et al. had a different approach to account for retraining in their APQ method [89]. They created an accuracy predictor by training a neural network model to predict the accuracy for different quantization configurations.

In this thesis, quantization is used as a compression method, and target hardware models are used to guide the compression. Both the model error/accuracy and hardware performance metrics (speedup and energy consumption) are allowed to be objectives. The hardware on-chip memory size
is used as a constraint to avoid high-cost off-chip communication. The effect of retraining on the quantized solutions has been considered during the multi-objective search.

3.3 Research Gaps

Post-training quantization methods have been mainly applied to CNN models more than RNN models. None of the discussed literature applied post-training quantization on SRU, GRU, or LiGRU units. Most of the work focus on CNNs and a few LSTM-based RNNs. In addition, no one addressed the effect of applying the delta network method to integer quantized RNNs. Also, the selection of the delta threshold in all of the previous work used to be heuristic, and no information could be predicted about how good or bad different thresholds are when applied to various models. Thus, this thesis focuses more on RNN models with different types of recurrent layers. The thesis also studies the effect of applying quantization and the delta-networks method together.

Despite the recent advances in flexible architectures for DL applications, there is still more work that needs to be done in the field. The proposed flexible architectures have shown the ability to run several DL model types and sizes and support some variation in algorithmic optimizations. Most of them either support the sparsity or the precision variations, but very few support both. In addition, the delta network method, which can eliminate a high percentage of operations, does not get enough attention in these architectures. Online learning, which gives the flexibility of updating the NN parameters on edge, is still not supported enough.

Many hardware-aware algorithmic optimization methods in literature cannot apply a multi-objective search and consider the retraining effect at the same time. The few who addressed this problem circumvented the necessity to retrain many solutions during the search either by decreasing the number of solutions [87], decreasing the amount of retraining per solution [88], or by using accuracy predictors [89]. This thesis has a different approach by relying on post-training quantization and proposing a novel method called "beacon-based" search.

This chapter presents the approach applied to study and propose methods for the efficient realization of deep learning models on embedded platforms. First, a study of the RNN implementations on embedded platforms is conducted to define the most promising optimizations that can be applied to achieve highly efficient implementations. Then, the chapter summarizes the proposed methods that apply these optimizations to DL models.

4.1 Study of the efficient implementations of DL models on embedded platforms

Neural Network models rely on a large number of high-precision weights and operations. Thus, deploying them directly on limited resources embedded platforms will give poor-performing solutions. As seen in Figure 4.1, algorithmic optimizations are applied to the models to reduce their size, and computation complexity and hardware-specific optimizations are applied to the hardware platforms to increase the implementation efficiency.

In the RNN implementations survey [1], I compared the hardware implementations of RNN models in terms of throughput and energy efficiency to find out which optimization methods are superior in achieving high efficiency [1]. Figure 4.2 shows a plot of the implementations’ throughput against their indices. Implementations are divided into ASIC, FPGA, and other types of hardware implementations. More details about the implementations and the optimization methods applied to them are in the survey article (Appendix 1). Figure 4.3 shows a plot of the implementations’ energy efficiency against their indices.
Figure 4.1: Two kinds of optimization applied to NN implementations. The first is algorithmic optimizations applied directly to the models. The second is hardware-specific optimizations applied to the hardware platform.

Figure 4.2: Effective throughput of RNN implementations understudy [1].

The comparison led to the following conclusions:

- It is vital to apply an algorithmic optimization method to the NN model to reach high throughput and low energy consumption.
- It is necessary to apply a memory-access optimization that hides or decreases the memory accesses to reach high throughput and low energy consumption.
• Processing in memory and analog computing is very promising in terms of improving throughput and energy efficiency.

• The optimizations applied to the top efficient implementations among the compared implementations are low precision quantization, structured matrices (block-circulant and circulant matrices), and the delta networks method.

Figure 4.3: Energy efficiency of RNN implementations understudy [1].

4.2 Proposed methods for efficient implementations of DL models on embedded platforms

This section explains two methods proposed to achieve high efficiency in NNs’ implementations. The two methods focus on quantized NN models. The first method maps quantized CNN models to the Epiphany manycore architecture. The second method applies quantization in addition to the delta networks method to RNN models.

4.2.1 Mapping of CNN convolution to the Epiphany manycore architecture

This section presents a method of mapping CNN convolution modules to the Epiphany manycore architecture. The CNN models are assumed to be quantized, but the method can be applied to non-quantized models as well. To increase the implementation efficiency, the method relies on two techniques.
The first one is to pipeline the CNN layers’ implementation over groups of cores. Each group is assigned a layer, as seen in Figure 4.4. This technique eliminates the need for storing intermediate results between NN layers in the off-chip memory. The second approach is to store all the weights in the cores’ on-chip memories to eliminate the need for loading the weights from the off-chip memory. Thus, the minimum number of cores required for the implementation is determined by the model size and precision.

Figure 4.4: Streaming of CNN layers over a pipeline of cores.

Figure 4.5 shows the expected throughput in terms of Frames Per Second (FPS) for AlexNet and GoogLeNet for different precisions using Epiphany cores with 64 KB memory size. The x-axis shows the minimum number of cores required to store the weights, and the y-axis shows the corresponding throughput. Lower precision implementations require less memory and hence allow the implementation on less number of cores. Having more cores budget gives more memory for higher precisions and more computation power for higher throughputs. Using Figure 4.5, a designer can decide on the optimal settings for executing CNN inference. If the designer wants to get a minimum throughout of 30 FPS on a budget of 500 cores, AlexNet with 8-bit and GoogleNet with 4-bit meet the selection criteria.

Figure 4.5: Estimated Performance of AlexNet and GoogLeNet for different precisions.
However, Epiphany uses floating-point operations and has no support for integer multiplications or low-precision operations. Therefore, if the cores are replaced by other cores that support low-precision operations, this method can achieve higher throughput values using the same number of cores.

4.2.2 Optimization of RNN models

This section presents the methods proposed in this thesis for RNNs optimizations. The optimization methods are applied to decrease RNNs computation and memory requirements. Quantization and delta methods are applied to different RNN models in a post-training approach.

• Post-training quantization of RNN models

As mentioned earlier, quantization is a method that reduces the precision of NN weights and activations to decrease their computational and memory needs. Post-training quantization is a more careful quantization technique that tries to keep the error within an accepted range without the need for retraining. This section presents a method for post-training quantization of 4 recurrent layers, LSTM, GRU, LiGRU, and SRU. Since the $M \times V$ multiplications are the bottleneck for computations in the recurrent layers, there is no promising benefit in the quantization of any other part in the layer. Thus, the weight matrices are quantized to integer precision while vectors’ weights are quantized to 16-bit fixed-point format. During inference, input and activation vectors are in a 16-bit fixed-point format. Then, the activation vector is quantized to the required integer precision before the $M \times V$ operations. Later, when the $M \times V$ is completed, the activation vector is requantized back to a 16-bit fixed-point as visualized in Figure 4.6. Linear quantization is used for the weights integer quantization, and the Minimum Mean Square Error (MMSE) method is used as a clipping method [57] during quantization. To requantize integer values back to 16-bit fixed-point format, the number of bits required to store the integer part is computed, and the remaining bits are used for the approximated fraction part.
Figure 4.6: Integer Matrix to Vector (M × V) multiplication in a recurrent layer gate. The input and the previous time-step output vectors are quantized to integer precision. Then, M × V multiplication is applied between integer vectors and integer weights. Afterwards, the vectors are re-quantized to a 16-bit fixed-point. If the gate is chosen to skip the h_{t−1} path quantization, the h_{t−1} vector is kept in a 16-bit fixed-point by skipping quantization, multiplied with fixed point weights, and requantization is skipped (turned-off).

Each gate in the LSTM, GRU, and LiGRU has two paths. One path for the input vector x_t and the other for the previous-time step output vector h_{t−1}. Running experiments with turning off quantization in some paths showed that some paths are more sensitive to quantization. In LSTM, the turned-off part is the candidate state computation h_{t−1} path. In GRU and LiGRU, the turned off part is the candidate output vector computation h_{t−1} path. In SRU, this turning-off is not applied as the SRU has no M × V applied to the h_{t−1} vector. Figure 4.6 shows an example of M × V computations in a gate during the integer quantization and how to turn off the quantization in the h_{t−1} path of the gate computation.

- **Applying Delta networks method on quantized RNN models**

  This section studies the effect of integer quantization on the delta networks method, as both are applied to activation vectors. Two quantization configurations have been selected, one with 8-bit activation vectors and the other with 4-bit activation vectors. The delta vectors are computed as described in Equation 2.18. Then, the histogram of the delta vectors is computed. A high percentage of zeros is found among the...
Figure 4.7: The potential of eliminated operations plotted vs. possible delta thresholds. In a delta vector, the eliminated operations potential is computed as the summed frequency of the delta threshold and smaller delta thresholds. Thus, if a value is selected as a delta threshold, the plot predicts the percentage of eliminated operations.

delta vectors values, as visualized in Figure 4.7. Therefore, if the delta networks method is applied with integer quantization, many computations can be eliminated without any extra increase in the error. Figure 4.7 also shows that if a delta threshold of a value greater than zero is used, more operations can be eliminated. However, this can increase the error level. Thus, the selection of the delta threshold needs to be cautious.

To find the delta threshold for each model at a specific activation precision, a combination of possible delta thresholds is used for the input, previous time-step output, and FC activation vector. The effect of each combination on the error and percentage of eliminated operations is evaluated by running inference while using a portion of the validation data. The combination of delta thresholds that can maximize the percentage of eliminated operations while keeping the error low is selected.
5. Flexibility in Deep Learning Models Realization on Embedded Platforms

This chapter covers the flexibility in DL implementations on embedded platforms from two perspectives. The first is the flexibility of the DL hardware implementations, and the second is the flexibility of the model optimization.

5.1 Flexibility of the DL hardware implementations

This section first presents the study of flexibility in RNN implementation on embedded platforms. Then, it discusses a method proposed for flexible mapping of CNN models on the Epiphany manycore architecture. Finally, it presents a set of features preferred to be present in a DL DSA.

5.1.1 Study of flexibility in RNN implementations on embedded platforms

The flexibility of an implementation solution is the ability of the solution to support various models and configurations. Models can be subjected to changes in the number of layers, the number of hidden cells in layers, and the optimization method and degree of optimization applied to each layer. Also, support of online training on the embedded platforms increases the implementation flexibility. However, it is not easy to quantify flexibility and measure it in units to compare different implementations. Thus, I defined flexibility levels and counted the percentage of implementations under study that supports these levels in Figure 5.1. Level 0 indicates no flexibility. It requires the implementation to support one recurrent layer setting only. Thus, all of the implementations under study meet the level 0 requirement. After that, implementations vary in meeting other flexibility levels. The flexibility aspects and how they can be met are discussed in Appendix A.

In Figure 5.1, it is observed that the variations in the recurrent layer are supported in about half of the implementations under study. Few implementations supported other NN layers, such as FC and convolution layers, and even
fewer implementations supported variations in the algorithmic optimization methods or training. None of the implementations supported variation in the application domain.

5.1.2 Flexible implementation of CNN on the Epiphany architecture

In Chapter 4, a method for mapping low precision CNN models on the Epiphany manycore architecture is presented [41]. The mapping method is also flexible by supporting variation in the architecture, the model, and the compression method. The method allows two different sizes for the local memory of epiphany cores (32KB and 64KB). Also, the method allows for two CNN models, AlexNet and GoogLeNet, with different precisions for the convolutions operations, as seen in Figure 5.2. The designer input the model type, the precision, and the memory size. Then, the architectural experiments module computes the number of cycles required for convolutions, the number of communications cycles, and the overhead for the network congestion. The CNN analysis module provides information about the CNN layers. Finally, the information supplied from the architectural experiments and CNN model modules is used in the estimation module. The final estimation module evaluates the minimum number of cores needed for the mapping and the corresponding throughput in terms of frames/sec.

Figure 5.1: Percentage of implementations supporting different flexibility levels. On the right, the definition of flexibility levels is presented.
Flexible architectures for DL models

Domain-specific architectures (DSAs), accelerators, or custom hardware are all names for hardware designed for a specific application domain. DSAs attract a significant research interest as a future opportunity in the computer architecture field [40]. Deep learning applications have known memory access patterns that enhance the use of memory hierarchy and parallelism in DL DSAs, as seen in Google edge TPU [90]. However, DL models are usually subjected to model optimization techniques that change these models’ weights precision, and memory access patterns. Thus, DL DSA should be flexible enough to support the optimized NN models with different types and degrees of optimization. The following list gives some examples of features preferred to be supported by such architectures.

- Variable bit-width operations as in the Bit fusion [91] to support various quantization configurations.
- Pre/post-processing on input vectors and weights as required by some optimizations.
  - Weights reordering used to utilize memory accesses.
– Delta networks method computations.
– Composing circulant matrices from equivalent vectors.

• Sparsity handling in input vectors for the delta-networks method.
• Training support that would imply the support of back-propagation to modify pre-trained weights.

5.2 Flexibility of RNN models optimization

This section presents the proposed methods for NN models’ flexible optimization. First, a method for Multi-Objective Hardware-Aware Quantization (MOHAQ) of RNN models is presented. Then, a comparison of different types of RNN models subjected to different levels of quantization and the delta-networks method under various memory and error constraints is discussed.

5.2.1 Multi-Objective Hardware-Aware Quantization (MOHAQ)

MOHAQ is a method for NN quantization [42]. The method searches for precision for the weights and activations for each layer in the model. The search is guided by the target hardware platform model equations. Thus, the search has two kinds of objectives. The first objective is the model error that should be kept low, and the hardware efficiency-related objectives such as throughput/speedup and energy consumption. MOHAQ gives the hardware designer a set of Pareto optimal solutions. Each solution is quantization precision for all layers’ weights and activations. The designer selects the most suitable trade-off solution for the target application. MOHAQ relies on a multi-objective Genetic Algorithm (GA) called NSGA-II [63] provided by a python library named Pymoo [92]. A genetic algorithm has been selected for this task as it is one of the efficient multi-objective search optimizers [60]. The NSGA-II algorithm showed the ability to find better convergence and better solutions spread near the actual Pareto-optimal front for many difficult test problems [62].

Usually, quantization affects the model error, and retraining has to be applied to decrease the error. Thus, if each solution in the search space requires retraining, the multi-objective search will be infeasible as it requires the evaluation of the error for each solution (quantization configuration). MOHAQ uses two techniques to do the search in a feasible time. First, post-training quantization is used as the quantization method. Post-training quantization methods do not require any training after the quantization of the pre-trained model. Thus, the evaluation of candidate solutions can be done in a reasonable time by running inference using a portion of the validation data. This technique is
called "inference-only search". However, post-training quantization success is not guaranteed in all scenarios. Thus, I extend the method with a novel technique named "beacon-based search". When a model is retrained using the quantization-aware training discussed in Chapter 2, the retrained parameters are in full precision format. Thus, post-training quantization can be applied with different quantization configurations to one retrained model. Experimental results show that the restrained model parameters decrease the error not only for the solution variables that have been used for retraining it but also for many other solutions. Thus, retraining the model for all candidate solutions quantization configurations is not required. It is enough to retrain the model a few times and use the retrained model parameters to predict the retraining effect on other solutions in the search space. Therefore, I called the retrained model a beacon as it guides the search when post-training quantization on the baseline model parameters is not successful. The steps of the MOHAQ method are visualized in Figure 5.3.
Figure 5.3: The Multi-Objective Hardware-Aware Quantization (MOHAQ) method steps. The input is the model’s pre-trained parameters, the hardware platform objectives equations, and hardware constraints. The designer can either apply inference-only search or beacon-based search. The beacon-based search uses a few retrained model parameters during the search to account for the retraining effect. The output of the MOHAQ method is a set of Pareto-optimal solutions. If the inference-only search solutions have high unaccepted error levels, the designer can apply the beacon-based method by repeating the search.

MOHAQ has been applied to two hardware platforms: SiLago and Bit fusion, using two SRU-based models and one LiGRU-based model. Figure 5.4 shows the result of applying MOHAQ to one SRU-based model for the Bit fusion architecture. The inference-only search Pareto-front is visualized in green, and the evaluated solutions are plotted in gray. After repeating the search using the beacon-based search, the Pareto-front is plotted in blue. The gap between the two Pareto sets in the figure is the outcome of applying retraining via beacons. The details of the Pareto sets are in Appendix C.
5.2.2 Comparison of RNN models under different quantization and delta method configurations

To achieve efficiency in RNN deployment on embedded platforms, new recurrent layers have been proposed with less computational and memory demands. On the other hand, algorithmic optimization methods are applied to the RNN model for the same reason. Chapter 4 explained how to apply the quantization and delta method to RNN models based on different recurrent layers. In this section, I compare these models under different quantization and delta threshold configurations against the models’ size (Figure 5.5) and the number of operations (Figure 5.6). The comparison helps in deciding which model is a more optimum choice under different constraints such as the size, the number of operations, and the error.

In Figure 5.5, at low error rates, the LSTM-based model gives the most optimal compressed models. At higher error levels, the SRU-based models provide the smallest size models. In Figure 5.6, the LSTM-based model provides optimal compressed models until 14.7% error rate. Then, the LiGRU-based model provides a lower count of non-eliminated operations. The SRU-based model decreases the operations count more at higher error rates. More analysis of these figures is found in Appendix D [44].
Figure 5.5: The size of compressed models at different error levels. For each model, the best points are connected using a line. Then, the best points of all models are highlighted with a gray shadow.

Figure 5.6: The number of operations in compressed models at different error levels. For each model, the best points are connected using a line. Then the best points of all models are highlighted with a gray shadow.
6. Summary of Contributions

This chapter provides a brief of each paper’s contributions.

6.1 Paper I: Recurrent Neural Networks: An Embedded Computing Perspective

This survey article defines the objectives and challenges of RNNs execution on embedded platforms. The article studies the components of the RNN models from a computer architecture perspective that focuses on the memory requirements and the number of computations. Then, the article divides the optimizations applied to RNN implementations into algorithmic and hardware-specific optimizations. Later, the article compares the state-of-the-art implementations of RNN models on embedded platforms against their throughput and energy consumption. The comparison includes the optimizations on each implementation to understand their impact in connection to the implementation objectives defined earlier. The comparison results were used to discuss how the defined objectives can be met and highlight some opportunities and research gaps in the field for future research. The contributions of the article can be summarized as follows:

- Compares RNN models’ components from a computer architecture point of view and presents the details of the computational and memory requirements.

- Summarizes the optimizations applied to RNN models implementation on embedded platforms.

- Compares RNNs’ recent implementations on embedded platforms in an application-independent manner.

- Points out potential opportunities for research in the field.
6.2 Paper II: Streaming Tiles: Flexible Implementation of Convolution Neural Networks Inference on Manycore Architectures

This paper proposes a method called "Streaming Tiles" for CNN implementation on manycore architectures. Epiphany manycore was used for implementation as a case study; however, it can be applied to other manycore architectures. The method has targeted both the implementation efficiency and flexibility objectives. The contributions of the method are summarized as follows:

- The proposal of a mapping method that eliminates the use of the off-chip memory by relying on the on-chip memory only for storing the weights.
- Pipelining the implementation to eliminate the need for loading the layers weights and storing intermediate results during execution.
- The method provides automated equations for mapping and load balancing to support two CNNs, two core memory sizes, and multiple precisions.

6.3 Paper III: MOHAQ: Multi-Objective Hardware-Aware Quantization of Recurrent Neural Networks

MOHAQ is a method that automates the selection of per-layer precision of a neural network for different platforms and application constraints using a multi-objective GA. The objectives of the optimization are the error rate and hardware-efficiency objectives such as speedup and energy efficiency. As a proof of concept, the MOHAQ method has been applied to SRU-based RNN models for speech recognition. The contributions of the method are summarized as follows:

- The methods allow the evaluation of candidate solutions for the multi-objective search in a feasible time by using post-training quantization.
- The method is extended to account for the retraining effect by proposing the "beacon-based search" that can predict the retraining effect on evaluated solutions by retraining the model a few times only.
- The method flexibility is demonstrated by applying it using two hardware models, Bit fusion and SiLago.
6.4 Paper IV: Shrink and Eliminate: A Study of Post-training Quantization and Repeated Operations Elimination in RNN Models

This paper proposes a method for applying a mix of quantization and delta network methods on RNN models. The method was evaluated on four RNN models in a post-training scenario. The contributions of the method can be summarized as follow:

- The method introduces "selected path skipping" during the recurrent unit quantization as a technique to decrease the post-training quantization error.
- The method highlights a positive synergic effect of activation vector integer quantization on the delta networks method.
- The method selects the delta thresholds for different RNN models for post-training inference.
- The method compares the four RNN models under different quantization and delta approximation configuration for the smallest size and highest percentage of eliminated operations.
7. Discussion and Limitations

The review in [1] intended to rank implementations of RNNs with respect to the efficiency objective of the thesis: high throughput and energy efficiency. The systematic review of algorithmic optimizations of DL models revealed that some of the reported approaches had not been implemented. For instance, the tensor decomposition and network distiller algorithmic optimizations have not been applied in any of the implementations under study. They have been proposed in papers that apply them to RNN models to decrease the models’ size and keep their high accuracy. Without finding them among the implementations, it was not possible to study their consequences for throughput and energy efficiency.

The algorithmic optimizations have been classified into three types in this thesis. A type that works on decreasing the number of weights by compression, a type that reduces the precision by quantization, and a type that eliminates operations by the delta networks method. Some prior research claimed that compression and quantization are orthogonal [10; 87]. Also, the shrink and eliminate method showed that the delta-networks method could benefit from quantization without an extra increase in the error [44]. Applying a mix of the three optimization types to the DL model can maximize the compression ratio and the percentage of eliminated operations within an acceptable accuracy range. Taking this into consideration while designing a DSA for DL applications, we should think of supporting the three types of optimizations. Also, variations in the degree of optimizations should be supported. For instance, Google edge TPU only supports 8-bit MAC operations [93]. If we want to keep some layers in fixed-point format due to their higher sensitivity to quantization, it will not be possible. Also, it is not possible to benefit from a lower precision in less sensitive layers. Chapter 5 gave an example of a set of features that could be supported on a DL DSA based on the study of the implementations. The block-circulant matrices support is included in these features as a compression method. In the study, block circulant matrices accelerators [34; 56] were more efficient than other compression-based accelerators such as pruning-based accelerators( [33]) due to the overhead of handling sparsity in the latter. It is an interesting question how beneficial it is to augment varying precision support to architectures such as ERNN [34] that supports varying block size block-circulant matrices and Sparten [79] that supports sparsity in
both activation vectors and weights. The trade-off with hardware efficiency has not been explored yet whether these three types of flexibility in algorithmic optimizations are supported. Each type would require extra hardware with extra cost, time, and area.

This thesis has applied some work to CNNs and others to RNNs. A few years ago, CNNs’ implementations were getting great attention in the embedded systems research community [29]. Later, RNNs started to get more research focus. However, they were less studied and used compared to CNNs. In addition, many algorithmic optimization methods treat the RNN models as CNN models by applying the same algorithmic optimization methods to them without considering their differences. Therefore, this thesis worked on methods that considered the temporal relationship between data in RNNs to investigate how it affects the optimization process. This thesis has conclusions on how quantization methods can change to perform better on RNN models and how the temporal relationship can help eliminate a high percentage of operations. The thesis conclusions can be invested in approaches that work with combined models of both RNNs and CNNs. Given that RNN can have a CNN as a feature extractor in the model, recurrent and convolution layers can both exist in one RNN model that works with temporally related spatial data.

Like CNNs, RNNs can be mapped into a 2D mesh of cores as done in the CNN mapping method [41]. The recurrent layers’ weights can be stored in the cores’ local memories, and the computation of RNN layers can be pipelined over groups of cores. Also, the MOHAQ method can be applied to CNNs such as RNNs. The MOHAQ method relies on the success of post-training quantization techniques. Even the beacon-based search idea is linked to post-training quantization, a.k.a; the idea of using the same model parameters for many quantization configurations with a minor effect on accuracy. And since post-training quantization methods are proven to be successful on CNNs, the MOHAQ method is also expected to be valid when applied to CNNs.

In the CNN mapping method [41], the hardware platform used was the Epiphany manycore architecture. Epiphany has been selected due to its energy efficiency. In addition, Epiphany supports up to 1024 cores. The large budget of cores enabled a complete pipeline implementation, where each layer is assigned to a group of cores, and the layer weights are loaded once with no need to reuse any core for another layer. However, the Epiphany cores support floating-point operations only and could not benefit from quantization computationally. These limitations raised the question of what is a suitable hardware architecture for deep learning. Thus, in the study of implementations, I investigated what kind of operations the DL application would require.

The main research focus in the MOHAQ method is how to enable the evaluation of candidate solutions error in a large search space, especially if
retraining is required [42]. SRU-based models have been used for two reasons. First, SRU is a promising recurrent unit that enables the parallelization of computations over multiple time steps by eliminating previous time step vectors from $M \times V$ operations and using them only in element-wise operations. Later, the SRU unit continued to evolve and included attention and was renamed SRU++ [94]. The second reason is the nature of the experiments used in this method. I wanted to apply retraining of solutions with many configurations to understand the nature of retrained quantized models and come up with the beacon idea and then understand how beacons are selected and define each beacon’s validity limits. A model that can be trained in a short time contributed to making the development of the method possible. The method has been evaluated on an LiGRU-based model to further show its validity in other kinds of models. However, the method has been applied to quantization only. It would be interesting to explore if the beacon-based method is still valid for other algorithmic optimization methods or not.

In addition, MOHAQ is a hardware-aware method. The hardware is used in the method as input objective functions that represent any hardware-related objectives, such as speedup, energy consumption, or area. The objective functions used were simple functions. For instance, the speedup function only considered the speedup of MAC operations. The quality of the proposed method is not affected by the simplicity of the equations, as the method is mainly trying to speed up the evaluation of the error objective. However, using hardware objective functions that consider more hardware components and overheads, such as the utilization of the systolic array and the frequency of the architecture, will provide a more reliable Pareto-front. Besides, I think that the modeling of PIM architectures to be used in HW-aware methods can be a promising direction, especially since, as discussed in Section 3, recent PIM architectures support varying precision in addition to sparsity. Thus, supporting such architectures in the MOHAQ method would have a combined benefit. First, it will show in the resulting Pareto-front how NN optimization differs on PIM architectures. Second, it gives an opportunity to apply the method to combined varying precision and sparsity.

Most of the post-training quantization methods in the literature (Chapter 3) work on removing the effect of the outlier values in the quantized vectors. Shrink and Eliminate [44] applied one of the clipping methods in the literature and added the skipping of selected paths from quantization to further decrease the error resulting from quantization (Chapter 4). These paths are selected empirically and found to be the recurrent paths in the candidate state or candidate output vector computation. No promising difference in the error is found when quantization is skipped in the gates computations. This behavior can be related to the role of gates and candidate vectors computations in recurrent units. The
gates only decide what to pass and what to prevent. Thus, a non-linear sigmoid function with an output range from 0 to 1 is suitable for the gates computation. The sigmoid gate second derivative is zero. Tanh and RELU are used in the candidate vectors computations as they better handle the vanishing gradient problem than the sigmoid function. This might have led the weights used in the candidate vectors computations to be more sensitive to quantization than the weights used for the gates computation.
8. Conclusion and Future Outlook

8.1 Conclusion

Deep Learning (DL) models have come to dominate many artificial intelligence applications. Realizing DL models on embedded processors would enable edge devices placed in cars, homes, wearable products, etc., to take advantage of these applications. However, the limited resources of embedded systems are in conflict with the DL models’ large appetite for computational power and memory. The aim of the thesis is to provide the embedded DL system designer with tools for design space exploration, helping to achieve an efficient implementation for a variety of scenarios. The studies and the proposed methods have the purpose of letting the implementations of DL models on embedded systems satisfy two objectives: implementation efficiency and flexibility.

For this thesis, an implementation is considered efficient if it achieves performance and energy efficiency. This thesis concludes that applying an algorithmic optimization method to the DL model is essential for an efficient implementation. In addition, at least one memory-specific optimization needs to be applied in order to avoid the high cost of frequent data loading from the off-chip memory. This thesis identified three algorithmic optimization methods, one from each of the types in the classification presented in the survey of RNN implementations, that proved to be highly beneficial. The three optimization methods are the block-circulant matrices method (which works on decreasing the number of weights), quantization (which reduces the precision), and the delta-networks method (which eliminates operations). A key result from the thesis is the experimental confirmation of the synergy of combining quantization and delta networks. Further, the thesis presents a way of applying quantization and selecting the delta-networks method threshold at low error rates when done in a post-training manner. Results show compression of 85% of the models’ understudy and the elimination of 50% of their operations with minor increases in the error levels.

The thesis discusses flexibility, the ability to support changes, from two different perspectives. The first perspective addresses the ability of the hard-
ware implementation to preserve efficiency gains achieved for an application when the model or the optimizations change. The second perspective addresses the ability of the algorithmic optimizations to keep producing optimally compressed versions of DL models under changes to the application constraints and the hardware platform. In exploring the first perspective, this thesis suggests a design space that includes a set of features that should be present in a domain-specific architecture for DL models. It also proposes parameters that can be chosen in order to configure how the convolution module of a CNN should be mapped onto the cores of a many-core architecture. This design space was tested by applying it to two CNN models using the Epiphany architecture, allowing for different precision in the models and two sizes of on-chip memory. The designer can use the design space exploration estimations to decide on the proper CNN model and precision depending on the budget of cores and the required throughput.

In exploring the second perspective, this thesis proposes methods that guide the designer in the selection of the algorithmic optimization parameters and the proper RNN model under different constraints. The thesis proposes a hardware-aware multi-objective quantization of RNN models. The optimization method considers both the error rate and hardware efficiency metrics as objectives. Each possible quantization configuration is considered a solution in the optimization search. At the end of the search, the embedded systems designer is given a Pareto-set of solutions with a trade-off between objectives. To enable the evaluation of the error of candidate solutions within a reasonable time, the method relied on two types of search. The first is called "inference-based search", where post-training quantization is used to avoid the time-consuming retraining of quantized solutions. To handle the situation when the inference-based search fails to provide low error solutions, the thesis introduces a novel search method called "beacon-based search". The beacon-based search considers the retraining effect by retraining a small set of solutions only. The method has been evaluated on three RNN models and two hardware architectures models. The beacon-based search managed to decrease the error rates by up to 4.9 percentage points compared to the inference-based search.

The thesis also conducted a comparison that guides the designer in the selection of the optimized RNN model under different constraints. The comparison is held between four RNN models subjected to different quantization and the delta networks method parameters. The comparison concluded that the LSTM-based model provides the highest compressed solutions at low error rates. At higher error rates, SRU-based models provide the smallest models size, and the LiGRU-based model provides the highest number of eliminated operations.
8.2 Future Outlook

To expand the work in this thesis, a study of all recent deep learning models and different application datasets from a compressibility point of view would be beneficial. This study would give insights into the main differences between different neural network layers and architectures when different algorithmic optimization methods are applied to them and when different datasets are used. So far, the methods proposed in this thesis work on decreasing the time for solutions evaluation to make the multi-objective search possible. However, if we want to extend the method to support all models, datasets, and possible algorithmic optimizations, a higher level of search space reduction should be involved. The search tool should gain some experience in making some selections before starting the search. Also, if minor changes happen, the tool should have the experience of finding new solutions within a minimal search. Thus, the high-level compressibility study and using some AI experience-based approaches, such as case-based reasoning, can make such tools powerful. Case-based reasoning solves new problems by adapting similar stored solved problems to satisfy the new requirements [95].

On the candidate solution evaluation level, it would be beneficial to extend the beacon-based search to support other compression methods such as block-circulant matrices and delta networks method. In addition, it would be interesting to apply the optimization methods proposed in this thesis to other recent NN models. For instance, the SRU-based models used extensively in this thesis can be replaced with SRU++-based models. SRU++ is a recent recurrent layer that has attention [94]. Thus, SRU++-based models are a promising direction as they have the same hardware-friendly characteristics as the SRU unit in addition to the attention. In the SRU++, the $M \times V$ applied to the input vector is replaced with the attention computation. The attention computation mainly involves a set of $M \times V$ computations, element-wise computations, and a non-linear computation. The same quantization approach used in the post-training quantization of the SRU in this thesis can be directly applied to the SRU++. Only the weights and activation vectors subjected to the $M \times V$ computations will be subjected to low precision integer quantization, and all other operations will be kept in 16-bit fixed-point operations. In addition, the percentage of eliminated operations in the quantized activation vectors should be studied to understand the potential of applying the delta networks method to quantized SRU++-based models.
References


S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally. Ese: efficient speech recognition engine with compressed LSTM on FPGA. CoRR, abs/1612.00694, 2016. 28, 51


C. Wittlinger, G. Spanakis, and G. Weiss. Flexible deep neural network structure with application to natural language processing. 11 2015. 6


N. M. Rezk, M. Purnaprajna, and Z. Ul-Abdin. Streaming tiles: flexible implementation of convolution neural networks inference on manycore architectures. In 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 867–876, May 2018. 11, 13, 40, 52


61


[51] T. Lei, Y. Zhang, and Y. Artzi. Training RNNs as fast as CNNs. CoRR, abs/1709.02755, 2017. 16, 18


63
Sparten: a 
sparse tensor accelerator for convolutional neural networks. 

Y. Chen, J. S. Emer, and V. Sze. 
CoRR, abs/1807.07928, 2018. 28

A configurable cloud-scale DNN processor for real-time AI. 
In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pages 1–14, June 2018. 28

J. H. Wilkinson. 
Rounding errors in algebraic processes. 

A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing. 

J. Kim, J. Lee, J. Lee, J. Heo, and J. Kim. 

Netadapt: platform-aware neural network adaptation for mobile applications. 
CoRR, abs/1804.03230, 2018. 29

Approximate FPGA-based 
LSTMs under computation time constraints. 
CoRR, abs/1801.02190, 2018. 29

Evolutionary multi-objective model compression for deep neural networks. 
IEEE Computational Intelligence Magazine, 16(3):10–21, 2021. 29, 30, 51

HW-flowq: a multi-abstraction level HW-CNN co-design quantization methodology. 

Appq: joint search for network architecture, pruning and quantization policy. 

Google edge TPU. 

CoRR, abs/1712.01507, 2017. 41

J. Blank and K. Deb. 
Pymoo: multi-objective optimization in python. 
IEEE Access, 8:89497–89509, 2020. 42

